0 – 7). Assume that a serial input at the SI reads a byte from a network and generates three types of interrupts. T generates timer-overflow and timer-capture interrupts, called TF and TCAPTURE interrupts. Worst-case interrupt latencies are as follows.

1. When the seventh byte is received, the controller generates interrupt FIFO_FULL and a FIFO_FULL flag sets in S0. Assume that it is the top priority serial interrupt and the ISR execution time is T_{exec} (FIFO_FULL). For the FIFO_FULL interrupt, the interrupt latency is $T_{switch} + T_{disable}$ because it is a top priority ISR.
2. When the zeroth byte is received, the SI generates an interrupt RI and an RI flag sets in the status register S0. Assume RI has the lowest priority serial interrupt and RI ISR execution time is T_{exec} (RI). For the RI interrupt, the interrupt latency is $T_{switch} + T_{exec}$ (TCAPTURE) + T_{exec} (TF) + $T_{disable}$, because it has the lowest priority than the timer interrupts.
3. When the third byte is received, the SI generates an interrupt FIFO_4thEntry and a FIFO_Half flag sets in S0. Assume that it is the middle priority serial interrupt and has priorities lower than the TCAPTURE interrupt but higher than the timer overflow. Assume that the ISR execution time is T_{exec} (FIFO_Half). For FIFO_4thEntry interrupt, the interrupt latency is $T_{switch} + T_{exec}$ (TCAPTURE) + $T_{disable}$, because it has higher priority than timer overflow but it has lower priority that TCAPTURE. The T_{exec} (RI) is not taken into account because if RI is not responded then only FIFO_Half interrupt occurs. Both interrupts RI and FIFO_Half belong to the same SI device.

Each running program when interrupts, the interrupting source service routine takes some time before starting the servicing codes. That time interval is called interrupt latency. It is the sum of the execution time of higher priority interrupts and the context switching period. If an interrupted routine is having a critical section (interrupts disabled), the interrupt latency increases by period equal to the interrupts disabled period.

4.6.2 Interrupt Service Deadline

For every source, the service of its ISR instructions can be kept pending up to a maximum period. This period defines the deadline during which the service must be completed. It should not be less than the worst-case interrupt latency. Figure 4.12(a) shows interrupt latency period and deadline for an interrupt.

A 16-bit timer device on overflow raises TF interrupt on transition of counts from 0xFFFF to 0x0000. It has to be responded by executing an ISR for TF before the next overflow of the timer occurs, else the counting period between 0x0000 after overflow and 0x0000 after the next-to-next overflow will not be accounted. The timer counts increment every 1 μs; the interrupt service deadline is 65536 μs.

Video frames in video conferencing reach after every 1 ÷ 15s. The device on getting the frame interrupts the system and the interrupt service deadline is 1 ÷ 15s, else the next frame will be missed.

Example 4.15 FIFO_Full interrupt must be executed fast as it has shorter deadline compared with RI and the fourth entry interrupt. If ISR for FIFO_Full interrupt does not execute before the next character at the SI device, the character will be missed. If ISR for FIFO_4th entry interrupt does not execute fast, it does not matter, because eventually there is a cushion of SI raising the FIFO_Full interrupt. If ISR for RI interrupt does not execute fast, it does not matter, because eventually there is a cushion of SI raising FIFO_4th entry interrupt as well as FIFO_Full interrupt. FIFO_Full interrupt is said to have a service interrupt service deadline. If SI device is receiving characters at 64 kbps and in 11-bit UART format, the FIFO_Full interrupt service deadline is 171.9 μs. FIFO_RI interrupt service deadline is 171.9 μs if SI device does not have the buffer and provisions for FIFO_Half and FIFO_full interrupts.

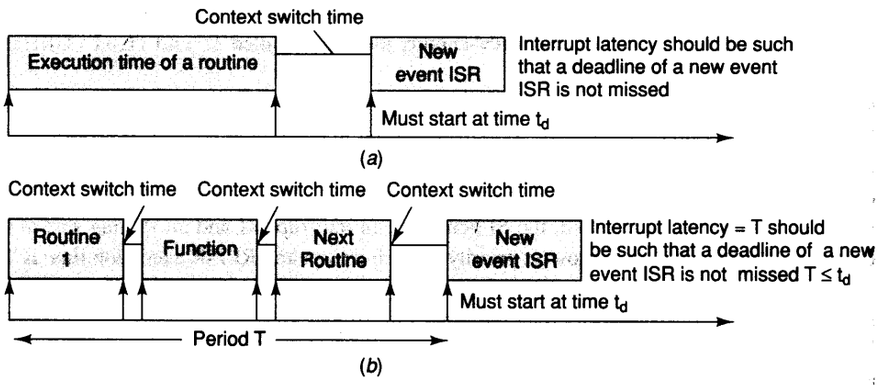


Fig. 4.12 (*a*) Interrupt latency period and deadline for an interrupt (*b*) Short interrupt service routines (ISR) and functions, which run at later instances so that the other ISR deadlines are not missed

A good software design principle for multiple interrupt sources is to keep the ISR as short as possible. Why? This is service the in-between pending interrupts and leave the functions that can be executed afterwards for a later time. When this principle is not adhered to, a specific interrupting source may not be serviced within the deadline (maximum permissible pending time). Section 4.2.3 described use of interrupt service threads, which are the second-level interrupt handlers. Figure 4.12(b) shows a short ISR and functions, which run at later instances so that the other ISR deadlines are not missed.

The system therefore has to meet the deadlines set for service of each system device. This can be understood by the following examples. Consider the example of a video system. When the system is running, two device-driver ISRs also run. One driver is for the voice device and the other for the image device. The ISRs and the other system software design for these two device drivers have to maintain synchronization else the next set of images and the next set of voice signals will be missed.

Therefore, the system software designer designs the appropriate ISRs for multiple device interrupts so that all device interrupt calls are serviced within the stipulated deadlines of each interrupt. The design should provide optimum latencies and set appropriate deadlines for each service routine and functions.

Each ISR may have a interrupt service deadline when interrupts. An ISR with a deadline must have interrupt latency less than the deadline.

4.6.3 Software Over-riding of Hardware Priorities to Meet Service Deadlines

Which source or source group has higher priority with respect to the others that is first decided among the ISRs that have been assigned higher priority in the user software. If user-assigned priorities are equal then the highest priority is that which is preassigned at the processor internal hardware. The 8051 internal interrupt mechanism is as follows. There is the interrupt priority (IP) register at 8051 in which there are five priority bits for the five interrupt sources in 8051. Also there are the five interrupt-enable bits in the IE register. These are secondary-level enable bits of the processor's service of ISRs. When a

priority bit at IP is set, the corresponding interrupt source gets a high priority, and if reset, it gets a lower priority. The 8051 first selects by polling among the high priority according to the bits at IP register.

There is a need for over-riding the priority order by assigning priorities. The need of reassigning priorities over hardware pre-assigned priorities can be understood from the following example.

Example 4.16

Assume that there are two sources of interrupts: serial port input and A/D conversion. A/D conversion time is 200 μs and SI device with no data buffer receiving inputs at 64 kbps with minimum separation between the characters equals 171.9 μs. The A/D conversion should therefore have the lower priority than RI interrupts of SI. When the system hardware has the internal devices, it assigns lower priority to the A/D end of the conversion interrupt. Suppose that the SI device is used to receive input at 16 kpbs and assume that the UART mode has 11 bits per character. When the A/D conversion is needed continuously (e.g., when ECG signals are input), the software should assign the higher priority to A/D, because SI receives character every 11/16 ms = 687 μs and A/D every 200 μs, at a rate faster than the SI.

Software-assigned priorities can be used to over-ride the hardware priorities. OS provides the functions, which assign the software priorities to each ISR, IST and task of the real-time system.

4.7 CLASSIFICATION OF PROCESSORS INTERRUPT SERVICE MECHANISM FROM CONTEXT-SAVING ANGLE

1. The 8051 interrupt service mechanism is such that on occurrence of an interrupt service, the processor pushes the processor registers PCH (program counter higher byte) and PCL (program counter lower byte) onto the memory stack. The 8051 family processors do not save the context of the program (other than the absolutely essential PC) and a context can save only by using the specific set of instructions in the called routine. For example, using push instructions. It speeds up the *start* of ISR and *return* from ISR but at a cost. The onus of context saving is on the programmer in case the context (SP and CPU registers other than PCL and PCH) is to be modified on service or on function calls during execution of the remaining ISR instructions.
2. The 68HC11 interrupt mechanism is such that processor registers save onto the stack whenever an interrupt service occurs. These are in the order of PCL, PCH, IYL, IYH, IXL, IXH, ACCA, ACCB and CCR. The 68HC11 thus does automatically save the processor context of the program without being so instructed in the user program. As context saving takes processor time, it slows a little the *start* of ISR and *return* from the ISR but at the great advantage that the onus of context saving is not on the programmer and there is no risk in case the context modifies on service or function calls.
3. Certain processor provides for fast context switching two stack frames with each stack frame consisting of the same number of registers, for example, 16 or 32 registers. The PC, stack-pointer and link-register define one stack frame. When context switches from one routine to another, only the pointer to the stack frame changes. The ISR stack frame that is called has the current program context and the interrupted program context becomes the saved program context. ARM7 provides such a mechanism. Certain processors provide for more than two stack frames with each stacking a context.

The OS program also provides for memory blocks, which are used as multiple stack frames for the tasks (processes or threads). This enables multi-threading and multi-tasking.

Certain processors provide for saving only the PC. Certain processors provide for saving only the PC and other CPU registers before calling the ISR and context switching. Certain processors provide for fast context switching by providing internal register frames for the stack or providing sets of local (internal) stack for the contexts. Fast context switching reduces the interrupt latencies and enables the meeting of each function or routine deadline for service. The operating system provides for multiple stack frames to enable multitasking and context switching using the multiple stack frames.

4.8 DIRECT MEMORY ACCESS

Assume that the data transfer is to occur between hard disk system memory. The DMA is used in that case. When the I/Os are needed for large amount data from a peripheral device to the memory addresses in the system or large amount of data is to be transferred by the I/Os, the interrupt-based mechanism is not suitable.

A DMA facilitates a multi-byte data set or a burst of data or a block of data transfer between the external device and system or between two systems. A device that facilitates DMA transfer has a processing element (single purpose processor). The device is called DMAC (DMA Controller). Data transfer occurs efficiently between the I/O devices and system memory with the least processor intervention when using DMAC. The system address and data buses become unavailable to processor and available to the IO device that interconnects using DMAC during the data transfer. Figure 4.13 shows the interconnections using the DMAC. It also shows the buses and control signals between the processor, memory, DMAC and data-transferring I/O device.

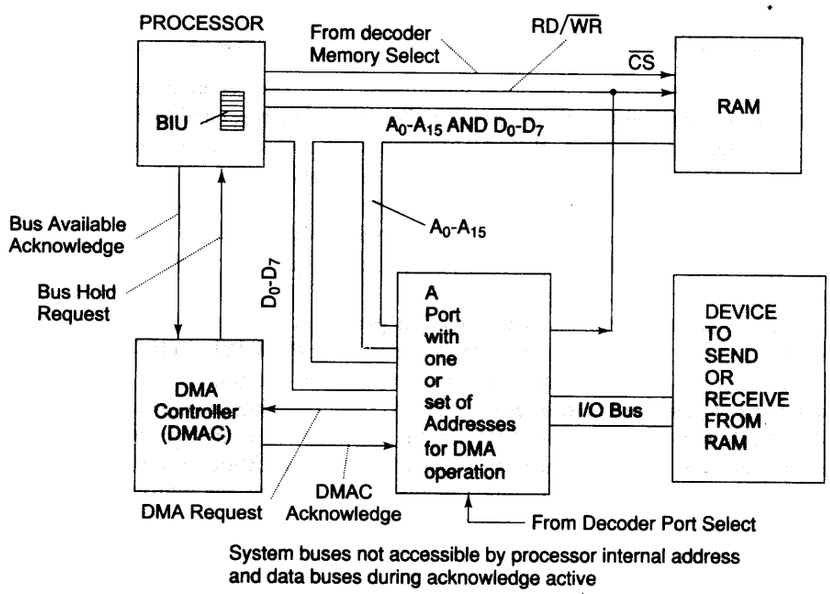


Fig. 4.13 The buses and control signals between the processor, memory, DMA controller and data-transferring I/O device

The DMAC sends a hold request to the CPU and the CPU acknowledges that if the system memory buses are free to use. Three modes are usually supported in DMA operations. (i) Single transfer at a time and then

release IO bus hold on the system bus after each transfer. (ii) Burst transfer at a time and then release of the IO bus hold on the system bus. A burst may be of a few kilobytes. (iii) Bulk transfer and then release of the IO bus hold on the system bus only after the transfer is completed.

4.8.1 Use of DMAC

Whenever a DMA request is made to the DMAC for the I/Os, the DMAC is first initialized. It is programmed for (i) read or write, (ii) mode (bytes, burst or bulk) of DMA transfer, (iii) total number of bytes to be transferred and (iv) starting memory address. Consider a read operation (external device to memory transfer). DMA proceeds without the intervention of the CPU, except (i) at the start of DMAC programming and initializing and (ii) at the end. Whenever a DMA request by the external device is made to the DMAC, the CPU is requested the DMA transfer by DMAC at the start to initiate the DMA and at the end to notify the end of the DMA by DMAC. Example 4.16 explains the data transfer operation.

Example 4.17

Assume that 2 kb of data needs to be transferred. One method is that device interrupts the processor, when 1 or 4 or 8 bytes of data are ready and generate the interrupt. The ISR reads the 1 or 4 or 8 bytes and put these into the memory addresses. Assume that the device generates the interrupt and transfers the 8 bytes. Number of interrupts required will be 2 k/8 = 2048/8 = 256 and ISR has to be run 256 times. A DMA is the better approach.

An IO program initializes the DMAC for 2 kb burst mode transfer from a memory address for the I/O to an external device starting from memory address M_1. DMAC loads 2048 in a data count register and loads M_1 in address register on initialization.

On an external device requesting the DMA, the DMAC sends HOLD request signal to the processor. Processor acknowledges by the HLDA (hold acknowledge) signal that when the system buses are not in use.

DMAC transfers the bytes from I/O bus to the memory bus in burst from IO bus to the memory bus D0-D7 lines and keeps track of the data counts in the DC (data count) register. Transfer takes place to addresses from M_1 to M_1 + 2047. DC = 0 after the transfer completes.

DMAC interrupts the processor so that the processor is notified at the end of DMA transfer and an ISR can re-initialize the DMAC for the next transfer.

A DMAC may also provide memory access to multiple channels. A multi-channel DMAC provides DMA action from system memories and two (or more IO) devices. There is a separate set of registers for programming each channel. There may be the separate or common interrupt signals in the case of multi-channel DMAC.

The 80x86 processors do not have on-chip DMAC units. The 8051 family member 83C152JA (and its sister JB, JC and JD versions) have two DMA channels on-chip. The 80196KC has a PTS (peripheral transactions server) that supports DMA functions. (Only single and bulk transfer modes are supported, not the burst transfer mode.) The MC68340 microcontroller has two on-chip DMA channels. 80960CA has four-channel DMAC on chip, with a mode called demand transfer mode also provided.

On-chip or a separate DMAC facilitates fast direct byte transfers between memory and I/O devices compared with interrupt-driven data transfer as that has in-built processing element and uses the system buses as and when they are made available by the processor. Designers can use DMAC in sophisticated systems so that the system performance improves by separate processing of bulk or burst data transfer from and to the peripherals.

4.8.2 Use of DMA Channel in Case of Multiple Interrupts in Quick Succession from the Same Source

A good feature of DMA-based data transfer service is very small latency periods compared with data transfer using multiple IO interrupt sources and multi-byte bulk or burst data transfers. The interrupt service routine period from start to end can now be very small as the ISR that initiates the DMA to the interrupting source, simply programs the DMA registers for the command, data count, memory block address and I/O bus start address (Section 4.8.1).

The use of DMA channels for the IO services in place of processor interrupt-driven ISRs provides an efficient method when the device has to transfer large amount of data by I/O. This is because a DMA transfer uses the periods when the system buses are free.

4.9 DEVICE DRIVER PROGRAMMING

A system has number of physical devices (Chapter 3). A device may have multiple functions. Each device function requires a driver. Examples of multiple functions in a device are as follows.

1. A timer device performs timing functions as well as counting functions. It also performs the delay function and periodic system calls.
2. A *transceiver* device transmits as well as receives It may not be just a repeater. It may also do the *jabber control* and *collision control*. (Jabber control means prevention of continuous streams of unnecessary bytes in case of system fault. Collision control means that it must first sense the network bus availability then only transmit.)
3. Voice-data-fax modem device has transmitting as well as receiving functions for voice, fax as well as data.

A common driver or separate drivers for each device function are required. Device drivers and their corresponding ISRs are the important routines in most systems. The driver has following features.

1. *The driver provides a software layer (Interface) between the application and actual device*: When running an application, the devices are used. A driver provides a routine that facilitates the use of a device function in the application. For example, an application for mailing generates a stream of bytes. These are to be sent through a network driver card after packing the stream messages as per the protocol used in the various layers, for example, TCP/IP. The network driver routine will provide the software layer between the application and network for using the network interface card (device).
2. *The driver facilitates the use of a device by executing an ISR*: The driver function is usually written in such a manner that it can be used like a black box by an application developer. Simple commands from a task or function can then drive the device. Once a driver function is available for writing the codes, the application developer does not need to know anything about the mechanism, addresses, registers, bits and flags used by the device. For example, consider a case when the system clock is to be set to tick every 10,000 μs (100 times each second). The user application simply makes a call to an OS function like OS_Ticks (100). It is not necessary for the user of this function to know which timer device will perform it. What are the addresses, which will be used by the driver? Which will be the device register where value 100 registers for the ticks? What are the control bits that will be set or reset? OS_Ticks (100) when run, simply interrupts the system and executes the SWI instruction which calls the signalled routine (driver ISR) for the system ticking device. Then the driver ISR which executes takes 100 as *input* and

configures the real time clock (Section 3.8) to let the system clock tick each 10,000 μs and generate the system clock interrupts continuously every 10,000 μs to get 100 ticks each second.

Generic device driver functions in high level language are used in high level language program. The functions are open, close, read, write, listen, accept etc.

Device driver ISR programming in assembly needs an understanding of the processor, system and IO buses and the addresses of the device registers in the specific hardware. It needs in-depth understanding of how the software application program will seek the device data or write into the device data and what is the platform. Platform means the operating system and hardware, which interfaces with the system buses.

A common method of using the drivers is as follows: a device (or device function module) is opened (or registered or attached) before using the driver. If means device is first initialized and configured by setting and resetting the control bits of device control register and use of the interrupt service is enabled. Using a user function or an OS function, a device (or device function module) can also be closed or de-registered or detached by another process. After executing that process, the device driver is not accessible till the device is re-opened (re-registered or re-attached).

4.9.1 Writing Physical Device-Driving ISRs in a System

For writing the software for driver in assembly, the following points must be clear.

1. Information about how the device communicates.
2. Information about the three sets of device registers–data registers or buffers, control registers and status registers. A device *initializes* (configures, registers, attaches) by setting the control register bits. A device *closes* (resets, de-registers, detaches) by resetting the control register bits. (Example 4.18)
3. Information of other registers and common addresses to a device register.
4. Control register bits control all actions of the device. A control bit can even control which address corresponds to which data register at an instant. For example, at the instance when the DLAB control bit is set, the 0x2F8 corresponds to the divisor-latch lower byte (Example 4.19).
5. Status register bits reflect the flags for status of the device at an instant and change after performing actions as per the device driver. A status flag at a status register reflects the present status of device. For example, an instance between finishing the transmission of bits from a TRH buffer register and obtaining the new bits for next transmission, a transmitter empty flag (TDRE) reflects it (Example 4.19).
6. Either setting of an enable bit (interrupt control flag) is used by the system to initiate a call for executing an ISR related to the device driver function. ISR executes if: (i) it is enabled (not masked at the system) and (ii) the interrupt system itself is also enabled.

The following information must therefore be available when writing a device control and configuring and driver codes.

1. *Addresses for each register*: Physical device hardware and its interfacing circuit fix the addresses for a physical device and they usually cannot be relocated. The device becomes the *owner* of these addresses. For example, IBM PC hardware is designed such that the device addresses are as following:
   a. *Timer* addresses between 0x0040-5F;
   b. *Keyboard* addresses between 0x00600-6FD, real-time clock (system clock) addresses between 0x0070-7F;
   c. *Serial COM port 2* addresses between 0x02F8-2F and *serial COM port 1* addresses between 0x03F8-3F.
2. There may be input-buffer register as well as output-buffer register at a common address. This is because during device write and read instructions at the control bus the different signals RD and WR

are issued. The physical device can thus select the appropriate register when taking action. For example, there is a register SBUF at 8051. It addresses both the output serial buffer and input serial buffer.

3. There may be multiple registers at the same address. Refer Example 4.19. This example shows the following. RBR (receiver data buffer register) and TRH (transmitter holding register) are at the same address (0x2F8) in PC COM2 serial device. This address is also common for the lower byte of divisor latch, which is used for presetting the device baud rate. A control bit is made 1 to write this byte when setting the device baud rate and later it is made 0 for using the same address as RBR and TRH during the device *'read'* and *'write'* instructions, respectively.
4. Purpose of each bit of the control register.
5. Purpose of each status flag in the status register. Which status bit when set and reflects a device interrupt, calls to which ISR.
6. Whether control bits and status flags are at the same address. The processor reads the status from this address during the read instructions. The processor writes the control bits at that address during the write instructions.
7. Whether both, control bits and flags coexist in the same register.
8. Whether the status flag, which sets on a device interrupt, auto-resets on executing the ISR or if an ISR instruction should reset.
9. Whether control bits need to be changed, reset or set again before return to the interrupted process.
10. List of actions required by the driver at the data buffers, control registers and status registers.

Section 8.6.1 will describe in detail the device management functions at an OS. The OS usually provides device-related functions so that for the new device also the drivers are written in an identical manner. For example, Unix device driver components are: (i) device ISR, (ii) device initialization codes (codes for configuring device control registers) and (iii) system initialization codes, which run just after the system resets (at bootstrapping). Microsoft OS Windows provides the Windows driver functions (WDF) and user-mode driver framework (UDMF). Linux provides device drivers (Section 4.9.6). Using object-oriented programming approach, *Class drivers* are also written for operation on large number of similar type of devices using identical bus or network protocol, for example, printers or CD drives or class drivers for the USB-based devices.

When a device driver function such as read or write or open is called, the OS first intiates the logical layer part. The logical layer then initiates the physical layer, which implements OS device function using driver ISR functions written in assembly so that the device hardware performs the actions accordingly. Similary the device sends the response of the commands to the logical layer of the driver through the physical layer.

The device drivers execute according to the device hardware, interrupt service mechanism, OS, system and IO buses. A device driving ISR is designed using the device addresses and three sets of device registers– data registe(s) or buffer(s), control register(s) and status register(s). A device is configured and controlled by the control bits. The driver ISR initiates and executes on status flag change. A list of actions required by the driver at the data buffers, control registers and status registers is needed and is prepared before writing the driver codes. The driver codes are sensitive to the processor and memory. This is because: (i) when the device addresses change, the program should also be also be modified and (ii) when a processor changes, the interrupt service mechanism changes. The OS usually provides device drivers for the system devices.

4.9.2 Virtual Device Drivers

Virtual device drivers emulate the device hardware, for example, hard disk and generate software interrupts similar to physical device drivers. The file and pipe are two examples of virtual devices.

Both virtual devices and physical device drivers have functions for device *open*, *read*, *write* and *close*. Consider the analogies of a file device with a physical device. (i) Just as a *file* needs to be *opened* to enable

read and write operations, a device may need to be sent an *interrupt call* for initializing and configuring it (opening, registering or attaching it. Setting control bits appropriately does this). (ii) Just as a file is sent a *read call,* a device must be sent another *interrupt call* when its input buffer(s) is to be read. (iii) Just as a file is sent a *write call,* a device needs to be sent *another interrupt call when its output buffer is to be written.* (iv) Just as a file is sent a *close-call* a device needs to be sent *another interrupt call to disable* (close or deregister or detach) it from the system for further read and write operations.

The concept of virtual (software) device drivers is very important in programming. Examples are as follows.

1. A memory block can have data buffers in analogy to buffers at an IO device and can be accessed from a *char* driver or a *block* driver. The device is called the *char* device or the *block* device when it can access a character or a block of characters, respectively.
2. A physical device transceiver (with input–output block buffer) or repeater is equivalent to a virtual device called *loop back device.* It stores allocated memory blocks using a block device driver and returns the data back from the memory.
3. A bounded buffer device in memory can be like a printer buffer. A data stream is sent by one routine (driver) and read by another routine (driver). Bounded buffer device is a virtual device, usually called *pipe* device.
4. A program can store in a set of memory blocks called *RAM disk* in the analogous way a file system does at the hard disk. RAM disk is a device that consists of multiple internal file devices.

The virtual device is an innovative concept for system software design. Drivers for these are also written like the physical device drivers. Important devices are char device, block device, loop back device, file device, pipe, socket and RAM disk. Device configuring is equivalent to creating a file. Device activation on the interrupt is equivalent to opening a file. Device resetting is equivalent to closing a file. Device detaching is equivalent to freeing the memory space allotted for a file data.

4.9.3 Parallel Port Drivers in a System

Device driver *read* ( ) function can be implemented by calling an ISR is Port_ISR_Input, which handle the port input. Figure 4.14(a) shows control and status bits used in the ISRs in the device drivers and port pins interface with the data bus. Figure 4.14(b) shows step *A* for port initialization, step *B* for calling the driver and steps 0 to 5 for driver Port_ISR_Input. The driver reads byte from port and puts it into a queue that builds in memory on successive inputs to the port.

Port_ISR_Input does the following:

1. Step *A* sets the device control bit for read. Step *B* is no action till input event.
2. Steps 0 to 2 are for reading the input buffer(s) by emptying the buffer and storing the byte(s) in memory or using the bytes received as per the system requirement.
3. Step 3 resets the device receive-buffer ready flag (in status register) and thus prepares the device for the next read after step 4. In step 4, interrupt flag resets to enable next byte read on next interrupt.

An example for device driver *write* ( ) function is a driver ISR for handling the port outputs. The ISR does the following:

1. Sets the device control bit for write.
2. Sends into the device output buffer (s) the byte(s) from the memory.
3. Resets the device-transmit buffer-empty flag (in status register) on completion of transmission of the byte(s) and prepare the device for the next write.

Example 4.18 gives a device driver ISR example using 68HC11 microcontroller port C (68HC11 microcontroller knowledge is presumed here).

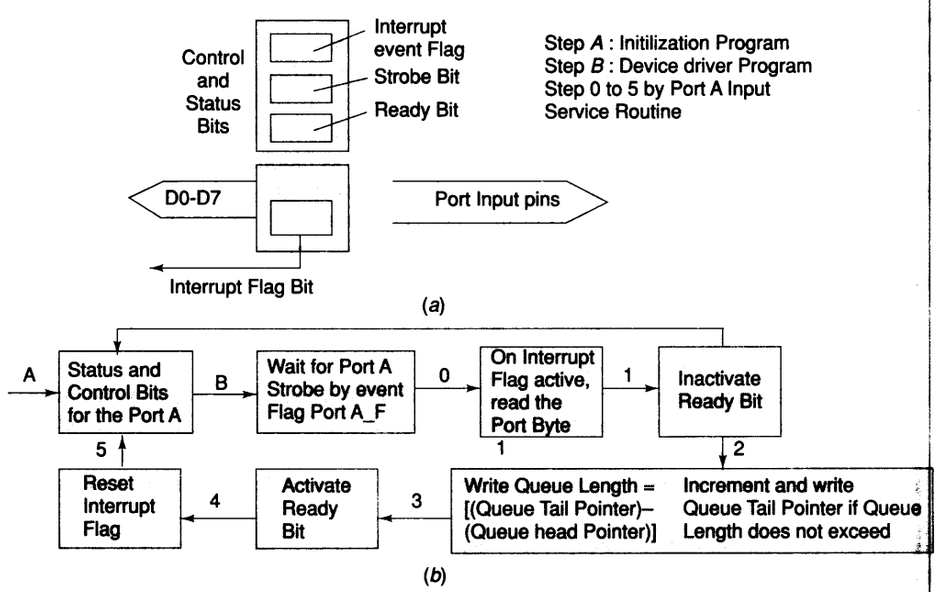


Fig. 4.14 (*a*) Control and status bits used in the interrupt service routines (ISRs) called by the device drivers and port pins used to interface the data bus (*b*) Step *A* for initialization, step *B* for the interrupt of the driver and steps 0 to 5 for driver Port_ISR_Input execution. The driver reads a byte from a port and puts it into a queue that builds in memory on successive inputs to the port

Example 4.18

Device driver *read* ( ) function calls an ISR PortC_ISR for handling the port C inputs in 68HC11. Port C uses the hand-shaking signals. Figure 4.15(a) shows hand-shaking signal to port C. Figure 4.15(b) shows control and status bits used in the *call* to the driver. Figure 4.15(c) shows port C as input and its interface with the data bus. Figure 4.15(d) shows port C as output. Figure 4.15(e) shows step *A* for initialization, step *B* of the interrupt and step *C* for executing PortC_ISR. The ISR reads from the port and inserts the byte into a queue. The latter builds in memory on successive inputs to the port. An external peripheral activates STRA pin. The peripheral requests a transfer of its byte to port C through the STRA. When STRA pin activates by '0', the port C gives an acknowledgment in case *STAI* (STRA interrupt mask bit) at a control register is not set (*STAI is not at* '1'). STRB pin sends hardware signal for the ready status (or acknowledgement) from the port C to the peripheral. When *STAI* is programmed to '0', the peripheral puts the byte into the port buffer as soon as STRB pin sends acknowledgement. As soon as the peripheral completes by putting the byte at the port C, the *STAF* sets (=0). *STAF* is at status register. *STAF* is the interrupt flag, which sets when the external device completes putting the byte at the port C.

Port C memory address is 0xp003, when page address configured on 68HC11 is 0xp000 (p is 4-bit maximum-significant nibble). A call to the device driver ISR for port C device *open* ( ), three actions occur by the device initialization program. (i) Define port C address as follows. # define PortC 0x1003 /* p bits as 0001 */. (ii) Reset all eight bits to 0s at DDRC so that port C becomes an input parallel port. DDRC is data direction

register for port C at memory address 0xp007. (iii) On initialization call, *STAI* sets to '1' for enabling interrupting by the peripheral, which connects to port C. *STAI* is the sixth bit of PIOC (port I/O control register). It is at memory address 0xp002.

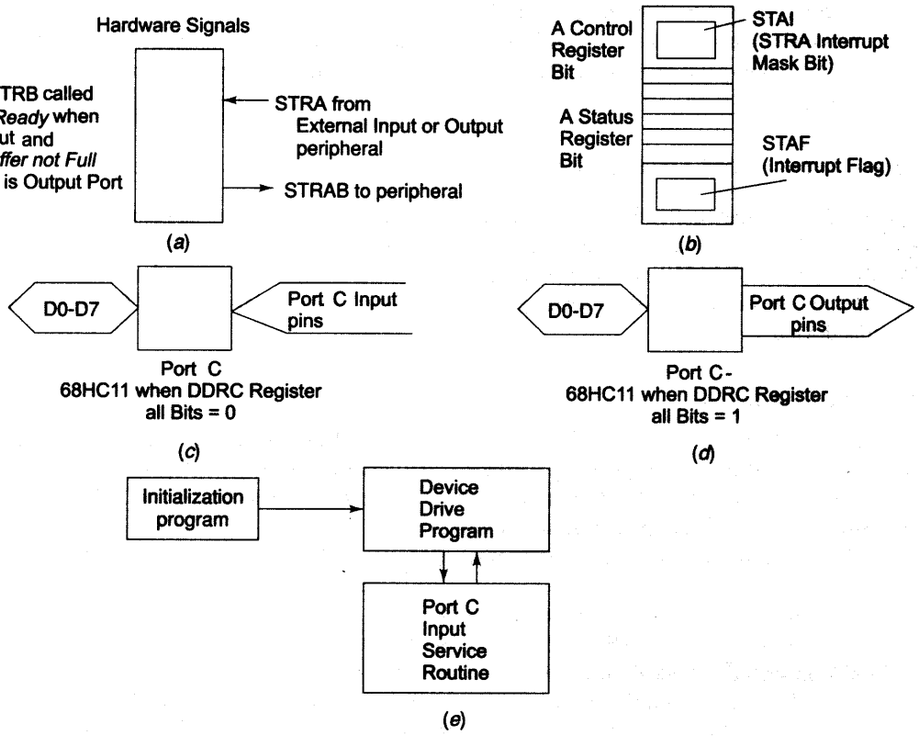


Fig. 4.15 (*a*) Hand-shaking signal to a parallel port (*b*) Control and status bits used in the system calls by driver functions (*c*) Port C as input and its interface with data bus (*d*) Port C as output (*e*) Step *A* for initialization, step *B* for system call to the driver and step *C* for driver PortC_ISR

A driver ISR program for Port C read will execute after the following actions.

1. If *STAI* is set '0' then read *STAF*. (*STAF* is the seventh bit at PIOC. PIOC also provides the status bits. It is for control cum status bits.)
2. If *STAF* is set '0' then interrupt for call to *portC_ISR* (port C service routine), otherwise wait.
3. There is no need to software reset *STAF* as there is automatic hardware reset of it by 68HC11 as soon as *portC_ISR* is called.

Driver routine *portC_ISR* programming is done as follows. Assume the name of pointers and variables are as following: (i) **portC_Queueback* is a pointer that points to a memory address where the byte from port C inserts into to a queue. (ii) *portC_Queuelength* is present queue length. (iii) *portC_Max QueueSize* is the maximum queue length defined for the port C received bytes.

1. If *quasi_bidir* bit does not equal to false, write 0xFF to port C.
2. Read port C.
3. If *portC_Queuelength* is less than the *portC_MaxQueueSize* store port bits at the address defined by **portC_Queuetail*.
4. If *portC_Queuelength* is not equal to *portC_MaxQueueSize* then increment **portC_Queuetail* to let it now point to next address. When equal then call an exception (*error* routine) for port C.

4.9.4 Serial Port Drivers in a System

There is IEEE standard called POSIX (portable operating system interface) standard. Portability of the UART drivers in different systems is essential. In a PC with 80x86 processor an UART 8250 or a new generation UART device UART 16550, which includes the 16 byte FIFO input and output buffer is used. Example 4.19 gives all three sets of the registers (data, control and status) for a serial-line UART device in a PC. All PCs have this device.

Example 4.19

A serial-line device 8250 or 16550 in a 80x86-based IBM PC has the addresses of device registers as follows. These addresses are fixed by hardware configuration of the UART port interface circuit in IBM PC system employing the 80x86 processor. They are from 0x2F8 to 0x2FE at COM2 port in a PC and 0x3F8 to 0x3FE at COM1 port. Consider COM 2.

1. Two I/O data buffer registers (RBR for receiving and TRH for transmitting) are at a common address, 0x2F8—
   (a) Provided a control bit at address 0x2FB is 0, (i) during read from the address, the processor accesses from the RBR or (ii) during write to the address, processor accesses the TRH.
   (b) Provided a control bit at address 0x2FB is 1, data of two bytes of *divisor latch* are at distinct addresses, 0x2F8 (LSB) and 0x2F9 (MSB). Divisor latch holds a 16-bit value for dividing the system clock. This then selects the rate of serial transmission of bits at the serial line. While writing a device driver, remember that a bit in another register (control register) changes the 0x2F8 access from access to the IO register to the lower byte register at divisor latch register.
2. Three control registers are at three distinct addresses 0x2FA, 0x2FB and 0x2FC. These are for writing in registers as follows—
   (a) IER (interrupt-enabling register). It enables the device interrupts.
   (b) LCR (line control register). It defines how and how many bits will be on the line.
   (c) MCR (modem control register). It defines how the modem handshakes and communicates.
3. Three status registers of the device are also at three addresses 0x2FA, 0x2FD and 0x2FE and are used during read from these. These are as follows—
   (a) IIR (interrupt identification register) for flags at 0x2FA. A flag sets on a device interrupt and resets at the servicing of corresponding device interrupt.
   (b) LSR at 0x2FD. It is for reading line status the number of bits that will be present on the line.
   (c) MSR at 0x2FE. It specifies the modem status bits during handshakes and communication.

Assume that device has been given identity number 5. It is also a file descriptor for the device that points to description parameters of the device.

(a) A serial device high-level driver function, open (5, baudrate) will configure and initialize the device. It sets the reset flag in IIR. The device initializes by unmasking the device interrupts and writing the control bits for clock divisor latch at the specified address. Divisor latch bits will define the baud rate configured for the device.

(b) A serial device high-level driver function *write* (5, *length*1, *memTxaddress*) will send bytes into TRH one by one and the device transmits total bytes = *length*1 from addresses memTxAddress to memTxAddress + *length*1 – 1.

(c) A serial device high-level driver function *read* (5, len2, memRxaddress) will receive bytes through RBR and the device receives the bytes one by one and len2 number of bytes are put in the buffer at memory address from memRxaddress to memRxaddress + len2 – 1.

(d) A serial device high-level driver function *close* (5) will close the device. It can then be reused only after opening it by open ( ). The device closes by masking the device interrupts.

4.9.5 Device Drivers for Internal Programmable Timing Devices

Generally, there is at least one hardware timer T as an internal device in any systems needing functions related to timers. Using the time-outs (ticks) from T (using overflow interrupts) several needed software timers (SWTs) as can also be driven.

Example 4.20

A given hardware timer time-outs every 2^{16} (16384) counts and let the timer clock give input at every 2 μs. Assume that an SWT is to be programmed to tick every 31 × 32,768 μs = 1.015808 s. SWT should interrupt after swt counts *swtcnt* becomes equal to *numTicks* (preset number of ticks). The SWT is first initialized to *swtcnt* equals 0 and on every T overflow an ISR increments *swtcnt* and when *swtcnt* equals 31 then *swtcnt* is reset to 0 after generating software interrupt by an SWI instruction. An SWT-ISR then executes to perform required actions on SWT interrupt.

Timer device driver function call an ISR. The ISR programming needs an understanding of the programming of each bit of timer control register(s) and status register(s). An important step is programming of each bit of one or two control registers present and the use of status register. The programmer must also take into account the following. (i) Instead of interrupt enable, a device may have a mask bit. Mask bit means interrupt disables on set and enables on reset. Its actions are opposite to that of the enable bit. The programmer must also remember that a certain interrupt cannot disable (cannot mask, NMI). These enable or mask bits are the secondary bits. There is an overall interrupt system enable bit, which is like a master key (primary-level bit) for all maskable interrupt sources. The driver must set that bit also.

Step I: Write in a register that holds the timer maximum count value, the number of count inputs, *numTicks* for the SWT.

Step II: Write in status register the timer status flag(s) equal to reset [in case the device does not reset flag(s) automatically on a read of the status flag(s)].

Step III: Write each bit present in the control register(s). Write interrupt secondary- and primary-level enable bits equals true in control register, write other bits according to their uses. It is essential to write the device enable bit to let the device work. Definition of each bit in the mode register, if present is also essential.

Assume that a free running counter (FRC) is used as a timing device. Device-driver ISR programming steps require use of 68HC11 RTC described in Section 3.8. Consider following example. (68HC11

microcontroller knowledge is presumed here). The example gives the details of bits that are initialized for using FRC device of 68HC11.

Example 4.21

1. *Step I*: Define the output compare register(s) to hold count instance(s) of the FRC when OC flag(s) sets and OC interrupt(s) occurr(s).
2. *Step II*: Flag(s) on its read from status register must be reset in case the device does reset automatically. The flags that may be present are the FRC overflow status flag, OC flag(s), ICAP_F flag(s), RTC flag and SWT flag(s). These are to be reset on a read of the status register.
3. *Step III*: Define control register(s) bits. Here, definition for each bit present is essential. The bits may be as follows. Prescaling bits for count input clock, overflow interrupt enable bit, RTC interrupt enable, OC interrupt enable bit(s), OC enable bit(s), OC output level bit(s), ICAP enable bit(s), ICAP input edge bit, ICAP input bit(s), ICAP interrupt(s) enable bit, SWT enable bits and SWT interrupts enable bit(s).
4. *Step IV*: Also enable the primary level interrupt enable bit, if already not enabled.

4.9.6 Linux Internals as Device Drivers and Network Functions

Drivers for port, keypad, display, timer and network devices (Sections 3.3, 3.6 and 3.9) are most commonly used in the systems. Drivers for PCS (physical coding sublayer) and PMA (physical media attachment) are required in media devices for most voice and video systems. It becomes impractical for a programmer to write the codes for each function of device. For commonly used devices, a programmer most often relies on drivers that are readily available in the thoroughly tested and debugged operating system (Refer to Chapters 9 and 10 for µCOS II, VxWorks OS, Windows CE, OSEK and Real Time Linux).

The Linux operating system is a tested and debugged operating system and is used throughout. It has a large number of drivers (Table 4.2) that are, moreover, in the public domain. Public domain means non-proprietary and usable by anyone. A programmer may therefore choose Linux drivers when the embedded system being designed has the devices that have the drivers available in Linux (refer http://www.linuxdoc.org).

Linux has internal functions called Internals. Internals exist for device drivers and network-management functions. Examples of useful Linux drivers for the embedded system are given in Table 4.2.

Linux internal functions exist for sockets, handling of socket buffers, firewalls, network protocols (e.g., NFS, IP, IPv6 and ethernet) and bridges. These are in the *net* directory. They work separately as drivers and also form a part of the network management function of the OS. (The reader can refer a standard textbook for bit-wise meaning of UDP, PPP and SLIP and for socket functions, firewall and network protocols. For example, refer *Internet and Web Technologies* from Tata McGraw-Hill, 2002 for bit-wise description of PPP, SLIP, TCP, IP, ethernet and other protocols).

Device drivers play a key role in most embedded system as these provide software layers between the application and devices. Drivers control almost all devices except the memory devices and the processor in a system. Linux device drivers are also used popularly because they are tested and debugged and are in the public domain. The Linux OS has internals and a large number of readily available device drivers for the most common physical and virtual devices and has the functions for the network sockets and protocols.

Table 4.2 Useful Linux Device Drivers

Driver type	*Explanation*
char	Drivers for char devices. A char device is a device for handling a stream of characters (bytes).
block	Drivers for block devices. A block device is a device that handles a block or part of a block of data. For example, 1 kB of data handled at a time. (*Note*: Unix block driver does not facilitate use of a part of the block during read or write.)
net	Drivers for network devices. A net device is a device that handles network interface device (card or adapter) using a line protocol, for example, tty or PPP or SLIP.
input	Drivers for the standard input devices. An input device is a device that handles inputs from a device, for example, keyboard.
media	Drivers for the voice and video input devices. Examples are video-frame grabber device, teletext device, radio-device (actually a streaming voice, music or speech device).
video	Drivers for the standard video output devices. A video device is a device that handles the frame buffer from the system to other systems as a char device does or a UDP network packet sending device does.
sound	Drivers for the standard audio devices. A sound device is a device that handles audio in a standard format.
system	Platform-specific drivers. Recently, system processor-specific drivers have also become available in this operating system. Examples are drivers for an ARM processor-based system.

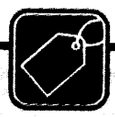
Summary

The following is a summary of the important points that were discussed in this chapter.

- Interrupt means event, which invites attention of the processor for the action of hardware. Event can be a hardware or software event. In response to the interrupt, running routine or program interrupts and a service routine executes.
- When a device or port is ready, a device or port generates an interrupt or when it completes the assigned action, it generates an interrupt. These interrupts are called hardware interrupts. When software run-time exception condition is detected, either processor hardware or a software instruction SWI generates an interrupt for *exception*. An SWI instruction is *INT* n in 80x86.
- Device, run-time error and software instruction-related interrupts are studied. Various possible sources of the software and hardware interrupts are listed.
- Device driver functions execute software-interrupt routines for servicing the device, and drive a peripheral or internal device by create, open, read, write, close or other device function. A device is configured and initialized by using the control bits at its control register(s). The device driver executes on hardware or software interrupts as per set flags in status register(s).
- Physical device drivers, virtual device drivers and ISRs for the software instruction, software-defined condition and error condition are used to program the system.
- Every system has an interrupt service mechanism.
- We learnt *device initialization, driver ISR and function coding* for the parallel ports, serial-line UART and internal timing device in 68HC11.
- Virtual devices like char device, block device or file device, RAM disk, socket, pipe, loop back device are used during programming. These are treated in a way analogous to the physical devices.
- There are the device interrupts as well as other interrupt sources, driver functions and ISRs for which must be written by the programmer. A list of the various possible sources of software and hardware interrupts is given. A

software instruction or a condition during running-related or run-time error-related or device driver function interrupts are important in the systems.

- Interrupt system and individual device interrupt enabling and disabling, interrupt vectors, interrupt pending registers and status registers, non-maskable, maskable, non-maskable only when so defined within a few clock cycles after reset.
- Each running program has a *context* at each running code instance. Context means a CPU state (PC, stack pointer(s), registers and program state (variables that should not be modified by another routine). The context must be saved on a *call* to another ISR or *task* or routine. It must be done before processor switching to another context. Processor switches to another context by retrieving called program context. Certain processors like those from the ARM family provide for fast context switching. These have the internal stack frames or sets of local registers for the contexts. Fast context switching reduces the interrupt latencies and enables the meeting of each real-time task deadline. The OS program provides for memory blocks (allocates the blocks) to be used as stack frames in a multitasking system.
- Programming should be such that interrupt latencies are made as short as possible. This helps in meeting the deadlines for each interrupt service. Use of interrupt service threads (slow-level ISRs initiated by SWIs) helps in having the main first-level ISR codes short. The use of a DMA channel facilitates the small interrupt latency periods of an IO interrupt source requiring bulk or burst data transfers.
- There may be simultaneous service demands from multiple sources. Assignment of software priorities among the multiple sources of interrupts keeping in view the available hardware priorities is essential.
- Linux has a large number of device drivers, which are open-source.



Keywords and their Definitions

Context	:	PC, stack pointer as well as the program status word and processor registers for a foreground program or ISR or task. It can also include memory block addresses allotted to the program or routine.
Context switching	:	Saving the foreground program [interrupted routine (or function)] context and retrieving or loading the new context of the called routine. The time taken in context switching is included in the interrupt latency period.
Deadline	:	A period during which service to an interrupt must start.
Device attaching (adding)	:	Configuring a device and enabling the use of its driver.
Device detaching (removing)	:	Disabling the use of a device driver by the system.
Device driver ISR codes	:	Codes for read and write or other operations at the device addresses after reading device status on interrupt.
Device initialization codes	:	Codes for programming the control register of a device.
Device opening	:	Resetting the device control bits and preparing it for the use of its driver.
Device closing	:	Resetting the device control bits and its next time use is then possible only by opening it again.
Exception	:	An interrupt on detection of a run-time event during computations or communication. Setting of a condition that may be defined by the programmer.
Exception handler	:	Programmer defines the exception handler ISR also for handling service for that condition. Error conditions are handled by the exception handlers. Exception handler is called on executing an SWI instruction.
Foreground program	:	Foreground program is one that is executed when no interrupt call is being serviced.
Hardware-assigned Priority	:	Priority assigned by the processor itself to service a source when several interrupts need the interrupt service.

Hardware Interrupt : Interrupt of devices or ports at the system.

Hardware timer : A timer present in the system as hardware and which gets inputs from the internal clock with the processor. A device-driver program programs it like any other physical device.

Interrupt : CPU on interrupt event may initiate a further action by vectoring to a vector address and calling an ISR or else it continues with the current process (task) if the interrupt is disabled or masked.

Interrupt enable bit : It enables (unmasks) the interrupts from a source(s).

Interrupt flag : A register bit for a Boolean variable that sets to reflect a need for executing an ISR. It resets when corresponding ISR starts executing.

Interrupt latency : A period for waiting for service after a service demand is raised (source status flag sets).

Interrupt mask bit : When this bit is reset (= false) the request for initiation of interrupt service is responded, otherwise it is not responded.

Interrupt pending register : A register to show the interrupt sources or source groups from various devices that are pending for service by executing the corresponding ISRs. It is a '*read and write*' register. A bit auto resets in it when the corresponding interrupt service starts. A user instruction can also reset a bit in the register.

Interrupt service mechanism : A mechanism for interrupt-driven service of the devices and ports. It saves the processor waiting time, because it lets the processor process the multiple devices and virtual devices. The mechanism also sets the priorities and provides for enabling and disabling the services.

ISR : A program that is executed on interrupt after saving the necessary parameters called context onto the stack so that the same can be retrieved on return from the routine last instruction. An ISR is also a device driver ISR when a high level language device driver function executes SWI and it services a device-interrupt. An ISR is also a trap when it services a software error or other condition-related interrupts with error detected by processor hardware. An ISR is also called *exception handler* on *exception,* which is *thrown* when it services software run-time condition detection and the condition is detected in the software routine. It is also called *signal* handler. When a *signal* function is called in a program. Each signal or exception throwing function executes an SWI, which initiates an ISR.

Interrupt vector : A memory address where there are bytes to provide the corresponding ISR address. The system has the specific vector addresses assigned by the hardware for each interrupting source for each internal device.

Interrupt vector table : A table for the interrupt vectors in the memory. The table facilitates the service of the multiple interrupting sources or source-groups for each internal device. In each row for an interrupt vector address, there are bytes to provide the corresponding interrupt service routine address.

Linux : An open source OS. It has a large number of device drivers and network management functions.

Linux device drivers : Device drivers taken from the Linux source.

Maskable interrupt source : A source, service routine call for which can be disabled or service of which masked.

Non-maskable interrupt source : A source, which cannot be disabled and which is used for the highest priority interrupt service cases, like RAM parity error.

Polling : A method to find the status of a peripheral or device. It is also a method by which at the end of an instruction or at the end of an ISR, the pending interrupts are searched by the processor from the status register or interrupt pending register to service the one with the highest priority.

Primary-level enable bit : A bit, which enables or disables any service on interrupt by all the maskable sources. It helps in executing critical section codes and preventing service to any other maskable source during disabling of service.

Secondary-level mask bit : It disables service from an individual source or source group.

Signal : Signal is function to initiate software-interrupt on an instruction to call a software-initiated ISR. An *exception* or trap may also be called a *signal*. Signal is called by SWI instruction in ARM and INT n in 80x86.

Software-assigned priority : (Hardware signal is different.) A priority for a source or source group. It is defined at a register called interrupt priority register. When several interrupts occur at the same time, software-assigned priorities over-ride the hardware priorities.

Software Interrupt : An interrupt by an error condition trap or illegal opcode or an SWI instruction (or INT n instruction 80x86) in a routine or ISR or software timer or signal.

Stack frame : A set of registers or a memory block that stores the context for a program or ISR.

Status register : A read only register for a device to set a flag on arising of an interrupt. A user instruction can also reset a bit in it. If a device has a number of sources the status register has a number of flags and a distinct source for each source. When it is read by a processor instruction, the flag resets.

Trap : An interrupt on detection by hardware a run-time computational or other event. The processor may also signal an exception on the *trap*. Example of a trap is division by zero in 80x86.

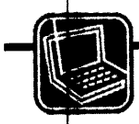
Virtual device : A device, which emulates the physical device and drives by virtual device drivers provided in an operating system. Examples are file, pipe, socket, etc.

Worst-case latency : Maximum interrupt latency found in the worst possible case.

Review Questions

1. What are the disadvantages and advantages of *busy and wait transfer* mode for the I/O devices?
2. What are the advantages and disadvantages of interrupt-driven data transfer?
3. What are the advantages of DMA based or peripheral-transcation-server based data transfer over the interrupt-driven data transfer?
4. How is the vector address used for an interrupt source?
5. Interrupt vector addresses are prefixed in the interrupt mechanism for the known internal peripherals in a microcontroller. How are the vector addresses assigned for exceptions and user-defined interrupts?
6. Interrupt mechanism in each processor differs from a processor family to another. Explain, why the device drivers are processor-sensitive programs.
7. How do you intialize and configure a device? Take an example of serial-line driver at COM port of PC.
8. How is a *file* at the memory act handled as a device?
9. What are the advantages of RAM disk?
10. Make a list of Linux internal *net* directory functions for sockets, handling of socket buffers, firewalls, network protocols (e.g., NFS, IP, IPv6 and ethernet) and bridges. Why are these device drivers assigned in a separate directory of network management function of Linux OS.
11. Define *context*, *interrupt latency* and *interrupt service deadline*.
12. Why is the context switching in an embedded processor faster than saving the pointers and variables on the stack using a stack pointer? How does the context switching time reduce in processor architectures for embedded systems?

13. How is the context switching handled in ARM7?
14. DMA helps in reducing the processor load by providing direct access for the IOs. How does it help in faster task execution in a multi-tasking system by the reduced interrupt service latencies?
15. What do you mean by throwing an exception? How is the exception condition during execution of a function (routine) handled?
16. How do the device driver functions and ISRs differ? How do the ISR calls differ in 80x86 and 8051?
17. How do you assign service priority to the multiple device drivers of a system? How do you assign priorities to the timer devices and ADC device?
18. What are the uses of hardware-assigned priorities in an interrupt service mechanism?
19. What are the uses of software-assigned priorities in an interrupt service mechanism?
20. How is break point interrupt important for debugging embedded software?
21. What do you mean by POSIX function?



Practice Exercises

22. How do you write device driver? List the steps involved in writing a device driver.
23. Search web and design a table to show features in device driver modules of embedded Linux OS. Explain with examples each a char device, block device and block device configurable as char device. UART is a char device. Why is it a char device?
24. Give software-related interrupt examples. What are the interrupts in 8086, which generate software error?
25. Show the state machine-generated states in a key marked as number 4 in the mobile device. How will you use the SWI instruction to generate an SMS message in a mobile phone having a T9 keypad?

Programming Concepts and Embedded Programming in C, C++ and Java

5

Let us recapitulate the following points covered in the previous chapters.

- System hardware consists of processor(s), memory (ROM and RAM), I/O port, timing and external devices.
- Programming is required for computational tasks.
- Programming is required for the ISRs called on software interrupts from traps, exceptions, errors, signals and interrupts from physical and virtual devices.
- OS software is required for using any device in a simple or sophisticated application(s).
- Certain software instructions are required according to the given processor, memory and device hardware and as per the system interrupt servicing mechanism.

Programming is the most essential part of any embedded system design. Except for certain processor and memory-sensitive instructions where program codes may be written in assembly, most of the codes are written in a high level-language.

LEARNING OBJECTIVES

For an in-depth learning of the programming language, a reader should refer to the standard textbooks and do required practice exercises. We will learn basics of programming the functions (methods) and concepts of object-oriented programming with reference to building software for the embedded systems. OS concepts, we will learn later.

The following are the topics that will be discussed in this chapter:

1. *Programming in the assembly language vs. high-level language and the powerful features of C for embedded systems.*
2. *Program elements: Preprocessor directives and the header files, include files and source files that are used in a program for an application.*
3. *Program elements: Macros and functions and their uses in a C program.*
4. *Program elements: Data types, pointers, data structures, arrays, queues, stacks, lists and trees, modifiers, conditional statements and loops.*
5. *Program elements: Function calls, multiple functions, function pointers, function queues and service-routine queues.*
6. *Object-oriented Programming concepts, embedded programming in C/C++, Java and J2ME*

Program models for building the software will be dealt with in Chapter 6. Concepts of the processes, tasks, threads and the concepts of interprocess synchronization will be covered in Chapter 7 and of RTOSes in Chapter 8. Popular RTOSes are described in Chapters 9 and 10.

5.1 SOFTWARE PROGRAMMING IN ASSEMBLY LANGUAGE (ALP) AND IN HIGH-LEVEL LANGUAGE 'C'

5.1.1 Assembly Language Programming

Assembly language coding of an application has the following advantages.

1. The assembly codes are sensitive to the processor, memory, ports and devices hardware. *It gives a precise control* of the processor internal devices and *complete use of the processor-specific features* in its instruction set and its addressing modes.
2. The machine codes are *compact*, processor- and memory-sensitive. This is because the codes for declaring the conditions, rules and data type do not exist. The system thus needs a smaller memory. Excess memory needed does not depend on the programmer data type selection and rule declarations. The program is also not compiler specific and library functions dependent.
3. Device driver codes may need only a *few assembly instructions*. For example, consider a small embedded system, a timer device in a microwave oven or an

automatic washing machine or an automatic chocolate-vending machine. Assembly codes for these can be compact and precise, and are conveniently written.

4. We use *bottom-up-design* approach. It is an approach in which programming is first done for the sub-modules of the specific and distinct sets of actions. An example of the modules for specific sets of actions is a program for a software timer, RTCSWT:: run (real-time clock software timer function *run*). Programs for delay, counting, finding time intervals and many applications can be written. Then the final program is designed. The approach to this way of designing a program is to first code the basic functional modules and then use these to build a bigger module.

5.1.2 High-level Language Programming

High-level language coding of *source files* in C or C++ or Java has great advantages and therefore most programming is in high-level language. Basic advantages are as follows:

1. *High-level program development cycle is short even for complex systems* because of the followings: use of routines (called functions in C/C++ and methods in Java), standard library functions, use of modular programming approach, top-down design or object-oriented design approach. Application programs are structured to ensure that the software is based on sound software engineering principles and programmed with the given OS, file systems, device and network drivers.
   (a) A function defines a method of operation, and sets of statements and commands are run when that function is called.
   (b) Library functions are standard functions, which are readily available to a programmer and the codes for them are not defined by programmer. For example, square root method of operation. The *use of the standard library function*, square root ( ), saves the programmer time for coding. New sets of library functions exist in an embedded system-specific C or C++ compiler. Exemplary library functions are delay ( ), wait ( ) and sleep ( ).
   (c) Identical devices such as serial-line device (UART) are used in a number of embedded systems. Directly programming these functions in each system will mean the *repetitive* and *redundant coding* for each device. It is better to use the device drivers in high-level language, which use the functions specified in the OS. The programmer simply specifies device ID and some of the function arguments passed to the device driver function when needed and use it at another instance of the device use.
   (d) *Modular programming approach* is an approach in which the building blocks are reusable software components. A module is built by software components. The components are built by a set of functions. Consider an analogy to an IC (integrated circuit). Just as an IC has several circuits integrated into one, similarly a module building block may call several functions and library functions. A module is then well tested for a well-defined goal and for the well-defined data input and outputs. It should have only one calling method. There should be one return point from it. It should not affect any data other than the one it operates; this means that there should be data encapsulation. It must return (report) error conditions encountered during its execution.
   (e) *Top-down design* is another programming approach in which the *main* program is first designed, then its modules, sub-modules, and finally, the functions.
2. *High-level program facilitates data type declarations*: Data type declarations provide programming ease. For example, there are four types of integers, *int*, *unsigned int*, *short* and *long*. When dealing with positive only values, we declare a variable as *unsigned int*. For example, numTicks (number of ticks to a clock) has to be unsigned. We need a signed integer, *int* (32 bits) in arithmetical calculations. An integer can also be declared as data type, *short* (16 bits) or *long* (64 bits). To manipulate the text and strings for a character, another data type is *char*. *Each data type is an abstraction for the methods, which are permitted for using, manipulating, representing and for defining a set of permissible operations on that data.*

3. *High-level program facilitates 'type checking'* making the program less prone to error. For example, type checking does not permit subtraction, multiplication and division on the *char* data types. It permits 'plus' operator to be used for concatenation when using *char* data types and lets the 'plus' operator to be used for arithmetic addition when using *int*, *unsigned int*, *short* and long type of data. (Concatenation operation can be understood as follows: the *micro* plus *controller* concatenates into the *microcontroller*, where micro is an array of *char* values and controller is another array of *char* values.)
4. *High-level program facilitates use of control structures* (e.g., *while*, *do-while*, *break* and *for*) and *conditional statements* (e.g., *if*, *if-else*, *else-if* and *switch-case*) to specify the program flow by simple statements.
5. *High-level program has portability* of non-processor-specific codes. Therefore, when the hardware changes, only the modules for the ISRs of device drivers and device management, initialization and program-locator modules and initial boot-up record data need modifications.

Additional advantages of C as a high-level languages are as follows. It is a *language between low-(assembly) and high-level languages*. Inserting the assembly language codes in-between is called in-line assembly. A direct hardware control is thus also feasible by in-line assembly, and the complex part of the program can be in high-level language.

High-level language programming makes the program development cycle short, enables use of the modular programming approach and allows us to follow sound software engineering principles. It facilitates the program development with top-down design approaches. Embedded system programmers have since long preferred C for the following reasons: (i) The feature of embedding assembly codes using in-line assembly. (ii) Readily available modules in C compilers for the embedded system and library codes that can directly port into the system-programmer codes.

5.2 C PROGRAM ELEMENTS: HEADER AND SOURCE FILES AND PREPROCESSOR DIRECTIVES

A C program has following structural elements.

1. Preprocessor declarations, definitions and statements.
2. Main function.
3. Functions, exceptions and ISRs.

A C program has the following preprocessor structural elements.

1. *Include* directive for the file inclusion.
2. *Definitions for preprocessor global variables* (global means throughout the program module).
3. *Definitions* of constants.
4. *Declarations* for global data type, type declaration and data structures, macros and functions.

The C program elements, header and source files and preprocessor directives are explained in the following subsections.

5.2.1 Include Directive for the Inclusion of Files

Any C program first includes the header and source files that are readily available. A case study of sending a stream of bytes through a network driver card using a TCP/IP protocol is given in Example 11.3. Its C program starts with the codes given in Example 5.1. The purpose of each included file is mentioned in the comments within the /* and */ symbols as per the practice in C.

Example 5.1

```
# include "VxWorks.h" /* Include VxWorks functions*/
# include "semLib.h" /* Include Semaphore functions Library */
# include "taskLib.h" /* Include multitasking functions Library */
# include "msgQLib.h" /* Include Message Queue functions Library */
# include "fioLib.h" /* Include File-Device Input-Output functions Library */
# include "sysLib.c" /* Include system library for system functions */
# include "netDrvConfig.txt" /* Include a text file that provides the 'Network Driver
  Configuration'. It provides the frame format protocol (SLIP or PPP or Ethernet) description, card
  description/make, address at the system, IP address (s) of the node (s) that drive the card for
  transmitting or receiving from the network. */
# include "prctlHandlers.c" /* Include file for the codes for handling and actions as
  per the network layer-protocols used for driving streams to the network. */
```

Include is a preprocessor directive to include the contents (codes or data) of a file. The files that can be included are given next. Inclusion of all files and specific header files has to be as per the requirements.

1. *Including code files*: These are the files for the codes already available. For example, # include '*prctlHandlers.c*'.
2. *Including constant data files:* These are the files for the codes and may have the extension '.const'.
3. *Including strings data files:* These are the files for the strings and may have the extension '.strings' or '.str.' or '.txt. For example, # include '*netDrvConfig*.txt' .
4. *Including initial data files:* There are files for the initial or default data for the shadow ROM of the embedded system. The boot-up program is copied later into the RAM and may have the extension '.init'. On the other hand, RAM data files have the extension, '.data'.
5. *Including basic variable files:* These are the files for the local or global static variables that are stored in the RAM because they do not possess the initial (default) values. The static means that there is a common not more than one instance of that variable address and it has a static memory allocation. For example, system has is only one real-time clock, and therefore only there is one instance of addresses of the clock variables. The basic variables of the clock are stored in the file with the extension '.bss'.
6. *Including header files:* It is a preprocessor directive, which includes the contents (codes or data) of a set of source files. These are the files of a specific module. A header file has the extension '.h'. Examples are as follows. The string manipulation functions are needed in a program using strings. These become available once a header file called '*string.h*' is included. The mathematical functions, *square root, sin, cos, tan, atan* and so on are needed in programs using mathematical expressions. These become available by including a header file, called '*math.h*' The preprocessor directives will be '# include <*string.h*>' and '# include <*math.h*>' for including standard library functions for the strings and mathematical operations in the program.

Also included are the header files for the codes in assembly, and for the I/O operations (conio.h), for the OS functions and RTOS functions. '# include VxWorks.h' is directive to the compiler, which includes VxWorks RTOS functions.

Note: Certain compilers provide for *conio.h* in place of *stdio.h*. This is when embedded systems do not need the file functions for opening, closing, read and write operations on the keyboard and video monitor. So when including *stdio.h*, it will make the code too big.

What is the difference between inclusion of a header file, and text file or data file or constant file? Consider the inclusion of *netDrvConfig*.txt.*txt* and *math.h*. (i) The header files are well-tested and debugged modules.

(ii) The header files provide access to standard libraries. (iii) The header file can include several text files or C files. (iv) A text file is a description of the text that contain specific information.

5.2.2 Source Files

Source files are program files for the functions of application software. The source files need to be compiled. A source file will also possess the preprocessor directives of the application and have the *first function from where the processing will start.* This function is called *main* function. Its codes start with *void main* ( ). The *main* calls other functions. A source file holds the codes.

5.2.3 Configuration Files

Configuration files are the files for configuration of the system. Device configuration codes can be put in a file of basic variables and included when needed. If these codes are in the file "serialLine_cfg.h" then # include '*serialLine_cfg.h*' *will be the preprocessor directive.* Consider another example. '*# include* '*os_cfg.h*'; this will include os_cfg header file.

5.2.4 Preprocessor Directives

Preprocessor constants, variables and inclusion of configuration files, text files, header files and library functions are used in embedded C programs. A preprocessor directive starts with a sharp (hash) sign. These commands are for the following directives to the compiler for processing.

1. *Preprocessor global variables*: For example, in a program the IntrDisable, IntrPortAEnable, IntrPortADisable, STAF and STAI may be the global variables for disabling interrupts, enabling port A, disabling port A, status flag, status flag for interrupt, respectively. Now '# define volatile boolean IntrEnable' is a preprocessor directive. It means it is a directive before processing to consider IntrEnable a global variable of boolean data type and is volatile. (Volatile is a directive to the compiler not to take this variable into account while compacting and optimizing the codes.)
2. *Preprocessor constants*: '*# define* false 0' is a preprocessor directive in an example. It means it is a directive before processing to assume 'false' as 0. The directive '*define*' is for allocating pointer values in the program. Consider # define portA (volatile unsigned char *) 0x1000 and # define PIOC (volatile unsigned char *) 0x1001. 0x1000 and 0x1000 are the addresses fixed for port A (port A register) and PIOC (port input–output control register) are the constants defined for the 68HC11 register addresses.

Strings can also be defined. Strings are the constants, for example, those used for an initial display on the screen in a mobile system. For example, # define *welcome* 'Welcome To ABC Telecom'.

5.3 PROGRAM ELEMENTS: MACROS AND FUNCTIONS

Table 5.1 lists these elements and gives their uses.

Preprocessor Macros: A macro is a collection of codes that is defined in a C program by a name. It differs from a function in the sense that once a macro is defined by a name, the compiler puts the corresponding codes for it at every place where that macro name appears. For example, consider the macros, 'enable_Maskable_Intr ( )' and 'disable_Maskable_Intr ( )'. (The pair of brackets in the macro is optional. If

it is present, it improves readability as it distinguishes a macro from a constant.) Whenever the name enable_Maskable_Intr appears, the compiler places the codes designed for it.

Table 5.1 Uses of the Various Sets of Instructions as the Program Elements

Program Element	*Uses*	*Saves context on the stack before its start and retrieves them on return*	*Feasibility of nesting one within another*
Macro	Executes a named small collection of codes.	No	None
Function	Executes a named set of codes with values passed by the calling program through its arguments. Also returns a data object when it is not declared as void. It has the context-saving and retrieving overheads.	Yes	Yes, can call another function and can also be interrupted
Main function	Declarations of functions and data types, typedef and either: (i) executes a named set of codes, calls a set of functions and calls on the interrupts the ISRs or (ii) starts OS Kernel.	No	None
Reentrant function	Refer Sections 5.4.6.	Yes	Yes to another reentrant function only
Interrupt service routine (ISR)	Declarations of functions and data types, typedef and executes a named set of codes. Must be short so that other sources of interrupts are also serviced within the deadlines. Must be a re-entrant routine. Not allowed to wait for interprocess messages (semaphore or mailbox or queue messages) but allowed to send the interprocess messages for the ISTs or tasks.	Yes	To ISR of higher- priority unmasked sources
Process or task or thread	Refer Sections 7.1–7.3. Must either be a reentrant routine or must have a solution to the shared data problem.	Yes	None
Recursive function	A function that calls itself. It must be a reentrant function also. Most often its use is avoided in embedded systems due to memory constraints. (Stack grows after each recursive call and it may choke the memory space availability.)	Yes	Yes

Macros, called test macros or test vectors are also designed during programming and are used for debugging. How does a macro differ from a function?

1. The codes for a function are compiled once only. On calling that function, the system has to save the context, and on return restore the context. Further, a function may return nothing (*void* declaration case) or return a Boolean value, or an integer or any primitive or reference type of data. (Primitive means similar to an integer or character. Reference type means similar to an array or structure.) For example, the enable_PortA_Intr ( ) and disable_PortA_Intr ( ) are the function calls. (The brackets are now not optional.)

2. The codes for macro are compiled in every function wherever that macro name is used, as the compiler, before compilation, puts the codes at the places whereever the macro is used. On using the macro, the processor does not have to save the context, and does not have to restore the context, as there is no return.
3. Macros are used for short codes only. This is because, if a function call is used instead of a macro, the overheads (context saving and new context retrieving, and other actions on function call and return) will take a time, $T_{overheads}$ that is the same order of magnitude as the time, T_{exec} for execution of short codes within a function. We use a function for codes when the $T_{overheads} << T_{exec}$, and a macro for codes when $T_{overheads} \simeq$ or $> T_{exec}$.

Macros and functions are used in C programs. Functions are used when the requirement is that the codes should be compiled once only. However, on calling a function, the processor has to save the context, and on return, restore the context. A function may also return either nothing (void declaration case) or return a Boolean value, or an integer or any primitive or reference type of data. Macros are used when short functional codes are to be inserted in a number of places or functions.

5.4 PROGRAM ELEMENTS: DATA TYPES, DATA STRUCTURES, MODIFIERS, STATEMENTS, LOOPS AND POINTERS

5.4.1 Use of Data Types

Whenever a data is named, it will have the address(es) allocated at the memory. The number of addresses allocated depends on the data type. For example, a data type *long* is declared for numTicks (number of ticks). Then numTicks will need four memory addresses.

C allows the following primitive data types. The *char* (8 bits) for characters, *byte* (8 bits), *unsigned short* (16 bits), *short* (16 bits), *unsigned int* (32 bits), *int* (32 bits), *long double* (64 bits), *float* (32 bits) and *double* (64 bits). (Certain compilers do not take the 'byte' as a data type definition. The 'char' is then used instead of 'byte'. Most C compilers do not take a Boolean variable as data type. The *typedef* is used to create a Boolean type variable in the C program.)

A data type appropriate for the hardware is used. For example, a 16-bit timer can have only the unsigned short data type, and its range can be from 0 to 65535 only.

The typedef is also used. It is made clear by the following example. A compiler version may not process the declaration as an unsigned byte. The 'unsigned character' can then be used as a data type. It can then be declared as follows.

```
typedef unsigned character portAdata
#define Pbyte portAdata 0xF1
```

5.4.2 Use of Pointers and Null Pointers

Pointers are powerful tools when used correctly and according to certain basic principles. Pointer is a reference to a starting memory address. A pointer can refer to a variable, data structure or function. Before a pointer, in C language symbol. * is used. For example, unsigned char *0x1000 means a character of 8 bits at address 0x1000.

A NULL pointer declares as following: '*#define NULL (void*) 0x0000*'. (We can assign any address instead of 0x0000 that is not in use in a given hardware.)

Exemplary uses are as follows.

Example 5.2

1. Consider 'unsigned short *timer1'. Pointer *timer1* will point to two bytes, and the compiler will reserve two memory addresses for the contents of timer1.
   Consider two statements, '*unsigned short* *timer1;' and 'timer1++;'. The second statement adds 0x0002 in the address of timer1. Why?
   In C, if x is a variable, then x ++ means increment of the value of x by 1. If p is a pointer, then p ++ means increment of the value of p to the next address.
   timer1 ++ means point to the next address, and unsigned short declaration allocated two addresses for timer1. (timer1 ++; or timer +=1 or timer = timer +1; will have identical actions.) Therefore, the next address is 0x0002 more than the address of timer1 that was originally defined. Had the declaration been 'unsigned int *timer1;' and 'timer1++;' (in case of 32-bit timer), the second statement would have incremented the address by 0x0004.
2. Let a byte each be stored at a memory address. Let a port A in a system have a buffer register that stores a byte. Now a program using a pointer declares the byte at port A as follows: 'unsigned byte *portA'. The * means 'the contents at'. This declaration means that there is a pointer and an unsigned byte for port A. The compiler will reserve one memory address for that byte.
3. Consider declarations as follows: void *portAdata; the void is the undefined data type for portAdata. The compiler will allocate address for the *portAdata without any type check.
4. A pointer can be assigned a constant fixed address. Two preprocessor directives: '# define portA (volatile unsigned byte *) 0x1000' and '# define PIOC (volatile unsigned byte *) 0x1001'. Alternatively, the addresses in a function can be assigned as follows: 'volatile unsigned byte * portA = (unsigned byte *) 0x1000' and 'volatile unsigned byte *PIOC = (unsigned byte *) 0x1001'. An instruction, 'portA ++;' will make the portA pointer point to the next address and which is PIOC.
5. Consider, unsigned byte portAdata; unsigned byte *portA = &portAdata. The first statement directs the compiler to allocate one memory address for portAdata because there is a byte each at an address. The & (ampersand sign) means 'at the address of'. This declaration means the positive number of 8 bits (byte) pointed by portA is replaced by the byte at the address of portAdata. The right side of the expression evaluates the contained byte from the address, and the left side puts that byte at the pointed address. As the right-side variable of portAdata is not a declared pointer, the ampersand sign is kept to point to its address so that the right-side pointer gets the contents (bits) from that address. (*Note*: The equality sign in a program statement means 'is replaced by'.)

5.4.3 Use of Data Structures: Queues, Stacks, Lists and Trees

A data structure is a way of organizing several data elements of same types or different types together at consecutive memory addresses. A data element in a data structure can then be identified and accessed with the help of a few pointers and/or indices and/or functions. Marks (or grades) of a student in the different subjects studied in a semester are put in a proper table. The table in the mark sheet shows them in an organized way. Similarly, when there is a large amount of data, it must be organized properly.

A data structure is an important element of any program. Few important data structures include: *stack*, *one-dimensional array*, *queue*, *circular queue*, *pipe*, a *table* (two-dimensional array), *lookup table*, *hash table* and *list*. Following describes different data structures and how it is put in the memory blocks in an organized way. Any data structure element can be retrieved using the pointers.

Stack A data structure, called *stack* is a special program element. A *stack* means an allotted memory block data from which a data element is read in a LIFO (*last in first out*) way and an element is popped or pushed from an address pointed by a pointer, called SP (stack pointer) or S_{top} and SP changes on each push or pop such that it points to the top of stack.

Various stack structures may be created during processing. For handling each stack, one pointer, which points to the stack top is needed. Figure 5.1 shows the various stack structures that are created during execution of the embedded software.

1. A call can be made for another routine during running of a routine. In order that on completion of the called routine, the processor returns only to the one calling, the instruction address for return must be pushed on the stack. Pushing means saving on the stack top and incrementing stack to point to the next top. Popping means retrieving the saved address from the stack top and decrementing the stack to point to the previous top. There can also be nested calls and returns. Nesting means one routine calls another, which calls another and return from the called routine is always to the calling routine. Therefore, at the memory a block of memory address is allocated to the stack that saves the pushed *return addresses* of the nested calls. It is shown in Figure 5.1(a). Two bytes of address are acquired in the PC from stack on return from a call to a routine (function). Assume 10 nested calls are present in the system or other functions. Assume that PC address is of 4 B. Memory allocation required for a stack structure for pushing the return instruction addresses is 40 B.
2. There may be at the beginning of an input data, for example, received call numbers in a phone, which is saved onto a stack at RAM in order to be retrieved later in the LIFO mode. It is shown in Figure 5.1(b). Consider for example, on each push the following are saved on a stack. (i) Four pointers (addresses each of 4 bytes); (ii) four integers (each of 4 bytes) and (iii) four floating point numbers (each of 4 bytes). Memory allocation required for a stack structure for pushing the function parameters $= 4 \times 4 + 4 \times 4 + 4 \times 4 = 48$ B.
3. An application may also create the run-time stack structures. There can be *multiple data stacks* at the different memory blocks, each having a separate pointer address. There can be *multiple stacks* shown as Stack 1, ..., Stack N in Figure 5.1(c).
4. Each task or thread in a multi-tasking or multi-threading software design (Sections 7.1–7.3) should have its own stack where its *context* (Section 4.6) is saved. The context is saved on the processor on switching to another task or thread. The context includes the return address for the PC for retrieval on switching back to the task. There can be *multiple stacks* shown as saved contexts of the threads as the stacks shown in Figure 5.1(d) at the memory for the different task contexts at the different memory blocks, each having a separate pointer address. Threads of application programs and supervisory (OS) programs have separate stacks at separate memory blocks.

Each processor has at least one stack pointer register so that the instruction stack can be pointed and calling of the routines can be facilitated.

Some advanced processors have multiple stack pointers. There are four pointers as follows:

1. RIP (return instruction pointer): RIP is for saving the return address of the PC when a routine calls another routine or ISR. RIP is called link register (LR) in ARM processor.
2. SP (stack pointer): SP is pointer to a memory block dedicated to saving the context on context switch to another ISR or routine. There is a stack pointer in 8051, 68HC11 and 80196.
3. FP (data frame pointer): FP is pointer to a memory block dedicated to saving the data and local variable values of presently running program (routine).
4. PFP (previous program frame pointer): PFP is pointer to a memory block dedicated to saved the program data frame.

Motorola MC68010 processor provides USP (user stack pointer) and SSP (supervisory stack pointer). Program runs in two modes: user mode and supervisory mode. In supervisory mode the OS functions execute.

There is switch from the user mode to the supervisory mode after every tick of the system clock (Section 8.1.2 will give the details). MC68040 provides for USP, SSP, MSP (memory stack frames pointers), ISP and (instruction stack pointer).

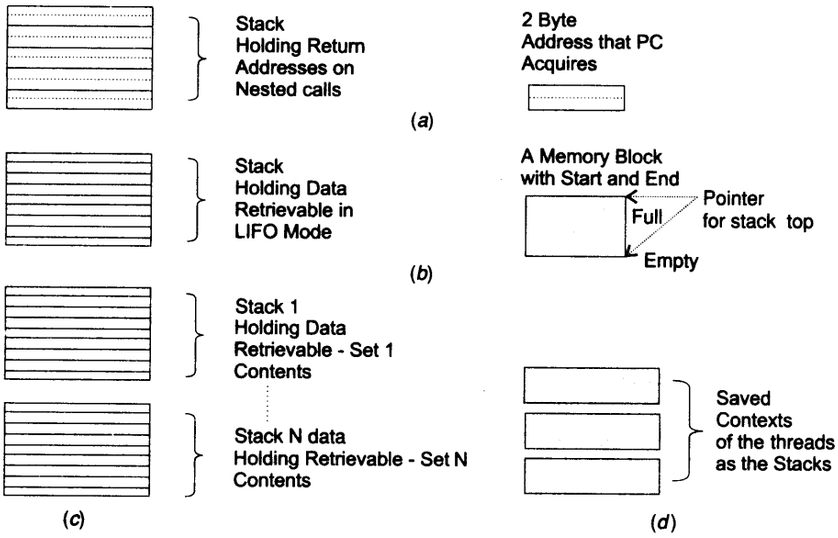


Fig. 5.1 (*a*) A stack due to nested function calls and pushing of program counters (*b*) Stack of pointers and parameters pushed onto the stack before the context switch (*c*) N stacks each having a separate pointer (*d*) Multiple stacks of contexts for the multiple threads

When a processor has only one SP, the OS allocates the memory addresses that are used as the pointers for the multiple instructions and data stacks.

A processor has at least one SP. Each process should have a separate top of SP and a separate block at its allocated memory for the nested function calls. A stack can also be a special data structure at the memory. It has a pointer address that always points to the top of stack. A value from the stack is retrieved from the memory in LIFO mode, while a row of data or a data in a table or a data in the queue is accessed in a FIFO (first in first out) mode. As there are multiple processes in an embedded system, each having separate context, there are multiple stacks.

Array A data structure, *array* is an important programming element. An array has multiple data elements, each identifiable by an index or by a set of indices and indices are unsigned integers. An n-dimensional array is a special data structure at the memory. It has a pointer address that always points to the first element of the array. For handling an n-dimensional array, one pointer, which points to the first element and n indices are needed.

Assume one-dimension array. From the first element pointer and an index of that element, an address is constructed from which the processor can access one of the array elements. Index is an integer that starts from 0 to (array-length–1) in a given dimension. Data word can be retrieved from any element address in the block that is allocated to the array. A processor register may also be used for storing the index and another register for array base pointer.

Following examples clarify the concept of array.

Example 5.3

1. Consider that `unsigned int [ ] phone_num` means an array of phone numbers, `phone_num [0]` refers to that first phone number, phone_num [1] refers to the second phone number, and so on. The `phone_num` itself points to the first element.]
2. Consider `unsigned char [ ] name` which means an array of characters for the name. `name [0]` refers to the first character, `name [1]` refers to the second character and so on. The name without index points to the first array element.
3. Consider the results of a test in a class with 30 students with roll numbers 1–30. Let i be an index used instead of a roll number. Let marks in the test of roll number 1 be in the scalar integer variable, M[0]. Let M[0], M[1], ..., M[28] and M[29] be the variables for the marks of roll numbers 1, 2, ..., 29, 30, respectively. There is a pointer that points to the first scalar value M[0]. A register called index register may point to M[0]. The index register could then be incremented from 0 to 29 by an instruction within a loop to point to the marks of students of succeeding roll numbers. Figure 5.2(a) shows an array in the memory block with the pointers for base and index that jointly point to its element marks [i].

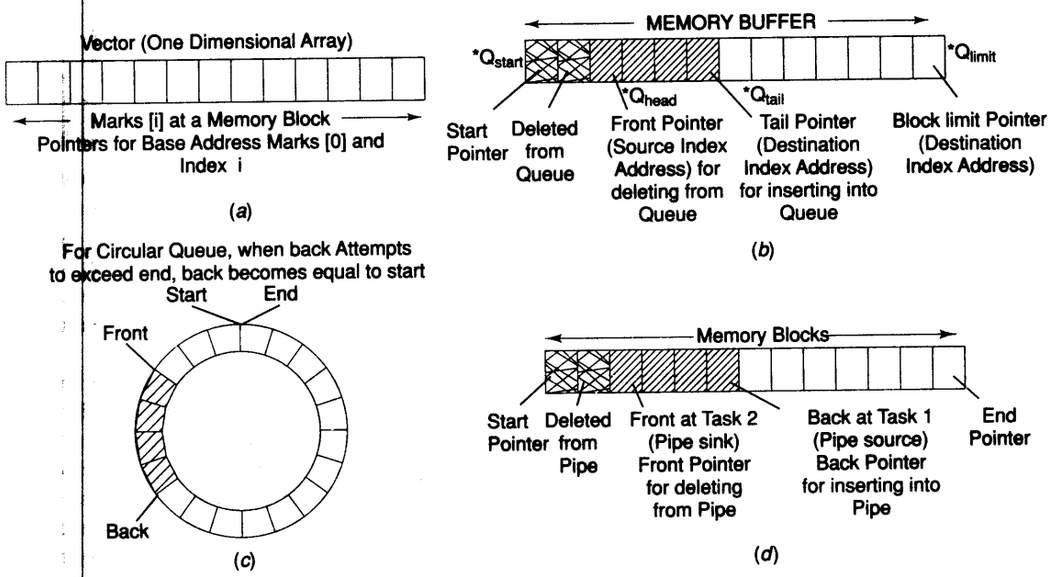


Fig. 5.2 (*a*) An array at a memory block with one pointer for its base, first element with index = 0. Data word can be retrieved from any element by defining the pointer and index (*b*) A queue at a memory block between $*Q_{start}$ and $*Q_{limit}$ with two pointers $*Q_{head}$ and $*Q_{tail}$ to point to its two elements at the queue front and back. A data word retrieves in the FIFO mode from a queue (*c*) A circular queue at a memory block with two pointers to point to its two elements at the front and back. A pointer on reaching a limit of the block returns to the start of the block (*d*) Memory blocks at source and sink for a pipe

Example 5.4

Take another example. An expression, $y_k = \Sigma(a_i . x_{k-i})$ has coefficient a_i. These are stored as an array. Input x_i also stores as another array and output y_k as yet another array. Here, i and k are the integers each varying from $-N$ to N–1, where N is per the limits. Three arrays y [ ], a [ ] and x [ ] are used for calculating 20 filtered output sequences by the expression, $y_k = \Sigma(a_i . x_{k-i})$. Memory allocation required for the each array structure = 20 × 8 = 160 B. [Assume each element as double precession floating pointer number of 8 B (64-bits IEEE 754 format).]

Example 5.5

When the index increments by 1 in case of an array, the pointer to the previous element actually increments by 4, and thus the address will increment by 0x0004 in case of an array of integers. [An integer stores as 4-byte number.] For array data type, * is never put before the identifier name, but an index is put within a pair of square brackets after the identifier. Consider a declaration, 'unsigned char portAMessageString [80];'. The port A message is a string, which is an array of 80 characters. Now, portAMessageString is itself a pointer to an address without the star sign before it. [*Note:* Array is therefore known as a reference data type.] However, *portAMessageString will now refer to all the 80 characters in the string. portAMessage String [20] will refer to the twentieth element (character) in the string.

Queue A data structure, called *queue* is another important programming element. In case of array, the reading is with the help of indices and first element address. So any element can be read or written at any instance. In queue, each element is read from an address next to the address from where the queue element was last read. This reading is called *deletion*. In queue, it is written to an address next to the address from where the queue element was last written. This writing is called *insertion*. A *queue* means an allotted memory block from which a data element is retrieved in the FIFO mode. Using the queues, the bytes are sent onto a memory buffer or network or printer.

For handling the queue, two pointers are needed and a memory buffer is allocated between buffer start address pointed by $*Q_{start}$ and buffer-end pointed by $*Q_{limit}$. One pointer $*Q_{tail}$ is for pointing to an address in a memory block where an element can be inserted (added on writing). For a queue of integers, int $*Q_{tail}$, $*Q_{head}$ are declared. $*Q_{tail}$ initially equals $*Q_{start}$ and it should increment on each insertion at queue back or tail pointer. The other $*Q_{head}$ (queue head pointer) initially equals $*Q_{start}$ and is for pointing to an address in a memory block from where an element can be deleted (remove on reading). This pointer should increment on each deletion. Both pointers $*Q_{tail}$ and $*Q_{head}$ point at the beginning to $*Q_{start}$ the starting memory buffer address at the block. Insertions into a queue are usually faster than deletions. For example, in a queue at the printer the system inserts the values faster than the rate at which values are printed. The difference in addresses at the two pointers at an instance is the present queue length.

There is a possibility that the tail pointer may increment beyond a limit set for the queue end address in the memory block. An exception (an error indication) is usually thrown whenever the pointer increments beyond the block end boundary. Else, on further increments an intrusion into other block may occur. Figure 5.2(b) shows a memory block between $*Q_{start}$ and $*Q_{limit}$ with the two pointers $*Q_{head}$ and $*Q_{tail}$ needed for deletions and insertions.

A queue is a data structure with an allotted memory block (buffer) from which a data element is retrieved in the FIFO mode. It has two pointers, one for its head and the other for its tail. Any deletion is made from the head address and any insertion is made at the tail address. An exception (an error indication) must be thrown whenever the pointer increments beyond the block end boundary so that appropriate action can be taken.

Circular Queue A queue is called *circular queue* when a pointer on reaching a limit $*Q_{limit}$, returns to its starting value $*Q_{start}$. (A circular *queue* means a bounded memory block allotted to a queue such that its pointer on incrementing never exceeds the set limit and returns to start on increment beyond the limit.) From a circular queue also, the data element is retrieved in the FIFO mode but no exception is thrown on exceeding the limit of the memory block allocated. Figure 2.4(c) shows a memory block with a circular queue with its two pointers needed for insertions and deletions.

A circular queue is a queue in which tail and head pointers cannot increment beyond the memory block (buffer) and reset to the starting value on insertion beyond the boundary.

Pipe A *pipe* is a device, which uses device driver functions and in which insertions are from the source end and deletions are at sink-end. The deletions are at the destination end, and are like in the queue. The insertion source has an identity distinct from a destination (sink) entity where deletions are made and source and destination are connected by some function pipe_connect ( ). Figure 5.2(d) shows memory blocks for a *pipe*.

A pipe is a device with insertions and deletions at distinctly defined source and destination.

Table A *table* is a two-dimensional structure (matrix) and is an important data set that is allocated a memory block. There is always a base pointer for a table. It points to its first element at the first column and first row. There are two indices, one for a column and the other for a row. Figure 5.3(a) shows a memory block with the pointers for a table. Like an array, any element can be retrieved from three addresses for the table base, column index and row index. Instead of a column or row pointer, a value is used in an instruction which is called the *displacement*. Displacement can be used for a column or row.

A *lookup table* is a two-dimensional structure (matrix) and is an important data set. It has only rows and each row has a key and on reading the key, the addressed data is traced.

A table is a data set allocated with a memory block. Three pointers, table base, column index and destination index pointers (or two pointers and one displacement) can retrieve any element of the table. Lookup table is a table of keys (pointers) and from reading a key the addressed data is retrieved.

Hash Table A *hash table* is a data set that is a collection of pairs of *key* and corresponding *value*. A hash table has a key or name in one column. The corresponding value or object is at the second column. The keys may be at non-consecutive memory addresses. Figure 5.3(b) shows a memory block with the pointers for a hash.

A hash table is a data set allocated with a memory block for key and value pairs.

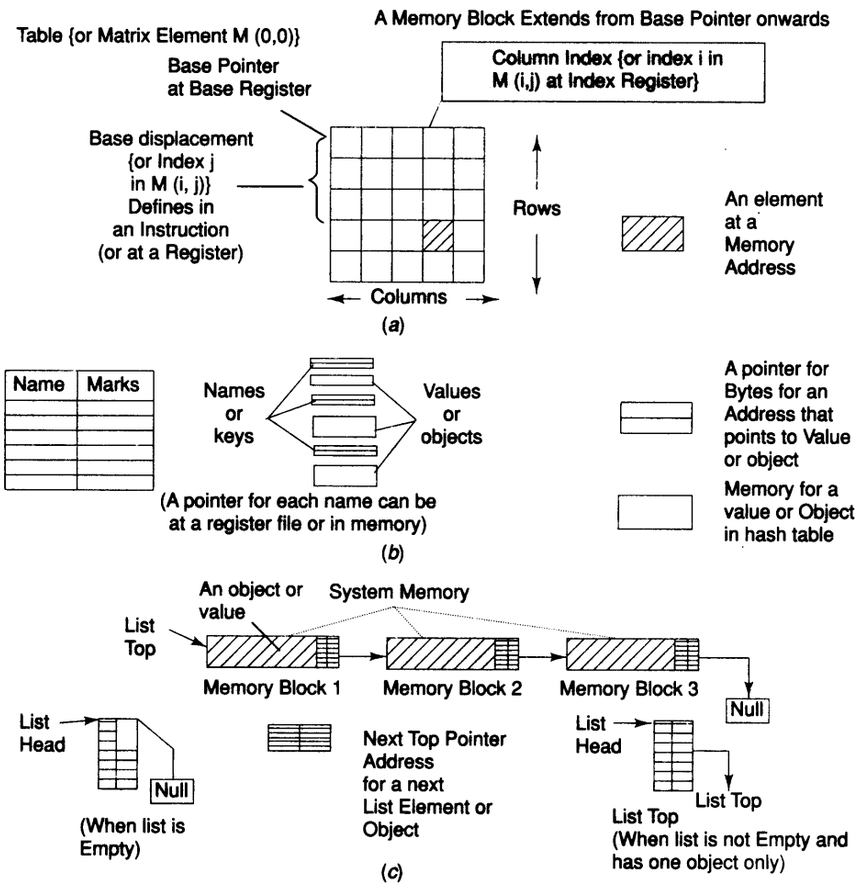


Fig. 5.3 (*a*) A memory block with the pointers for a table (*b*) A memory block for hash table with the pairs of key and value in a hash (*c*) The memory blocks in system memory with the pointers for a list

List A *list* is a data structure with a number of memory blocks, one for each element. A list has a top (head) pointer for the memory address from where it starts. Each list element at the memory also stores the pointer to the next element. The last element points to *null*. A *list* is for non-consecutively located objects at the memory. Figure 5.3(c) shows the memory blocks with the pointers for a list.

Example 5.6

Assume on a real-time clock tick, the ISR increments the counts in a number of RTCSWTs (real-time clock interrupts-triggered software timers). Assume that there is a list of RTCSWT timers that are active at an instant. The top of the list can be pointed as '*RTCSWT_List.top' using the pointer. RTCSWT_List.top is

now the pointer to the top of the contents in a memory for a list of the active RTCSWTs. Consider the statement 'RTCSWT_List.top ++;' It increments this pointer in a loop. It will not point to the next top of another object in the list (another RTCSWT) but to some address that depends on the memory addresses allocated to an item in the RTCSWT_List. Let *ListNow* be a pointer within the memory block of the list top element. A statement '*RTCSWT_List. ListNow = *RTCSWT_List.top;' will do the following. RTCSWT_List pointer is now replaced by RTCSWT list top pointer and now points to the next list element (object). (*Note*: RTCSWT_List.top ++ for pointer to the next list object can only be used when RTCSWT_List elements are placed in an array. This is because an array is analogous to consecutively located elements of the list at the memory).

Consider a statement: 'while (* RTCSWT_List. ListNow -> state != NULL) {numRunning ++};'. When a pointer to ListNow in a list of software timers that are running at present is not NULL, then only execute the set of statements in the given pair of opening and closing curly braces. One of the important uses of the NULL pointer is in the last element of the list to point to the end of a list, or to no more contents in a queue or empty stack, queue or list.

A list is a data structure in which each element (object or data structure) also stores a pointer to the next element at the list. It has one memory block allotted to each element. The list top pointer points to first element and the last element points to NULL.

Table 5.2 summarizes the exemplary uses of queues, stacks, arrays, lists and trees.

Table 5.2 Uses of the Various Data Structures in a Program Element

Data Structure	*Definition and when used*	*Example(s) of its use*
Queue	It is a structure with a series of elements with a header element waiting for a read operation, called deletion operation. An operation can be done only in the first in first out (FIFO) mode. It is used when an element is not to be accessible by any index and pointer directly, but only through the FIFO. An element can be inserted only at the end in the series of elements waiting for an operation. There are two pointers, one for deleting after the operation and the other for inserting. Both increment after an operation.	(i) Print buffer. Each character is to be printed in the FIFO mode. (ii) Frames on a network (Each frame also has a queue of a stream of bytes). Each byte has to be sent for receiving as a FIFO. (iii) Image frames in a sequence. (these have to be processed as a FIFO).
Stack	It is a structure with a series of elements with its last element waiting for an operation. An operation can be done only in the last in first out (LIFO) mode. It is used when an element is not to be accessible by any index or pointer directly, but only through the LIFO. An element can be pushed (inserted) only at the top in the series of elements still waiting for an operation. There is only one pointer used for pop (deleting) after the operation as well as for push (inserting). Pointers increment or decrement after an operation. It depends on insertion or deletion.	(i) Pushing of variables and context on interrupt or call to another function. (ii) Retrieving popping the pushed data from a stack.

(*Contd*)

Data Structure	*Definition and when used*	*Example (s) of its use*
Array (one-dimensional vector)	It is a structure with a series of elements with each element accessible by identifier name and index. Its element can be used and operated easily. It is used when each element of the structure is to be given a distinct identity by an index for easy operation. Index stars from 0 and is a positive integer.	ts = 12 * s(1); Total salary, ts is 12 times the first month salary. marks_weight [4] = marks_weight [0]; weight of marks in the subject with index 4 is assigned the same as in the subject with index 0.
Multi-dimensional array	It is a structure with a series of elements each having another sub-series of elements. Each element is accessible by the identifier name and two or more indices. It is used when every element of the structure is to be given a distinct identityby two or more indices for easy operation. The dimension of an array equals the number of indices that are needed to distinctly identify an array element. Indices start from 0 and are positive integers.	Handling a matrix or tensor. Consider a pixel in an image frame. Consider Quarter-CIF image pixel in 144 × 176 size image frame (recall Section 1.2.7.) *pixel* [108, 88] will represent a pixel at the 108-th horizontal row and the 88-th vertical column. [1]See the following note also.
List	Each element has a pointer to its next element. Only the first element is identifiable and the list top pointer (header) does it. No other element is identifiable and hence is not accessible directly. By going through the first element, and then consecutively through all the succeeding elements, an element can be read or read and deleted or can be added to a neighbouring element or replaced by another element.	A series of tasks which are active. Each task has pointer of the next task. Another example is a menu that points to a sub-menu.
Tree	There is a root element. It has two or more branches each having a daughter element. Each daughter element has two or more daughter elements. The last one does not have daughters. Only the root element is identifiable and it is done by the treetop pointer (header). No other element is identifiable and hence is not accessible directly. By traversing from the root element, then proceeding continuously through all the succeeding daughters, a tree element can be read or read and deleted or can be added to another daughter or replaced by another element. A tree has data elements arranged as branches. The last daughter, called leaf node has no further daughters. A binary tree is a tree with a maximum of two daughters (branches) in each element and none at leaf-element.	An example is a directory. It has number of file folders. Each file folder has a number of other file folders and so on. In the end is a file.

[1] pixel [0,0] represents the pixel at the left corner on the top and pixel [143, 175] represents that at the right bottom. pixel [10,108, 88] is a pixel data element in a three-dimensional array form. It represents pixels at the same position (108 × 88) in the tenth frame.

The reader may refer to a standard textbook for the C and C++ data structure algorithms.

5.4.4 Use of Modifiers

The actions of modifiers are as follows:

Case (i): Modifier *'auto'* or no modifier means that there is ROM allocation for the variable by locator if it is initialized in the program. RAM is allocated by the locator, if it is not initialized in the program.

Case (ii): Modifier '*unsigned*' is a modifier for a short or int or long data type. It is a directive to permit only the positive values of 16, 32 or 64 bits, respectively.

Case (iii): Modifier '*static*' declaration is inside a function block. Static declaration is a directive to the compiler that the variable should be accessible outside the function block also, and there is to be a reserved memory space for it. It then does not save on a local parameter stack. When several tasks are executed in cooperation, the declaration *static* helps. Consider an exemplary declaration, 'private: static void interrupt ISR_RTI ( ). The static declaration here is for the directive to the compiler that the ISR_RTI ( ) function codes limit to the memory block for ISR_RTI ( ) function. The private declaration here means that there are no other instances of that method in any other object. It then does not save on the stack. There is ROM allocation by the locator if it is initialized in the program. There is RAM allocation by the locator if it is not initialized in the program.

Case (iv): Modifier *static* declaration is outside a function block. It is not usable outside the class or module in which it is declared. There is ROM allocation by the locator for the function codes.

Case (v): Modifier *const* declaration is outside a function block. It must be initialized by a program. For example, #define *const* Welcome_Message 'There is a mail for you'. There is ROM allocation by the locator.

Case (vi): Modifier *register* declaration is inside a function block. It must be initialized by a program. For example, '*register* CX'. A CPU register is temporarily allocated when needed. There is no ROM or RAM allocation.

Case (vii): Modifier *interrupt*: It directs the compiler to save all processor registers on entry to the function codes and restores them on their return from the function (this modifier is prefixed by an underscore, '_interrupt' in certain compilers).

Case (viii): Modifier *extern*: It directs the compiler to look for the data type declaration or the function in a module other than the one currently in use.

Case (ix): Modifier *volatile* outside a function block is a warning to the compiler that an event can change its value or that its change represents an event. An event example is an interrupt event, hardware event or inter-task communication event. For example, consider a declaration: '*volatile* Boolean IntrEnable;' It changes to false at the start of service by a service routine, if true previously. The compiler does not perform optimization for a *volatile* variable. Let a variable be assigned, c = 0. Later, it is assigned c = 1. The compiler will ignore the statement *c* = 0 during code optimization and will take c = 1. But if c is an event variable, it should not be optimized. IntrEnable = 0 is at the beginning of service routine in case an interrupt-enabled variable is used for disabling any interrupt during the period of execution of ISR. IntrEnable = 1 is executed before return from the ISR. This re-enables the interrupts at the system. Declaration of IntrEnable as volatile directs the compiler not to optimize two assignment statements in the same function. There is no ROM or RAM allocation by the locator.

Case (x): Modifier *volatile static* declaration is inside a function block. Examples are: (a) '*volatile static* boolean RTIEnable = true'; (b) '*volatile static* boolean RTISWTEnable'; and (c) '*volatile static* boolean RTCSWT_F.' The static declaration is for directive to compiler that the variable should be accessible outside the function block also, and there is to be a reserved memory space for it. The volatile is a directive that cannot optimize as an event can modify. It then does not save onto the local parameter stack of the function. When several tasks are executed in cooperation, the declaration static helps. The compiler does not optimize the code because of declaration volatile. There is no ROM or RAM allocation by the locator.

5.4.5 Use of Loops, Infinite Loops and Conditions

Sometimes a set of statements is repeated in a loop. Generally, in case of array, the index changes and the same set is repeated for another element of the array. Loops are used when executing a set of statements repeatedly. A loop starts from an initial value or condition and executes till the limiting condition is fulfilled. There can be certain parameter, which changes each time from its initial condition up to a limiting condition.

For example, consider the following. `for (i = 0; i < = 100; i++) { /*` a set of statements which repeatedly execute `*/ }`. The initial condition is assigned as i = 0 and the last condition for the loop to execute till i is less or equal to 100. The set of statements in the bracket executes from start to end and before return to start the i increments by 1. The *for* statement allows set of statements to repeatedly execute 101 times with values of i = 0, 1,..., 99, 100.

For another example, consider the following. `i = 0; while (i < = 100){ { /*` a set of statements which repeatedly execute `*/ }; i++;}`. The initial condition is assigned as i = 0 and is set before the *while* loop. The *while* loop executes till i remains less or equal to 100. i++ increments before the return to test the *while* condition. The *while* statement allows the set of statements to repeatedly execute with values of i = 0, 1,..., 99, 100.

If a condition remains true, then *while* loop will execute infinitely. For example, *while* (1){ { /* a set of statements which execute repeatedly execute */ }. The loop will execute infinitely because 1 is always true. Infinite loops are never desired in usual programming. Why? The function or task will never end and never exit or proceed further to the codes after the loop. *Infinite loop is a feature in embedded system programming!* The system software in the telephone has to be always in a waiting loop that finds the ring on line. An exit from the loop will make the system hardware redundant.

Example 5.7 gives a C program design in which the program starts executing from the main ( ) function. There are calls to the functions and calls on interrupts in-between. It has to return to the start. The system main program is never in a halt state. Therefore, the main ( ) is in an infinite loop within the start and end.

Example 5.7

```
# define false 0
# define true 1
/*************************************************************************/
void main (void) {
/* The Declarations here and initialization here */
.
.
/* Infinite while loop follows. Since the condition set for the while loop is always true, the statements
within the curly braces continue to execute */
while (true) {
/* Codes that repeatedly execute */
.
.
}
```

Example 5.8 gives an example for use of polling for an event or message in a program.

Assume that the function *main* has a waiting loop and simply passes the control to an RTOS. Each task controlled by the RTOS will also have codes in an infinite loop. Example 5.9 demonstrates the infinite loops within each task.

Example 5.8

```
# define false 0
# define true 1
/*************************************************************************/
```

```
void main (void) {
/* Call RTOS run here */
rtos.run ( );
while (1) {
/* Infinite while loops follows in each task. So never there is return from the RTOS. */
}
}
/*************************************************************************/
void task1 (....) {
/* Declarations */
.
.
while (true) {
/* Codes that repeatedly execute */
.
.
/* Codes that execute on an event */
if (flag1) {....;}; flag1 =0;
/* Codes that execute for sending message to the kernel */
message1 ( );
}
}
/*************************************************************************/
void task2 (...) {
/* Declarations */
.
.
while (true) {
/* Codes that repeatedly execute */
.
.
/* Codes that execute on an event */
if (flag2) {.....;}; flag2 =0;
/* Codes that execute for sending message to the kernel */
message2 ( );
};
}
/*************************************************************************/
.
.
.
/*************************************************************************/
void taskN (...) {
/* Declarations */
```

```
.
.
.while (true) {
/* Codes that repeatedly execute */
.
.
/* Codes that execute on an event*/
if (flagN) {....;}; flagN =0;
/* Codes that execute for sending message to the kernel */
messageN ( );
};
}
```

There can be more than one infinite loops. The code inside infinite-loop waits for an inter-process-communication (IPC) message or event flag or a set of events through the OS.

The code inside the loop of the running task generates a message that transfers to the kernel. The OS kernel, which passes to the waiting task message, detects it and when that task starts the OS pre-empts the previously running task.

Let an event be setting of a flag, and the flag setting is to trigger the running of a task whenever the kernel passes it to the waiting task. The instruction SWI executes to send the message to the another task function for a service.

Conditional statements are used very often. If a defined condition(s) is fulfilled, the statements within the curly braces after the condition (or a statement without the braces) are executed, otherwise the program proceeds to the next statement or to the next set of statements.

A set of statements is called switch-case. A program switches to a case as per the result of switch expression result. For example, Switch (*i*) means switch as per the case for the value of i. Example 5.9 shows an application of infinite loop and switch case statement for programming for GUI in mobile phones (Example 1.5.4).

Example 5.9

Consider a smart mobile phone (Example 1.5.4). Assume that the screen state j is between 0 and K, among 0, 1, 2, … or K – 1 possible states (set of menus). An interrupt is triggered from a touch screen GUI and an ISR posts an event message m = 0, 1, 2, …, or N – 1 as per the selected the menu choice 0, 1, 2, …, N – 1 when there are N menu choices for a mobile phone user to select from a screen in state j. The m will depend on the screen position at the touched position. Figure 5.4 shows the use of a programming model here, which facilitates execution of one of the multiple possible function calls; a function executes after polling for screen state j and for a message m from an ISR as per the user choice

```
# define true 1
# define false 0

/*************************************************************************/
void main (void) {
/* declarations */
while (true ) { /* Execute infinite loop */
poll_Screen_State (i); /* Call a function to poll screen state. A state means a set of choices of menu
displayed on the touch screen */
```

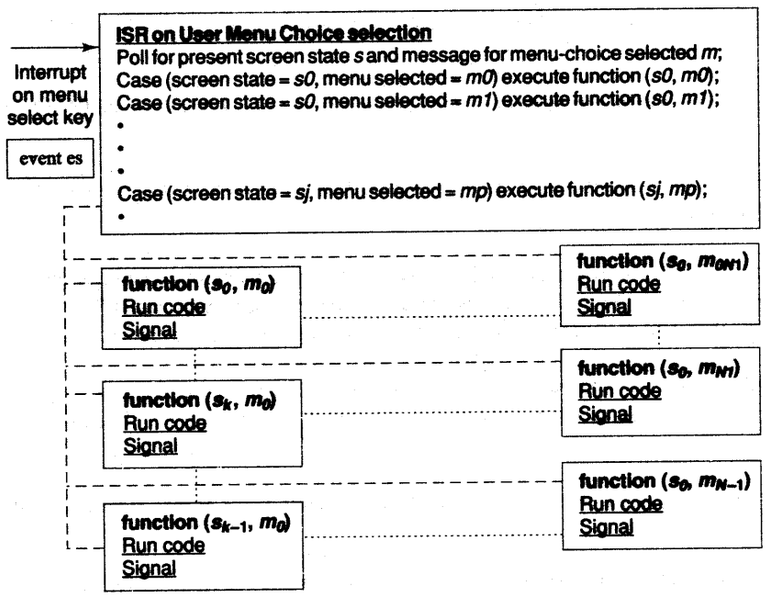


Fig. 5.4 Programming model use here, which facilitates execution of one of the multiple possible function calls and the function executes after polling for screen state j and for a message m from an interrupt service routine as per the user choice

```
}
/*********************************************************************/
void poll_Screen_State (j)
/* Let number j identify a screen state */
Switch (i) {
Case 0: poll_menu0 ( ); exit ( )
Case 1: poll_menu1 ( ); exit ( )
.
.
Case j: poll_menuJ ( ); exit ( )
.
.
Case K: poll_menuK ( ); exit ( )
}
}
/*********************************************************************/
void poll_menu0 {/* Code for polling for choice from menu 0 for screen state 0 */ }
/* An ISR sends message m as per the choice selected by the user from the menu in screen state j */
Switch (m ) {
Case 0: {/*Code, which executes when the choice is menu 0 Screen state 0*/ ; exit ( );}
```

```
Case 1: {/*Code, which executes when the choice is menu 1 Screen state 0*/ */ ; exit ( );}
.
.
Case N – 1: {/*Code, which executes when the choice is menu N – 1 Screen state 0*/ */ ; exit ( );}
   }
}
/*********************************************************************/
. /* Codes for Screen state 1, 2, ..., j */
.
/* An ISR sends message m as per the choice selected by the user from the menu in screen state j */
void poll_menuJ {/* Code for polling for choice from menu m for screen state J */
/* An ISR sends message m as per the choice selected by the user from the menu in screen state j */
Switch (m ){
Case 0: {/*Code, which executes when the choice is menu 0 Screen state j*/ ; exit ( );}
Case 1: {/*Code, which executes when the choice is menu 1 Screen state j*/ */ ; exit ( );}
.
.
Case N – 1: {/*Code, which executes when the choice is menu N – 1 Screen state j*/ */ ; exit ( );}
   }
}
/*********************************************************************/
. /* Codes for Screen state j + 1, 2, ..., K – 1 */
.
void poll_menuK {/* Code for polling for choice from menu m for screen state K – 1*/
}
}
/*********************************************************************/
```

5.4.6 Use of Function Calls

Table 5.1 gave the meanings of the various sets of instructions in the C program. There are special functions for starting the execution of a program, 'void main (void)'. Given next are the steps to be followed when using a function in the program.

1. *Declaring a function:* Just as each variable has to have a declaration, each function must be declared.

Example 5.10

1. Declare a function as follows: 'int **run** (int indexRTCSWT, unsigned int maxLength, unsigned int numTicks, SWT_Type swtType, SWT_Action swtAction, boolean loadEnable);' The **run** is the function name. Here *int* specifies the returned data type. There are arguments inside the brackets. Data type of each argument is also declared. A modifier is needed to specify the data type of the returned element (variable or object) from any function. Here, the data type is specified as an integer. (A modifier for specifying the returned element may also be *static*, *volatile*, *interrupt* and *extern*.)
2. Consider a device driver function open (fd, options, device_parameter). The called function name is 'open'. It sets the device configuration. When the function is called by statement, open (4, O_RDWR, 9600). First, second and third arguments that are passed, are 4, O_RDWR and 9600. First argument

is for the device descriptor and passes the value fd = 4. The descriptor is an identity, which is an integer number. The second argument describes the device option setting as read and write device. The third argument describes the device parameter, baud_rate, the rate by which the serial line device is to be configured for UART communication. The same open function is used for the other options and parameters of the device. For example, open (4, O_RD, 1200) devices, which means that the device 4 is read only and the device parameter is 1200.

2. *Defining the statements in function:* Just as each variable has to be given the contents or value, each function must have statements. Consider the statements of the function 'run'. These are *within a pair of curly braces* as follows: 'int RTCSWT:: run (int indexRTCSWT, unsigned int maxLength, unsigned int numTicks, SWT_Type swtType, SWT_Action swtAction, boolean loadEnable) {...};'. The last statement in a function is for the *return* and may also be for returning an element or data structure or object.
3. *Call to a function:* Consider an example: 'if (delay_F = = true && SWTDelayIEnable == true) ISR_Delay (100);'. There is a call on fulfilling a condition. The call can occur several times and can be repeatedly made. On each *call*, the values of the arguments given within the pair of brackets pass for use in the function statements. There is only one argument in ISR_Delay.

Given next are the steps in transfer of values from the arguments of calling function to called function's arguments.

(1) *Passing the Values (elements)*: The values are copied from argument in calling to called function argument. When the function is executed in this way, it does not change a variable's value at the calling function on return from the called function. A function can only use the copied values as its own variables through the arguments.

Example 5.11

Consider a statement, 'run (int indexRTCSWT, unsigned int maxLength, unsigned int numTicks, SWT_Type swtType, SWT_Action swtAction, boolean loadEnable) {...}'. Function 'run' arguments indexRTCSWT, maxLength, numTick, swtType and loadEnable original values in the calling program will remain unchanged during execution of the codes. The advantage is that the same values for use in the remaining instructions are present on return to the calling function. The arguments that are *passed by the values* are saved temporarily on a local parameter stack and retrieved on return from the called function.

(2) *Reentrant functions call*: Re-entrant function is usable by the several tasks and routines synchronously (at the same time). This is because all its argument values are retrievable from a stack of the local variables, data structures and objects. A function is called re-entrant function when the following three conditions are satisfied.

(i) All the arguments pass the values and none of the argument is a pointer (address) whenever a calling function calls it. There is no pointer as an argument in the Example 5.11 of function 'run'.

(ii) When an operation is not atomic, the function should not operate on any variable, which is declared outside the function or which an ISR uses or which is a global variable but passed by reference and not passed by value as an argument into the function. (The value of such a variable or variables, which is not local, does not save on the stack when there is a call to another program.)

Let us understand *atomic operation*. The following is an example that clarifies it further. Assume that at a server (software), there is a 32-bit variable *count* to count the number of clients (software) needing service. There is no option except to declare the *count* as a global variable that shares with all clients. Each client on

a connection to a server initiates increment of *count*. Count increment operation should be atomic and till it completes the server disables requests from other clients. Assume that a service routine for real-time clock tick increments the clock ticks in numTicks, which is a 64-bit variable. All operations using four or eight bytes of the variable represent one atomic unit. The implementation by the assembly code for increment at that memory location becomes non-atomic in the following situation. Assume that the processor is of 8 bits, and therefore the 32- or 64-bit increment is done in four or eight operations, respectively. Assume that there is no disabling of interrupts or execution of other routines or tasks till all operations for 64-bit increment are complete. Now if an interrupt occurs in-between the incrementing, the wrong values of count can be passed.

(iii) That function does not call any other function that is not itself re-entrant.

(3) ***Passing the references***: When an argument value to a function passes through a pointer, the called function can change this value. On returning from this function, the new value will be available in the calling program or another function called by this function. This is because there is no saving on stack of a value that either *passes through a pointer in the function arguments* or *operates on the function on a global variable* or *operates through a variable declared outside the function block.*

5.4.7 Multiple Function Calls in Cyclic Order

One of the most common methods is multiple function calls made in a cyclic order in an infinite loop.

Example 5.12

Assume 64 kbps network (Example 4.1). Figure 5.5 shows the model of the multiple function *calls*.

```
typedef unsigned char int8bit;
# define int8bit boolean
# define false 0
# define true 1

void main (void) {
/* The Declarations of all variables, pointers, functions here and also initializations here */
```

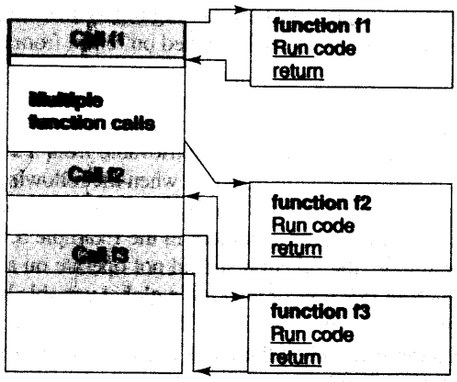


Fig. 5.5 Programming model of multiple function calls

```
unsigned char *portAdata;
boolean charAFlag;
boolean checkPortAChar ( ); /* An interrupt service function to return a Boolean flag, if there is character
received at port A */
/* A declarations for the functions */ void inPortA (unsigned char *);
void decipherPortAData (unsigned char *);
void encryptPortAData (unsigned char *);
void outPortB (unsigned char *);
.
while (true) {
/* Codes that repeatedly execute */
/* Function for availability check of a character at port A*/
    while (charAFlag != true) checkPortAChar ( );
/* Function for reading PortA character*/
    inPortA (unsigned char *portAdata);
/* Function for deciphering */
    decipherPortAData (unsigned char *portAdata);
/* Function for encoding */
    encryptPortAData (unsigned char *portAdata);
/* Function for retransmit output to PortB*/
    outPort B (unsigned char *portAdata);
    };
}
/*************************************************************************/
/* An interrupt service function to return a Boolean flag, if there is character received at port A */
boolean charAFlag;
boolean checkPortAChar ( );
void inPortA (unsigned char *{...}; /* The ISR, which gets the input character at the address
portAdata */
/*************************************************************************/
```

5.4.8 Function Pointers, Function Queues and ISR Queues

Let the * sign not be put before a function's name and there are arguments within a pair of brackets after the name. The statements for the function execute using the argument values or references for data variables. The statements are present inside a pair of the curly braces. Consider a declaration in Example 5.12, 'boolean checkPortAChar_( );'. 'checkPortAChar' is a function, which returns a boolean value. Now, checkPortAChar is itself a pointer to the code's starting address. The address has the codes for statements. The PC will fetch the address of checkPortAChar when a call the function is made, and the CPU sequentially executes the function statements from here.

Now, let the * sign be put before the function's name. '* checkPortAChar' will now refer to all the compiled statements in the memory that are specified within the curly braces.

Consider a declaration in the example, 'void inPortA (unsigned char *);'.

1. inPortA means a pointer to the statements of the function. Inside the bracket, there is an unsigned character pointed by some pointer.

2. *inPortA will refer to all the compiled statements of inPortA.
3. (* inPortA) will refer to calling of statements of inPortA.
4. What will a statement, 'void create (void (*_inPort_A) (unsigned char *), void *portAStack, unsigned char portApriority);' mean?
   a. First modifier 'void' means *create* a function, which does not return any thing.
   b. 'create' is the function, which can be called after its declaration in a statement.
   c. Consider first argument of this function 'void (*inPortA) (unsigned char *portAdata)'. (*inPortA) means to call the statements of inPortA the argument of which is 'unsigned char *portAdata'.
   d. The second argument of *create* function is a pointer for the portA stack at the memory.
   e. The third argument of *create* function is a byte that defines the portA priority.

An important lesson to be remembered from this discussion is that a returning data type specification (e.g., void) followed by '(*functionName) (functionArguments)' calls the statements of the functionName using the functionArguments, and on a return it returns the specified data object. We can thus use the function pointer for invoking a call to the functions in a C function.

5.4.9 Queuing of Functions on Interrupts

When there are multiple ISRs, a high priority ISR is executed first and the lowest priority, in the end. (Refer Section 4.5). It is possible that function calls and statements in any of the higher priority interrupts may block the execution of low priority ISR and there may be deadline miss for the low priority ISR. Using the function pointers in the routines, and forming a queue (FIFO) for the function pointers is a solution for the deadline problem for low priority routines. The queued functions are then executed at a later stage. The queued functions are the deferred procedure calls (DPCs) and can be called ISTs when the OS handles them as threads (Section 7.3). Figures 5.6(a), (b) and (c) show the main function, function multiple calls calling the FunctionQueues and ISR, respectively. The figure shows a programming model example in which the multiple function pointers are queued by the ISRs and device-driving ISRs. Each ISR statements have a short set of codes. It executes essential codes within the ISR and rest of the codes in queued functions. Figure 5.6(d) shows queue of function pointers.

Example 5.13

The following is code in which on a return from a service routine, the operation function '*operation*FunctionQueues ( )' gets the function pointers from the queue and then executes the pointed functions.

```
/* Insert here all preprocessor directives, commands and functions except the main and portA_ISR_Input
( ) functions. */
void main (void) {
/* The Declarations of all variables, pointers, functions here and also initializations */
.
.
while (true) {operationFunctionQueues ( ); /*Call Functions from the Queue in cyclic (Round Robin)
Mode*/
      };
}
/*************************************************************************/
void operationFunctionQueues ( ) {
unsigned char *portAdata;
```

```
boolean checkPortAChar ( );
void inPortA (unsigned cha. ');
void decipherPortAData (unsigned char *);
void encryptPortAData (unsigned char *);
void outPortB (unsigned char *);
void checkPortAChar ( );
/* In_A_Out_B is an array, which inserts the queue elements */
QueueElArray In_A_Out_B = new QueueElArray (QElType * QelementsArray, 65536);
portAIF = false; portAIEnable = true;
/* Codes that repeatedly execute */
while (portAFlag != true) checkPortAChar (In_A_Out_B, STAF);
In_A_Out_B.QElReturn ( ); /* Get an In_A_Out_B queue element */
In_A_Out_B.QElReturn ( ); /* Get next In_A_Out_B queue element */
In_A_Out_B.QElReturn ( ); /* Get next In_A_Out_B queue element */
In_A_Out_B.QElReturn ( ); /* Get next In_A_Out_B queue element */
};
/*******************Interrupt Service Routine ********************/
```

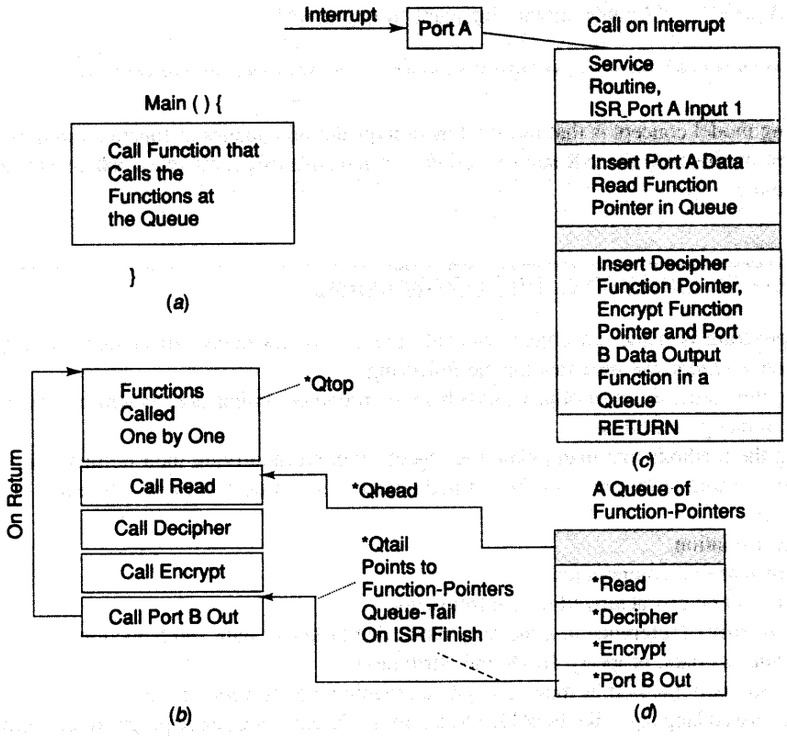


Fig. 5.8 (*a*) Main ( ) function (*b*) Function 'opertionFunctionQueues ( )' (*c*) Creation of a queue of the function pointers by the interrupt service routine (*d*) Queue of function pointers

```
void checkPortAChar (QueueElArray In_A_Out_B, volatile boolean portAIF) {
while (portAIF != true) { }; /*Wait till the occurrence of Port A Interrupt */
/* Call ISR_PortAInputI, an Interrupt Service Routine called on the port interrupt */
ISR_PortAInputI (QueueElArray In_A_Out_B);
}
/*******************Interrupt Service Routine ********************/
void interrupt ISR_PortAInputI (QueueElArray In_A_Out_B) {
disable_PortA_Intr ( ); /* Disable another interrupt from port A*/
void inPortA (unsigned char *portAdata); /* Function for retransmit output to Port B*/
void decipherPortAData (unsigned char *portAdata); /* Function for deciphering */
void encryptPortAData (unsigned char *portAdata); /* Function for encrypting */
void outPortB (unsigned char *portAdata); /* Function for Sending Output to Port B*/
/* Insert the function pointers into the queue */
In_A_Out_B.QElinsert (const    inPortA & *portAdata);
In_A_Out_B.QElinsert (const decipherPortAData & *portAdata);
In_A_Out_B.QElinsert (const encryptPortAData & *portAdata);
In_A_Out_B.QElinsert (const outPortB & *portAdata);
enable_PortA_Intr ( ); /* Enable another interrupt from portA*/
}
      /*************************************************************/
```

A programming model concept is that use the function queues and queues of function pointers built by the ISRs. It reduces significantly the ISR latency periods. Each device ISR is therefore able to execute within its stipulated deadline.

5.5 OBJECT-ORIENTED PROGRAMMING

When a large program is made, an object-oriented language offers many advantages. An *object-oriented programming (OOP) language* provides for the following:

1. Defining the object or set of objects, which are common or similar objects within a program and can be used in many programs.
2. Defining the methods that manipulate the objects without modifying their definitions.
3. Creation of multiple instances of the defined object or set of objects or new objects.
4. Inheritance.
5. Data encapsulation.
6. Design of reusable components.

   An object can be characterized by the following.
   (a) An *identity* (a reference to a memory block that holds its state and behaviour).
   (b) A *state* (its data, property, fields and attributes).
   (c) A *behaviour* (method or methods that can manipulate the *state* of the object).

In a procedure-based language, like FORTRAN, COBOL, Pascal and C, large programs are split into simpler functional blocks and statements. In an object-oriented language like Smalltalk, C++ or Java, the logical groups (also known as *classes*) are first made. Each group defines the data and the methods of using the data. A set of these groups then gives an application program. Each group has internal user-level fields for the data and the

methods of processing that data at these fields. Each group can then create many objects by copying the group and making it functional. Each object is functional. Each object can interact with other objects to process the user's data. The language provides for formation of classes by the definition of a group of objects having similar attributes and common behaviour. A class creates the objects. An object is an instance of a class.

5.6 EMBEDDED PROGRAMMING IN C++

5.6.1 Advantages of C++

C++ is an OOP language, which in addition, supports the procedure-oriented codes of C. Program coding in C++ codes provides the advantage of OOP as well as the advantage of C and in-line assembly. Programming concepts for embedded programming in C++ are as follows:

1. A class binds all the member functions together for creating objects. The objects will have memory allocation as well as default assignments to its variables that are not declared *static*. Let us assume that each software timer is an object. It gets the count input from a real-time clock. It has a terminal count value after which it generates a software interrupt. It is initialized to a count value. Now consider the codes for a C++ class RTCSWT. A number of software timer objects can be created as the instances of RTCSWT. Each instance of RTCSWT can have different values of present, initial and terminal counts but has identical methods to manipulate the count.
2. A class can derive (inherit) from another class also. Creating a *child* class from RTCSWT as a *parent* class creates a new application of the RTCSWT.
3. Methods (C functions) can have the same name in the inherited class. This is called *method overloading*. Methods can have the same name as well as the same number and type of arguments in the inherited class. This is called *method over-riding*. These are the two significant features that are extremely useful in a large program.
4. Operators in C++ can be overloaded like method overloading. Following statements show how the operators ++ and ! are overloaded to perform a set of operations.(Usually the ++ operator is used for postincrement and preincrement and the ! operator is used for a *not* operation.)

```
const OrderedList & operator ++ ( ) {if (ListNow != NULL) ListNow =
ListNow -> pNext;
return *this;}
boolean int OrderedList & operator ! ( ) const {return (ListNow != NULL)
;};
```

(Java does not support operator overloading, except for the 'plus' operator, which is used for summation as well string concatenation.)

There is *struct* that binds all the member functions together in C. But a C++ *class* has object features. It can be extended and child classes can be derived from it. A number of child classes can be derived from a common class. This feature is called polymorphism. A class can be declared as public or private. The data and methods' access are restricted when a class is declared private. *Struct* does not have these features.

5.6.2 Disadvantages of C++

Program codes become lengthy, particularly when following features of the standard C++ are used.

1. Template.

2. Multiple inheritance (deriving a class from many parents).
3. Exceptional handling.
4. Virtual base classes.
5. Classes for IO streams. [Two library functions are *cin* (for character(s) in) and *cout* (for character(s) out).] The I/O stream class library provides for the input and output streams of characters (bytes). It supports *pipes*, *sockets* and *file management features*.

5.6.3 Optimization of Codes in Embedded C++ Programs to Eliminate the Disadvantages

Embedded system codes can be optimized when using an OOP language by the following

1. Declare private as many classes as possible. It helps in optimizing the generated codes.
2. Use *char*, *int* and *boolean* (scalar data types) in place of the objects (reference data types) as arguments and use local variables as much as feasible.
3. Recover memory already used once by changing the reference to an object to NULL.

A *special compiler for an embedded system* can facilitate the disabling of specific features provided in C++. Embedded C++ is a version of C++ that provides for a selective disabling of the aforementioned features so that there is a less run-time overhead and less run-time library. The solutions for the library functions are available and ported in C directly. The IO stream library functions in an embedded C++ compiler are also re-entrant. Hence using embedded C++ compilers or the special compilers make the C++ a significantly more powerful coding language than C for embedded systems.

GNU C/C++ compilers (called *gcc*) find extensive use in C++ environment in embedded software development. Embedded C++ is a new programming tool with a compiler that provides a small run-time library. It satisfies small run-time RAM needs by selectively de-configuring features like template, multiple inheritance, virtual base class and so on, when there is a less run-time overhead and when the less run-time library-using solutions are available. Selectively removed (de-configured) features could be template, run-time type identification, multiple inheritance, exceptional handling, virtual base classes, IO streams and foundation classes. [Examples of foundation classes are GUIs (graphic user interfaces). Exemplary GUIs are buttons, checkboxes or radios.]

An embedded system C++ compiler (other than *gcc*) is Diab compiler from Diab Data. It also provides the target (embedded system processor) with specific optimization of the codes. The run-time analysis tools check the expected run-time error and give a profile that is visually interactive.

Embedded system programmers use C++ due to the OOP features of software reusability, extendibility, polymorphism, function over-riding and overloading along with the portability with the C codes and in-line assembly codes. C++ also provides for overloading of operators. Embedded C++ is a C++ version, which makes large program development simpler by providing OOP features of using an object, which binds state and behaviour and which is defined by an instance of a class. We use objects in a way that minimizes memory needs and run-time overheads in the system.

5.7 EMBEDDED PROGRAMMING IN JAVA

5.7.1 Java Programming Basics

Java programming starts from coding for the classes. A class has members. A field is like a variable or struc in C. A method defines the operations on the fields, similar to function in C. Table 5.3 summarizes the basic

uses and the exemplary uses. Class instance fields and instance methods are the members, whose new instances are also created as when the objects are created from the class. Class is a named set of codes that has a number of members – data fields (variables), methods (functions), and so on, so that the object can be created from it. The operations are performed on the objects by passing the messages to objects in OOP. Each class is a logical group with identity, state and behaviour specifications. For an in-depth learning of the programming language, a reader should refer to the standard textbooks and do the required practice exercises.

Table 5.3 Various Elements in a Java Program

Java Program Element	*Explanation*	*Example(s) of its use*
Local variable	A variable within a block of codes is defined inside the curly braces and has limited scope.	`{for (int i = 0; int totalOfMarks = 0; i<5; i++) {totalOfMarks + = subjectMarks[i];}; return totlaOfMarks;}.` Here *i* is the local variable. The *i* does not have any scope outside the *for* loop.
Instance method	Blocks of Java codes, which are given a name, a call (invocation) is made by other Java codes that can also pass (transmit) the needed reference to the values, parameters, and so on.	`findTotalMarks ( ) { };` The method find TotalMarks ( ) { } will also be created in object created from the class.
Instance field	An identifier with a name and using that name a declaration is made in a Java class. It does have a default value and the field is also present in the objects which are instances of the class.	`String tele_number;` Here, tele_number is an instance field of the class and will also be created in the objects created from that class.
Class	A class is a basic structural unit in a Java program. A class consists of data fields and methods that operate on the fields. A class defines a group of objects with similar attributes and common behaviour and relationships. A class is used to create objects as its instances. It has instance and static fields and methods.	`public class Salaries { public float monthly salary, totalSalary; public float findTotalSalary ( ) { }; }`
Inheritance	Java class inherits members when a Java class is extended from a parent class called super class. The inherited instance fields and methods can be over-ridden by redefining them in extended class using same name, arguments and argument-types. Methods can be overloaded by redefining them for different numbers or types of arguments.	`public class AccountDetails extends BankDetails {…};.` The class AccountDetails will inherit members of class BankDetails.
Interface	Interface has only the abstract methods and the corresponding static data fields and the methods do not have implementation in the interface. A Java class which is interfaced to an interface implements the abstract methods specified at the interface.	`public class AccountDetails extends BankDetails implements InterestComputations{…}.`

(Contd)

Java Program Element	*Explanation*	*Example(s) of its use*
Data types	Java class uses primitive data types: byte (8-bit), short (16-bit), int (32-bit), long (64-bit), float, double, unicode char (16-bit). Java class uses reference data types. A reference can be to the class type in which there are groups of fields and methods to operate on the fields. A reference can be to the array type in which there are groups of objects as array elements.	`byte portData; /* 8-bit port data */ short counts; /* 16-bit count data */ int num Ticks; /* 32-bit number of clock ticks */ String accountNum, eMailId; /* account Number and email ID as String class objects */`
Exception	Java has built-in exception classes. The occurrences of exceptional conditions are handled when exception is thrown. It is also possible to define exception conditions in a program so that exceptions are thrown from try block codes and caught by catch exception method . (Section 4.2.2).	`java.lang.ArrayIndexOutOf BoundExceptions: 0 at addArray (....` This throws an exception. javal.lang package has an Object java.lang.Throwable. We can also define exceptions in try `{..} catch (Exception e1){ }` Finally { }; (Example 4.6).

5.7.2 Java Programming Advantages

Java has advantages for embedded programming as follows:

1. Java is completely an OOP language. Java program starts with classes. Application program consists of classes, objects and interfaces.
2. There is a huge class library on the network that makes program development quick.
3. Java has extensibility.
4. Java has in-built support for creating multiple threads. It obviates the need for an OS-based scheduler for handling threads.
5. Java generates byte codes. These execute on an installed JVM (Java virtual machine) on a machine. Virtual machine takes the Java byte codes in the input and runs on the given platform (processor, system and OS). [Virtual machine (VM) in embedded systems is stored at the ROM.] Therefore, Java codes can host on diverse platforms. Platform independence in hosting the compiled codes permit Java for network applications.
6. Platform independence gives *portability* with respect to the processor and OS used. Java is considered as write once and run anywhere.
7. Java is the language for most Web applications and allows machines of different types to communicate on the Web.
8. Java is easier to learn by a C++ programmer.
9. Java does not permit pointer manipulation instructions. So it is robust in the sense that memory leaks and memory-related errors do not occur. A memory leak occurs, for example, when attempting to write after the end of a bounded array.
10. Java does not permit dual way of object manipulation by value and reference. There are no struc, enum, typedef and union. Java does not permit multiple inheritances. Java does not permit operator overloading except for the 'plus' sign used for string concatenation.

5.7.3 Disadvantages of Java

Java has following disadvantages for embedded programming as follows:

1. As Java codes are first interpreted by the JVM, it runs comparatively slowly. This disadvantage can be overcome as follows: Java byte codes can be converted to native machine codes for fast running using just-in-time (JIT) compilation. A Java accelerator (co-processor) can be used in the system for fast code-run.
2. Java byte codes that are generated need a larger memory. An embedded Java system may need a minimum of 512 kB ROM and 512 kB RAM because of the need to first install JVM and run the application.

5.7.4 J2ME

Use of J2ME (Java 2 Micro Edition) or Java Card or Embedded Java helps in reducing the code size to 8 kB for the usual applications like smart card. How? The following are the methods.

1. Use core classes only. Classes for basic run-time environment form the VM internal format and only the programmer's new Java classes are not in internal format.
2. Provide for configuring the run-time environment. Examples of configuring are *deleting the exception handling classes*, *user-defined class loaders*, *file classes*, *AWT classes*, *synchronized threads*, *thread groups*, *multi-dimensional arrays* and *long and floating data types*. Other configuring examples are adding the specific classes—datagrams, input, output and streams for connections to network when needed.
3. Create one object at a time when running the multiple threads.
4. Reuse the objects instead of using a larger number of objects.
5. Use scalar types only as long as feasible.

JavaCard, EmbeddedJava and J2ME are three versions of Java that generate a reduced code size. J2ME provides the optimized run-time environment. Instead of the use of packages, J2ME provides for the codes for the core classes only. These codes are stored at the ROM of the embedded system. It provides for two alternative configurations, connected device configuration (CDC) and connected limited device configurations (CLDC). CDC inherits a few classes from packages for net, security, io, reflect, security.cert, text, text.resources, util, jar and zip. CLDC does not provide for the applets, awt, beans, math, net, rmi, security and sql and text packages in java.lang. There is a separate javax.mircoedition.io package in CLDC configuration. A PDA (personal digital assistant) or mobile phone uses CDC or CLDC.

There is scaleable OS feature in J2ME. There is new virtual machine, KVM as an alternative to JVM. When using the KVM, the system needs a 64 kB instead of 512 kB run-time environment. KVM features are as follows:

1. Use of following data types is optional. (a) Multi-dimensional arrays, (b) long 64-bit integer and (c) floating points.
2. Errors are handled by the program classes, which inherit only a few needed error-handling classes from the java I/O package for the exceptions.
3. Use of a separate set of APIs (application program interfaces) instead of JINI. JINI is portable. But in the embedded system, the ROM has the application already ported and the user does not change it.
4. There is no verification of the classes. KVM presumes the classes as already validated.
5. There is no object finalization. The garbage collector does not have to perform time-consuming changes in the object for finalization.
6. The class loader is not available to the user program. The KVM provides the loader.
7. Thread groups are not available.
8. There is no use of java.lang.reflection. Thus, there are no interfaces that do the object serialization, debugging and profiling.

J2ME need not be restricted to configure the JVM to limit the classes. The configuration can be augmented by profiler classes. For example, MIDP (mobile information device profiler) is a profiler class for mobile devices. A profile defines the support of Java to a device family. The profiler is a layer between the application and the configuration. For example, MIDP is between CLDC and application. Between the device and configuration, there is an OS, which is specific to the device needs.

A mobile information device has the following.

1. A touch screen or keypad.
2. A minimum of 96 × 54 pixel colour or monochrome display.
3. Wireless networking.
4. A minimum of 32 kB RAM, 8 kB EEPROM or flash for data and 128 kB ROM.
5. MIDP used as in PDAs, mobile phones and pagers.

MIDP classes describe the displaying text. It describes the network connectivity. For example, HTTP (Internet Hyper Text Transfer Protocol). It provides support for small databases stored in EEPROM or flash memory. It schedules the applications and supports the timers.

An RMI (remote method invocation) profiler is an exemplary profiler for use in distributed environments.

5.7.5 JavaCard and Embedded Java

A smart card (Section 1.10.4) is an electronic circuit with a memory and CPU or a synthesized VLSI circuit. It is packed like an ATM card. For smart cards, there is Java card technology. (Refer to http:// www.java.sun.com/ products/ javacard.) Internal formats for the run-time environments are available mainly for the few classes in Java card technology. Only one applet can run and each applet is stateless. Java classes for connections, datagrams, input, output and streams, security and cryptography provide the environment.

There is restricted runtime environment. A smart card simple application uses a JavaCard. The Java advantage of platform independence in byte codes is an asset. The smart card connects to a remote server. The card stores the user account past balance and user details for the remote server information in an encrypted format. It deciphers and communicates to the server the user needs after identifying and certifying the user. The intensive codes for the complex application run at the server.

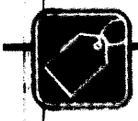
For EmbeddedJava, refer to http://www.sun.java.com/embeddedjava. It provides an embedded run-time environment and a closed exclusive system. Every method, class and run-time library is optional.

Java objects bind the state and behaviour and are instances of a Java class. EmbeddedJava is a Java version, which makes large program development simpler by providing complete OOP features in Java. JVM is configured to minimize memory needs and run-time overheads in the system. Embedded system programmers use Java in a large number of readily available classes for the IO stream, network and security. Java programs possess the ability to run under restricted permissions. JavaCard is a technology for the smart cards and is based on Java.

Summary

- Programming in the assembly language gives the important benefits of precise control of the processors, internal devices and complete use of processor-specific features in its instruction set and addressing modes.

- Program in a high-level language gives the important benefits of short development cycle for a complex system and portability to system hardware modifications. It easily makes larger program development feasible.
- C language support to in-line assembly (fragments of codes in assembly) gives the benefits of both.
- The C program uses various instruction elements, preprocessor directives, macro and constants, including of the source files and header files and functions. Basic C programming elements are the data types, data structures, modifiers, conditional statements and loops, function calls, multiple functions, function queues and service routine queues.
- Infinite looping is a greatly used feature in embedded systems, as it keeps a task or system ready for execution whenever passed a message or signalled to run.
- The C function arguments pass the variable values as well as pass reference to the functions, pointers, NULL pointers and function pointers.
- Queue is an important data structure used in a program. The queue data structure-related functions are 'constructing' a queue, 'inserting' an element into it, deleting an element from it and 'destruction' of the queue. A queue is a FIFO data structure. Queues of bytes play a vital role in a network communication or client server communication also.
- Queuing of pointers to the function on interrupts and later on calling the functions from this queue is a better approach (programming model) as it provides the use of short execution time ISRs.
- Use of 'stack' is very frequent for saving the data in case of interrupts or function calls. Stack-related functions are: 'constructing' a stack, 'pushing' an element into it, popping an element from it and 'destruction' of stack.
- The 'list' and priority-wise 'ordered list'-related functions are 'constructing' a list, 'inserting' an element into it, finding an element from it, deleting an element from it and 'destruction' of the list. One exemplary application is a list of real-time clock interrupts-driven software timers. Another is the list of ready tasks for scheduling the multiple tasks.
- C++ provides all the advantages of C as wall as OOP. Its code size can be reduced by optimizing the generated codes as follows: (a) Declaring private as many classes as possible. (b) Using *char*, *int* and *boolean* (scalar data types) in place of objects (reference data types) as arguments and use local variables as much as feasible. (c) Recovering memory once already used by changing reference to an object to NULL. (d) Selectively de-configuring certain C++ features to get less run-time overhead and less run-time library use. (e) Selectively removing the features of template, run-time type identification, multiple inheritances, exceptional handling, virtual base classes, IO streams and foundation classes.
- Java provides the benefits of extensive class libraries availability, modularity, robustness, portability and platform independence. J2ME is Java 2 Micro Edition, which configures and profiles for the small devices. Java Card and Embedded Java is used in smart card and small embedded devives.



Keywords and their Definitions

Class : A named set of codes that has a number of members – variables, functions, etc. so that the objects can be created from it. The operations are done on the objects by passing the messages to the objects in OOP. Each class defines a logical group with identity, state and behaviour specifications.

Class libraries : Classes for a number of applications like exception, encryption, security, may be provided after thorough debugging and testing for using these in the requirements. Use of class libraries speeds up program development cycle.

Data structure : A multi-element structure that can be referenced by a common name (identity).

Data type : Type of data for a variable, for example, an integer, and on which only a defined set of operations can be performed.

Development cycle : A cycle of coding, testing and debugging. A number of cycles may be needed before finalizing the source codes for porting in the embedded system ROM.

Exception handling : A way of calling the functions to handle on development of an exceptional condition. For example, buffer unable to store any further byte. A programmer thinks of the exceptional conditions and provides for the functions and their calling on occurrence (throwing) of the *exception*.

Foundation classes : Classes meant for GUIs (e.g., button, checkbox, menu and so on.)

Function queue : A queue of pointers for the functions awaiting execution later.

Header file : File containing codes (mostly standard functions) for the user. For example, a file 'math.h' containing codes for the mathematical functions.

High-level language : Programming language in which it is easier to write codes than in the assembly language, and which also gives the important benefits of short development cycle for a complex system, OOP, data types and many features such as portability to system hardware modifications.

In-line assembly : A fragment of codes in assembly language in a high-level language that gives the benefits of processor-specific instructions and addressing modes.

Include file : File that is included along with the user source code before the compilation by the compiler.

Infinite loop : A loop from the program that cannot exit except on interrupt or on a change in certain parameters used by it or on requiring certain parameter or message or token at certain instruction in the loop.

IO stream : A memory buffer created by sending the bytes or characters from a source to a destination so that the destination acts as a sink and accepts them in the sequence that they are sent. An IO stream object does the writing to a file or printer or to a queue or to a network device.

List : A data structure into which elements can be sequentially inserted and retrieved, not necessarily in FIFO or LIFO mode. Each element has a pointer also which points to the address of the next element at the list. Last element points to NULL. A top (head) pointer points to its first element.

Local variable : A variable defined within a function which no other function can use or modify.

Memory optimization : Certain steps changed to reduce the need for memory and having a compact code. It reduces the total size of the memory needed. It may also reduces the total number of CPU cycles, and thus, the total energy requirements.

Modularity : A set of codes are said to be modular if they are usable in multiple applications.

Multiple inheritance : A daughter (derived class) inheriting the member functions from more than one class.

NULL : When a pointer points to NULL, it means that there is no reference to the memory. A memory occupied by an element or object or data structure can be freed by pointing it to the NULL.

Object-oriented programming : A programming method in which instead of operations on data types and structures, variables and functions as individuals, the operations are done on the objects. A class creates the objects in C++ and Java.

Ordered list : A priority-wise ordered list in which it is easy to delete operations (read and then set the pointer to next). It is done sequentially, starting from top.

Passing the reference : From a function, an address of the argument value is passed from the called function when transferring processing to another calling function. The called

function argument becomes modified when operated and the function may get a different argument value back after return from the calling function. The argument value does not save on the stack when that passes by its reference.

Passing the value : From a function, a value is transferred to another function but the same value is reassigned to the original function after return from the called function. Before passing, the argument values saves on the stack and retrieves back on return from the function.

Platform independence : A code that can port on different machines and OSs.

Portable : A code that can be ported in another program by suitable configuration changes.

Preprocessor directives : Program statements and directives for the compiler before the main function to include files and define global variable, global macro (section of code), new data type and global constants.

Private : A variable belonging to a specific class and not usable outside that class.

Queue : A data structure into which elements can be sequentially inserted and deleted in FIFO mode. It needs two pointers, one for the queue tail (back) for insertion and the other for queue head (front) for deletion (read and point to next element).

Reference data types : Array and strings are examples of reference data type.

Robustness : A program is said to be robust if it can function without errors like stack overflow and out of memory errors. Avoiding pointer manipulation instructions, frequently freeing the memory if not needed later and using exceptions, make a code robust.

Run-time library : A library function that links dynamically at the run time. Run-time links increase run-time overheads and out of memory errors can arise.

Run-time overhead : Use of RAM for data and stack is called run-time overhead.

Scalar data types : The character, integer, unsigned integer, floating point numbers, long and double are called scalar data type. Unlike an array, data consist of one single element.

Stack : A data structure in which elements can be pushed for saving in certain memory blocks and can be popped in LIFO mode. It needs one pointer for the stack head (top) for popping (read and point to next element) as well pushing.

Source code engineering tool : A power tool to engineer source codes and also to help in debugging and performance analysis of the codes in high-level languages.

Template : A set of classes using which new classes are built.

Virtual base classes : A special type of class provided in C++.

Review Questions

1. What are the criteria by which an appropriate programming language is chosen for embedded software of a given system?
2. What is the most important feature in C that makes it a popular high-level language for an embedded system?
3. What is the most important feature in Java that makes it a highly useful high-level language for an embedded system in many network-related applications?
4. What is the advantage of polymorphism, when programming using C++?
5. Why do you break a program into header files, configuration files, modules and functions?
6. Design a table to give the features of top-down design and bottom-up design of a program.

7. Explain the importance of the following declarations: static, volatile and interrupt in embedded C.
8. How and when are the following used in a C program? (a) # define (b) typedef (c) null pointer (d) passing the reference (e) recursive function.
9. What are the advantages of using freeware, GNU C/C++ compiler?
10. Why do you need a cross-compiler?
11. Why do you use infinite loop in embedded system software?
12. What are the advantages of re-entrant functions in embedded system software?
13. What are the advantages of using multiple function calls in cyclic order in the main?
14. What are the advantages of building ISR queues?
15. What are the advantages of having short ISRs that build the function queues for processing at a later time?
16. How are the queues used for a network?

Practice Exercises

17. Why do the features in C++ make the code lengthy when using template, multiple inheritance (deriving a class from many parents), exceptional handling, virtual base classes and IO streams? Tabulate the reasons.
18. Write a device driver for a COM serial line port in C including in-line assembly codes.
19. What are the most commonly used preprocessor directives? Give four example of each.
20. How does the use of a macro differ from a function? Explain with exemplary codes.
21. Write program C codes for a loop for summing 10 integers with odd indices only. Each integer is 32 bits. Now unroll the loop and write C codes afresh. Compare the code length in both cases.
22. A set of images in a video frame are to be processed. Which data structure will be best suited for storing the inputs before compressing in an appropriate format?
23. How does combining two functions reduce the memory requirement? Explain with four examples.
24. Consider the format of PPP (point-to-point protocol). Write a C program to transmit PPP data frames encapsulating 4096 data bits. Bits are to be transmitted in a sequence of 32-bit integers stored in memory as in big-endian format.
25. Give two programming examples each in an embedded software, which employs data structures: (a) array (b) queue (c) stack (d) list (e) ordered list (f) binary tree.

Program Modeling Concepts

6

1. Two models for programming languages are procedure and object oriented programming (OOP).
2. Procedure-oriented language examples are ALP and C. The C language provides for functions and *main* C function defines the first function that executes and the other functions are called from the *main*. A function can call another function. There can be nesting of function-calls. There can also be multiple function calls within a function. (Example 5.12, Section 5.4.7).
3. Programming elements in C are preprocessor directives, modifiers, conditional statements and loops, pointers, function calls, multiple functions, function pointers, function queues and ISRs. Program uses data of various types and with various structures: arrays, queues, stacks, lists and trees.
4. OOP language examples are C++ and Java. C++ support object-oriented as well as procedure-oriented codes. Java is purely object-oriented. Object is a reusable software unit and using these units the reusable software components are built. Large complex software can be easily built using the software components.

LEARNING OBJECTIVES

A standard design practice adopted by engineers is to use a model when solutions to problems are to be found. *The events polling based programming, concurrent processes programming, sequential programming and OOP are the programming models most often used. The objective of this chapter is to learn the important concepts of program modeling. The following concepts of program modeling are explained.*

1. *Data flow model using data flow graph and control data flow graph.*
2. *State machine model.*

A powerful modeling language is UML based on objects oriented design. UML fulfills the need for a unified language, which can model many types of processes, classes, objects, activities, designs and development process approaches. It should also be understood. An objective of this chapter is to learn the following.

1. *UML basic elements.*
2. *UML diagram.*

Example of a use of UML is modeling a software implementation.

Embedded systems may be considered in concurrent processing model as systems with concurrently running processes and the processes may require real-time constraints. Concurrent process model and interprocess communication will be described in Chapter 7.

6.1 PROGRAM MODELS

1. *Polling for events model*: Section 5.4.5 explained this model by Example 5.9 and Figure 5.4. There is polling in cyclic loop for the events, state variables, messages, and signals using the switch-case statements.
2. *Sequential program model:* Example 5.12, Section 5.4.7 gave an example of sequential programming model in which there are sequential multiple function calls within a function. Section 5.4.9 gave another example of sequential program model in which the ISRs provided short period deviations from the sequence for executing short codes and sent function pointers as messages inserted into the queue and then the functions executed in FIFO order.

Example 6.1

Figure 6.1 shows a sequential program model for ACVM (Section 1.5.2). The following functions run in sequence.

1. Run function get_user_input ( ) for obtaining input for the choice of chocolate from the child.
2. Run function read_coins ( ) for reading the coins inserted into the ACVM for the cost of chocolate.
3. Run function deliver_chocolate ( ) for delivering the chocolate.
4. Run function display_thanks ( ) for displaying 'Collect the nice chocolate. Visit again!'

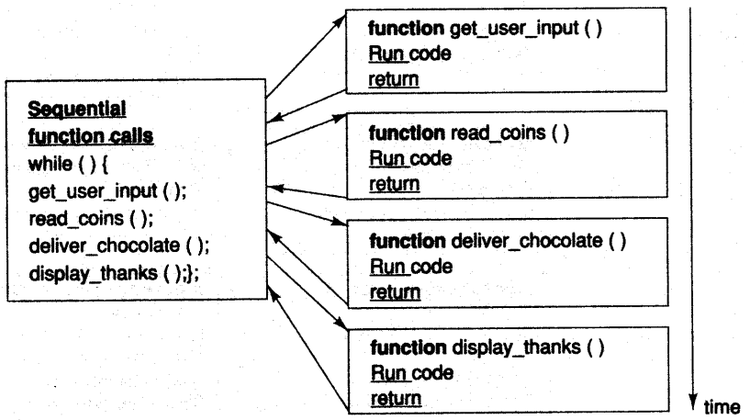


Fig. 6.1 Sequential programming model of an ACVM

3. *Data flow model:* Data flow graphs, abbreviated as DFGs and control data flow graphs, abbreviated as CDFGs are used for modeling the data paths and program flows of software. A program is modeled as handling the input data streams and creating output data streams. The models based on data flow model concept will be described in Section 6.2.
4. *State machine model:* A programming model is that there are different states and the model considers a system as a machine, which is producing the states. Example 5.9 considered different states, which have different displayed menus and the program action depended on the state. Program sequentially polled for the screen state and menu choice selected by the user. Example 3.6 showed how a key marked 5 can produce on pressing different states (0, 5), (1, 5), (1, j), ... The transition of a key occurs if it is pressed again within an interval. The state of the key undergoes in a cyclic fashion as: (1, 5) → (1, j) → (1, k) → (1, l) → (l, 5) → (1, j). The models based on state machine concept will be described in Section 6.3.
5. *Concurrent processes and interprocess communication model:* A programming model is that there are several concurrent tasks (or processes or threads) and each task has the sequential codes in infinite loop. A task sends a message or signal for another task. A task, which gets a message or signal, runs and the remaining tasks remain in the blocked state. Example 5.8 gave the exemplary codes. Example 6.2 gives the concurrent process model based program for the sequential program model in Example 6.1. The model of concurrent processes, tasks or threads and interprocess communication between the concurrent processes will be described in detail in Chapter 7.

Example 6.2

Figure 6.2 shows a program model based on concurrent running of the processes in ACVM (Section 1.10.2). Assume that the program consists of following processes, which can run concurrently.

1. Process `get_user_input ( )` for obtaining input for the choice of chocolate from the child and signalling to process read_coins start.
2. Process `read_coins ( )` wait for signal `get_user_input ( )` and start reading on signal from for reading the coins inserted in the ACVM for the cost of chocolate. Post a signal to process deliver chocolate to start and also post a signal to process `display_wait ( )` to start.

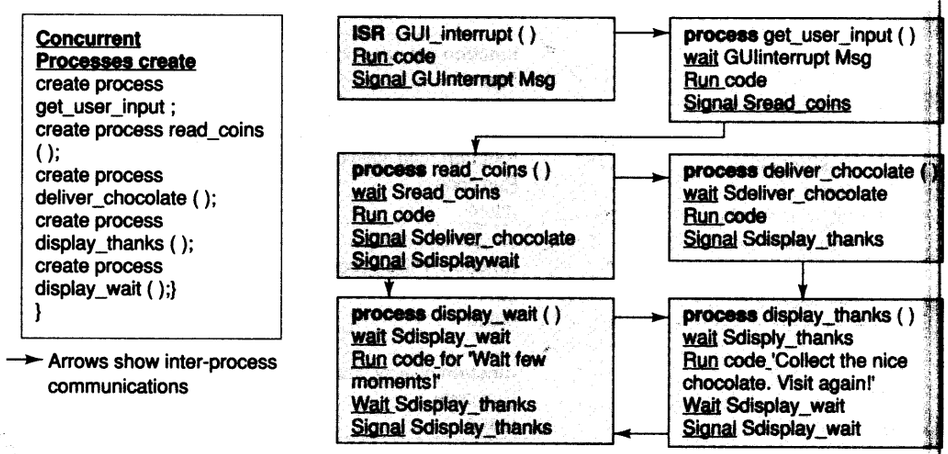


Fig. 6.2 Concurrent processing program model of ACVM

3. Process `deliver_chocolate ( )` wait for signal from `read_coins ( )` and starts delivering the chocolate and post a signal to `display_thanks ( )` to start.
4. Process `display_wait ( )` waits for signal from `read_coins ( )` and starts displaying 'Wait few moments!' and then wait for signal for `display_thanks ( )`.
5. Process `display_thanks ( )` waits for signal from `deliver_chocolate ( )` and from `display_wait ( )` and starts displaying 'Collect the nice chocolate. Visit again!'
6. *OOP model:* Object-oriented language is used for the following features:
   (a) When there is a need for reusability of the defined object or set of objects that are common within a program or between many applications; when there is a need for abstraction and when, by defining objects by inheritance and polymorphs, new objects can be created. There is data encapsulation within an object.
   (b) An object is characterized by its identity (a reference to it that holds its state and behaviour), by its state (its data, property, fields and attributes) and by its behaviour (operations, method or methods that can manipulate the state of the object).
   (c) Defining the logically related group makes a class. Class defines the state and behaviour. It has internal user-level fields for its state and behaviour. It defines the methods of processing the fields.
   (d) Objects are created from the instances of a class. A class can thus create many objects by copying the group and making it functional. Each object is functional. Each object can interact with other objects to process the states as per the defined behaviour. A set of classes and their objects then gives application-program.

Example 6.3

This example gives an object-based model instead of the ACVM sequential program model and concurrent processes-based model given in Examples 6.1 and 6.2, respectively. Figure 6.3 shows classes and objects,

and inheritance and interface features in a program model based on the ACVM (Section 1.10.2). The following can be the classes and objects.

1. `Class GUI` for graphic-user interaction. It has two methods `display_menu ( )` and `get_user_input ( )` and for obtaining input for the choice of chocolate from the child. It has method `set_choice ( )` to set the choice selected.
2. `Class Read Coins ( )` for reading the coins inserted. It has a method read ( ), to read one, two and five rupee coins from three ports and a method sum ( ) for summing the total coins.
3. `Class Deliver_chocolate`. It has methods, `get_choice ( )` to get the choice and `deliver ( )` for delivering the chocolate.
4. `Class MsgDisplay`. It has methods `display_wait ( )` and `display_thanks ( )` for displaying wait and thank messages.

`Class GUI` is used to create GUI objects as multiple instances of GUI. `Class MsgDisplay` is used to create message display objects as multiple instance of wait and thanks messages. `Class MsgDisplay` can be interfaced to an interface `screen_size ( )`, which has an abstract method `screen_size ( )`. The abstract method `screen_size ( )` is implemented in class MsgDisplay

Extending `class MsgDisplay` can specify a new `class MsgTime_Display`. Extended `class MsgTime_Display` inherits all attributes and methods of `class MsgDisplay`. Extended class have another method `display_time_date ( )` for displaying time and date also with each message. Extended class can interface to interface `set_display_period`. `MsgTime_Display` will now implement the method `set_display_period ( )` to set display period of 1 or 2 minutes for thanks and wait messages.

In the objected-oriented approach, there is reusability of defined objects from GUI and a set of objects that are common within a program or between the many applications are created. Also we have abstract methods, `screen_size ( )`. and `set_display_period` which are defined in the interfaces but implemented in the interfacing classes. There is inheritance in the new objects, which are created by extending the class `MsgDisplay`. There is encapsulation of methods and attributes in the class and objects.

UML is modeling language based on the object-oriented model. Section 6.5 will describe the UML.

6.2 DFG MODELS

6.2.1 Data Flow Graph

A data flow means that a program flow and all program execution steps are determined specifically only by the data. The software designer predetermines the data inputs and designs the programming steps to generate the data output. For example, a program for finding an *average* of the grades in various subjects will have the data inputs of the grades and data output of the *average*. The program executes a function to generate the appropriate output. The DFG model is appropriate to model the program for the average.

How does data flow in a program? Data that is input after the operations in the program becomes data that is output after a data flow. A diagram called the DFG represents this graphically. A DFG does not have any conditions within it so that the program has one data entry point and one data output point. There is only one independent path for the program flow when the program is executed.

A circle represents each process in DFG. An arrow directed towards the circle represents the data input (or set of inputs) and an arrow originating from the circle represents a data output (or a set of outputs). Data input

along an input edge is considered as token. An input edge has at least one token. The circle represents the node. The node is said to be fired by the tokens from all input edges. The output is considered by the outgoing tokens, which are produced by the node on firing.

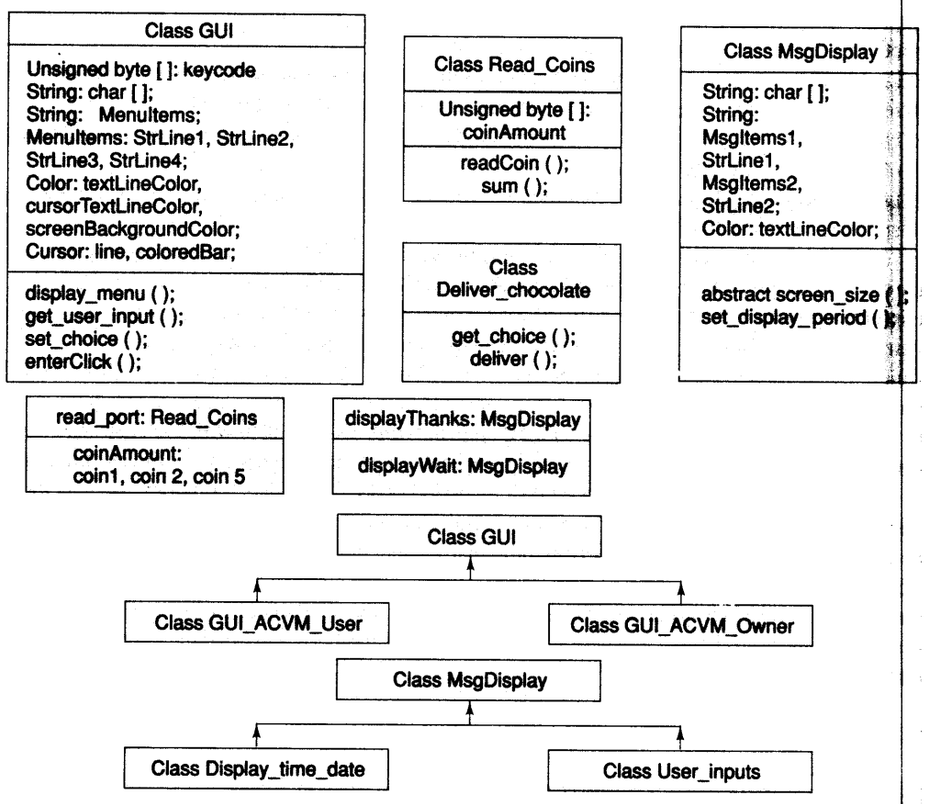


Fig. 6.3 Classes, objects, inheritance and interface features in OOP model based ACVM program

When there is only one set of values of each of the inputs and only one set of values of the outputs for the given input, a DFG is also known as the ADFG, (acrylic data flow graph). All inputs are instantaneously available in APDFG. Examples of non-acrylic data input are as follows: (i) an event; (ii) a status flag setting in a device and (iii) input as per output condition of the previous process.

Example 6.4 gives a DFG during a DSP algorithm.

Example 6.4

Figure 6.4 shows a DFG of the following expression for an output sequence y_6 of a 'finite impulse response (FIR) filter'. An n-th filtered output sequence, $y_n=\Sigma\ (a_i.x_{n-i})$ where the sum is made for i = 0, 1, 2, ..., N–1.] Figure 6.4(a) shows the DFG for a process for the sixth FIR sequence and Figure 6.4(b) shows the DFG for a set of processes of the same sequence. Following are the points notable for the process of calculating $y_6 = a_0.x_6 + a_1.x_5 + a_2.x_4 + a_3.x_3 + a_4.x_2 + a_5.x_1 + a_6.x_6$.

1. There is one input point to the process represented by the circle for calculating y_6.
2. There is one output point for y_6.
3. There is only one memory address and variable for each coefficient and each filter input. There is only one value of each of the six inputs for x and there is only one value of each of the coefficients, a. (DFG is therefore also the ADFG.)

The order in which inputs are obtained and the summation is done is also immaterial.

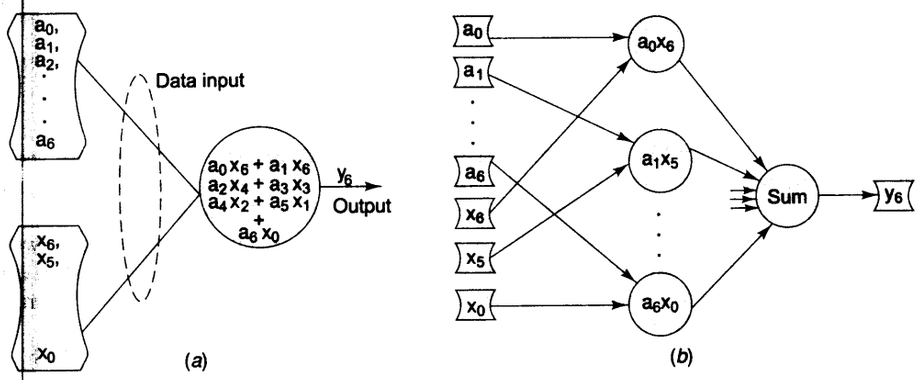


Fig. 6.4 (*a*) Data flow graph (DFG) for a process for the sixth finite impulse response (FIR) sequence (*b*) DFG for a set of processes of the same sequence for an FIR filter with 6 inputs and 6 coefficients

It must be noted from Example 6.4 that there is no complexity in the process for y_6. DFG models help in a simple code design. A simple code design can be defined as that in which the program mostly breaks into DFGs. A DFG models a fundamental program element having an independent path. It gives that unit of a system, which has no control conditions and thus a single path for the program flow. A unit gives the program context and helps in analysing a program in terms of complexity. A more complex program would have a lower number of DFG processes than a simple program.

Figure 6.5 shows a DFG model for the program for saving a picture in a digital camera.

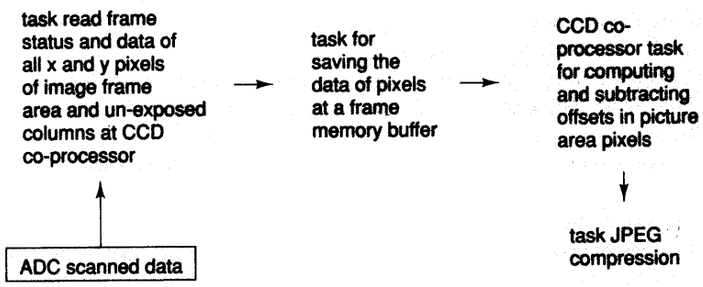


Fig. 6.5 DFG model for program for saving a picture in a digital camera

DFG model program translates and executes as a single-process sequential model program. A program executes as per the input (message or event or set of events) and the input determines the output.

Software implementation becomes greatly simplified when using the DFGs because in the DFG model, there is a single data-in point and a single data-out point, with a process or set of processes that are represented by circle(s). Programming tasks are simplified by representing the code for each process by a circle, using the data input from incoming arrow(s) and generating data output along outgoing arrow(s). When the assignment to an input is fixed in a DFG, it is also called ADFG. Programming complexity is minimized by modeling a program in terms of as many DFGs as possible and the use of as many ADFGs as possible.

6.2.2 Control DFG Model

A control flow means that specifically only the program determines all program execution steps and the flow of a program. The software designer programs and predetermines these steps. How does one design a process that incorporates controls for taking decisions during the data operations and data flow into a program? A process may have the statements that control the inputs or outputs. It may have loops or condition statements in-between (recall Section 5.4.5). Data that is input generate the data output after a control data flow as per the controlling conditions. Output(s) depends on the control statements for various decisions in a process. A CDFG is a diagram, which graphically represents the conditions and the program flow along a condition-dependent path.

The CDFG diagram also represents the effect of events among the processes and shows which processes are activated on each specific event. Here, a variable value changing above a limit or below a limit or falling within a range is also like an event that activates a certain process.

A circle also represents each process (called node) in a CDFG. A directed arrow towards the circle represents the data input (or set of inputs) and a directed arrow from the circle represents a data output (or a set of outputs). A box (square or rectangle with its diagonal axes horizontal and vertical) may represent a condition, for example in Figure 6.6(a). Alternatively, a condition can be marked (or denoted) at the start of the directed arc or arrow. A directed arrow from the box or a marked starting condition determines the action to be taken when the condition is true.

Example 6.5

Figure 6.6(a) shows the controlling input (decision) nodes by the test condition specifying boxes, and the data inputs to a CDFG for an FIR filter with 10 inputs and 10 coefficients [recall Example 6.4 for meanings of various terms in the n-th filtered output sequence, $y_n = \Sigma(a_i.x_{n-i})$; where the sum is made for i = 0, 1, 2, ..., 9]. Following are the points notable for the process of calculating y_n. There is one input point to the process represented by the circle for calculating y_n.

1. There is one output point for y_6. There is only one memory address and variable for each coefficient and each filter input. These are the variables, i n, s and e, which take multiple values during the program flow.
2. The order in which inputs are obtained and summation is done does matter.

Figure 6.6(b) shows the controlling input in the In_A_Out_B program of Examples 4.1 and 4.2. Here, instead of boxes, the condition is marked at the start of the arc.

There is increased programcomplexity in the process for y_n. The CDFG model helps in understanding all conditions and in determining the number of paths a program may take. It also shows us that the software must be tested for each path starting from a decision node, and helps in analyzing the program in terms of complexity.

CDFG model program translates and executes as a concurrent process model program. A controlled decision as per the message or event or set of events determine, which process to execute at an instance.

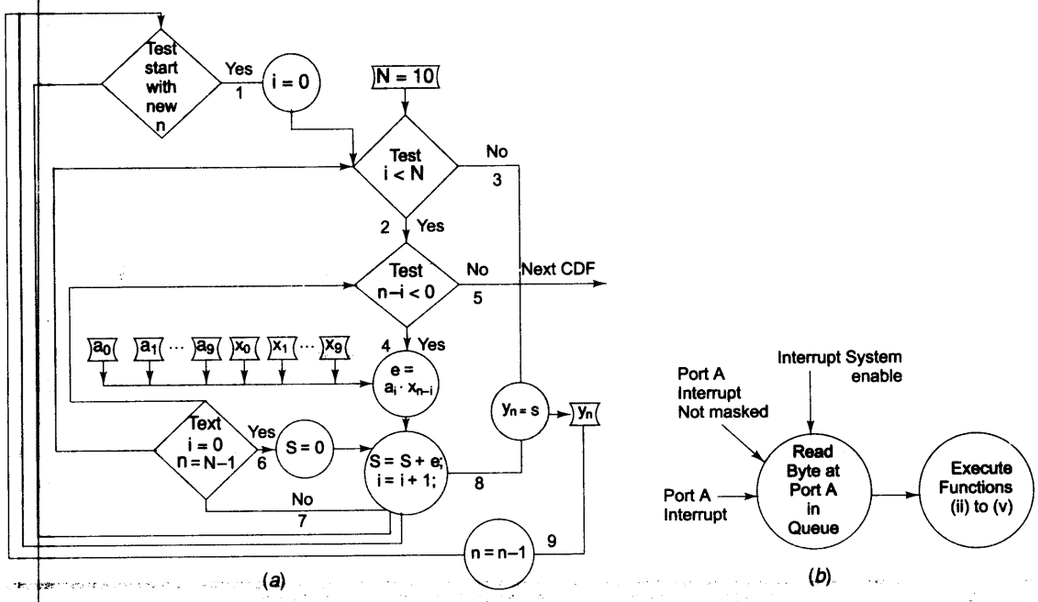


Fig. 6.6 (*a*) Data inputs and controlling input (decision) nodes shown by test boxes in a Control data flow graph for a finite impulse response filter with 10 inputs and 10 coefficients (*b*) Controlling input conditions marked in the In_A_Out_B programs in Examples 4.1 and 4.2 (instead of a box, the condition is marked at the start of an arc)

Software implementation becomes simplified when using the specifications of the conditions and decision nodes in the CDFGs that represent the controlled decision at the nodes, and the program paths (DFGs) that are traversed consequently from the nodes after the decisions.

6.2.3 Synchronous Data Flow Graph (SDFG) Model

When there are number of tokens (inputs) required for a computation to generate more tokens (outputs) in a single firing, the data flow is said to be synchronous. The SDFG model is as follows. [Refer E. A. Lee and D. G. Messerschmitt, 'Static scheduling of synchronous data flow', *IEEE Transactions on Computers*, Feb. 1987.] Let an arc represent a buffer in physical memory. The arc can contain one or more initial tokens with the delays. A token does not fire the computations at a vertex till it is received at the vertex. Vertices (circles) in this graph are called the actors. Actors do the computations. An actor also represents a complete DFG within itself. An edge between the vertices (arcs with an arrow for the direction) represents a queue of output values from one vertex and a queue of input values to another vertex. Edges carry the values from one actor to another.

Let X and Y be two sets of instructions that once fired (started), do not need any further inputs from any source during the computations. Let X generate the output values (tokens/data) a, b and c. Let Y get the input values (tokens/data), a, c, i and j and let i have a delay. The number of inputs to Y need not equal

the number of outputs from X. Y gets additional inputs and does not need all the outputs from X. These computations and data are now modelled by a directed DFG that exists from X to Y. The number of outputs and inputs are labelled near the arc origin and arc end. Figure 6.7 shows actors (vertices, which does the computations on firing) and arcs in a directed graph between X and Y. The figure shows the outputs a, b and c and inputs a, c, i and j. The i is with a delay (dot). The dot on an arc represents the initial token(s) in an SDFG model. Then an initial token may also represent a *delay* that is shown by a dot on the edges of the SDFG. If there is more than one initial token the number of initial tokens are mentioned on the dot (Figure 6.7). The i and j are initial tokens for the vertex Y in Figure 6.7 which show that i has a *delay*.

A number of vertices may be present in a system. All computations are static scheduled in SDFG execution at each vertex (firing elements for the computations and creating another set of output tokens). SDFG model program translates into a sequential model program.

An SDFG model is like a DFG, but also models the delays as well as the number of inputs and outputs. The edges directed to the circle can be assumed to have a physical memory buffer and till the buffer has the data, the computations do not fire.

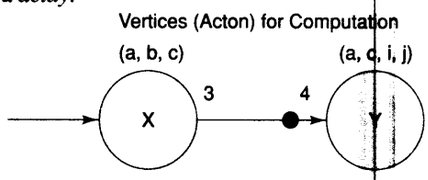


Fig. 6.7 Actors and arcs in a directed graph between X and Y. The outputs a, b and c and inputs a, c, i and j also shown. The i is with a delay (dot)

6.3 STATE MACHINE PROGRAMMING MODELS FOR EVENT-CONTROLLED PROGRAM FLOW

A state machine is a model in which it is assumed that there are states and state transition functions, which produce the states. A state transition function is a function which changes a state to its next state.

Example 6.6

(a) Telephone system *idle*, *receiving a ring*, *dialing*, *connected* and *exchanging messages*. There are five finite number of states.

(b) Consider states of a timer in running state. Figure 6.8 shows the states of timer by circles and state transition by arrows. The count input is the clock input. The changed count value is the output. The output function is the increment in the count value. The state transition function is the time-out on overflow when a predetermined numbers of count inputs are reached. A timer has four finite states: 'idle', 'start', 'running' and 'finished'.
   1. 'Idle' state starts state transition on loading an input, *numTicks* (number of ticks after which the timer finishes).
   2. 'Running' state: on each clock input for decrement, the count value decrements.
   3. 'Finish' state: program flows to the finished state. This is when the count value reaches 0.

(c) A task has four finite states–*idle, ready, running* and *finished* [Figure 6.9(a)]. For output from one state, which becomes the input to the next state, tokens (inputs) from the scheduler are *ready flag* and *block flag*. The tokens (outputs) to the scheduler are *running flag, blocked flag* and *finish flag*.

1. *An 'idle state'* to the *'ready state'* transition occurs when the RTOS schedules this task by sending a token (message) to it. Output from this state consists of saving the scheduler context onto the scheduler stack.
2. The *'running state'* has instructions being executed and the PC continuously changes as per the program flow.
3. Program flows to the *'blocked state'* when the scheduler pre-empts a task. It sends a token (message) to the task. Output from this state consists of saving of the task context at the task stack.
4. Program flow to the *'running state'* again occurs on a token from the scheduler and after retrieving the values from the stack.
5. The flow to the *'finish state'* happens when the instruction reaches the end stage. The output is a message to the scheduler.
6. The flow to the *'ready state'* instead of *'finished state'* occurs when the tasks are in an infinite waiting loop.
7. The flow to the *'idle state'* occurs when a message to the scheduler is sent by the task and the task is deleted from the ready list.

When is a system modeled as the states and state machine? Frequently, there are inputs to a program that change the state of the systems to a new state, and generate outputs, which may also be the inputs for the next state. Now it can be assumed that in a model the running of the program and its flow can be considered as running of a machine generating the states. The program flow can be modeled simply by interstate transitions (from one state to another) from next state transition-functions (Moore model) or next output transition-functions (Mealy model).

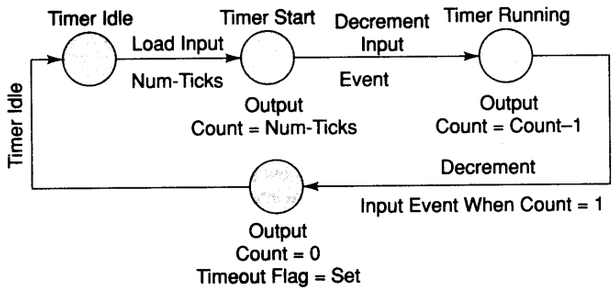


Fig. 6.8 States of a timer using finite states machine model

Following subsections describes finite state machine model.

6.3.1 Finite States Machine (FSM) Model

FSM model states that there is finite number of possible states in a system and a system can only exist in one of these states at an instance. Figure 6.8 showed the *states* modelled as FSM of a timer since there are finite number of timer states. Figure 6.9(a) shows how the states of a *task* can be modelled as a FSM (refer to Section 7.3 for understanding the concept of a task). Figure 6.9(b) shows the FSM states in a program model of an ACVM. There can be transition of the present state to the next state, which depends on the inputs and state transition function. A set of outputs represents a state in the Moore model and a set of outputs represent a state transition in the Mealy model.

Let a circle represent a state and let a directed arc (or an arrow) represent the program flow from a state to another. When modelling a process as FSM, the software designer specifies the following for each state.

1. The state one of the finite number of states.
2. Finite set of inputs (tokens or event flags or status flags) with their values for the state.

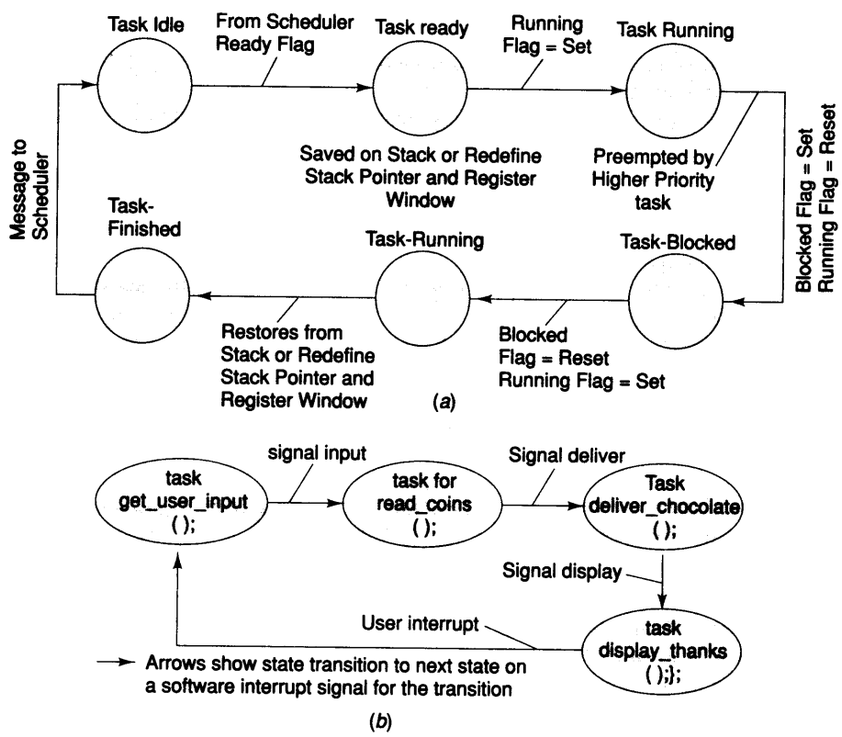


Fig. 6.9 (*a*) States in the finite states machine (FSM) of a task in a multi-tasking program (*b*) FSM states in a program model of an ACVM

3. Finite actions (e.g., computations) during the state and finite set of outputs with their possible values (or tokens or event flags or status flags) and an output (action) *function* for the state that gives the outputs.
4. *State transition function* for each state to take it to the next state.

The steps that model or represent the *states* and *interstate transitions* in FSM data path are as follows.

1. A transition to a new state occurs from the previous state on an *event* (*input*). The event may be setting a value of a certain parameter or the result of the execution of certain codes. A transition may be also be interrupt flag-driven (after a flag sets) or semaphore-driven or interrupt source servicing need-drive.
2. A state can receive multiple tokens (inputs, messages, flags interrupts or semaphores) from another state(s). A *token* (*event*) is used here as a general term that means either an *input* or *event input*. An event input characteristic is that it is asynchronous (one never knows when an event may happen). An event input may happen when there is setting or resetting of a flag. It may occur when there is: (i) a semaphore given or taken or (ii) some indication for a resource or signal or data item generated or (iii) completion of execution of a set of codes. (Refer Sections 7.7, 7.10 and 7.11 for meanings of *semaphores* and *signal*)
3. A state can generate multiple tokens (outputs, messages, flag interrupts or semaphores). An output or set of outputs and variables identifies the next state on mapping the inputs, variables and previous states using the output state transition (action) function (Mealy model). A flag indicating the state

condition or a set of codes being executed or a set of values of certain parameters identifies the next state on mapping the inputs, variables and previous states using the next state transition function (Moore model).

When the FSM model is represented graphically with circles and directed arcs, it becomes complex in the case of a complex process with a large number of states. FSM state table can then be helpful.

6.3.2 FSM State Table

To design a software using the FSM model, a *state table* can be designed for representation of every state in its rows. The following columns are made for each row.

1. *Present state* name or identification.
2. *Action(s)* at the state until some event(s).
3. The *events* (*tokens*) that cause the execution of the state transition function.
4. *Output(s)* from the state output function(s).
5. Next *State*.
6. *Expected time interval* for finishing the transitions to a new state after the event.

The coding using each row can now be easily done as follows.

```
while (presentState) {action ( ); if (event = .....; token = .... )
     {output = ......; stateTransitionFunction ( ); };}
or
Switch (State)
Case presentState: action ( ); if (event = .....; token = .... )
     {output = ......; stateTransitionFunction ( ); };}'
```

Here *presentState* is a *boolean* variable, which is true as long as the present state continues and turns false on transition to the next. The *action* ( ) is a function that executes at the state. If certain events occur and tokens are received (e.g., clock input in a timer), a state transition function, *stateTransitionFunction*, is executed which also makes *presentState* equals false and transition occurs to the next state by setting *nextState* (a *boolean* variable) equals true.

Example 6.7

Figure 6.10 shows the states, state transitions, events, outputs from state output function and finite number of state transitions of a mobile phone key '5' of T9 keypad (Example 3.6). A mobile phone T9 keypad's key marked 5 has five states. It undergoes transitions from initial state (0, 5) as follows: (0, 5) → (1, 5) → (1, j) → (1, k) → (1, l) → (l, 5) → (1, j). First state, variable $s_0 = 0$ represents initial key inactive state. When it is 1, it represents the active state. Second-state variable $s_1 = 5$ or j or k or l represents the final character, which is accepted as output after 1-second delay. A transition occurs when pressed again within a period less than 1 second. The timer has count register, compare register and two flags—TR (timer running) and TF (time compare output) flags. State transition occurs when key input interrupt flag KF is equal to set or TF = 1. The transition on TF resets the key. The timer starts on key interrupt and when timer is in ON state, timer-run flag TR equals 1. When the timer is off or timer time outs, TR = 0. When count = x, timer flag TF equals false before timeout. Timeout occurs when count *x* equals compare register loaded for 1 second. After 1 second, the TF equals 1. There are 12 finite number of states of the key and the timer together. Examples 6.8 and 6.9. give the state table and state machine program.

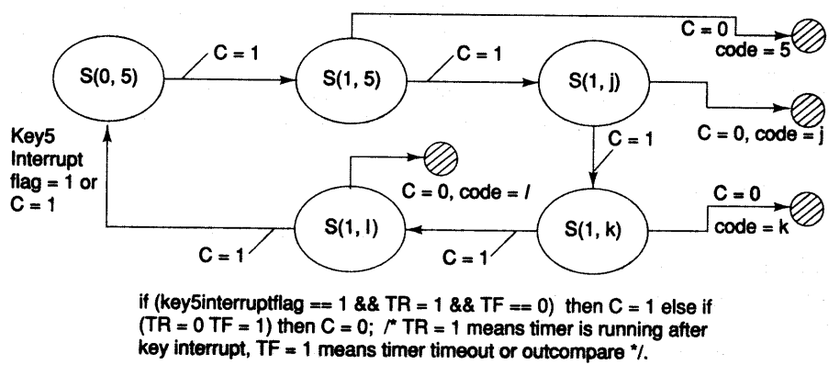


Fig. 6.10 The states, state transitions and finite number of state transitions in a key '5' in mobile phone T9 keypad

Example 6.8

Make state table for FSM in Example 6.7. Table 6.1 gives the state table for the key '5' in T9 keypad.

Table 6.1 State Table for the Key '5' in T9 Keypad

Present State				*Action*	*Events*		*Next State*				*Output*
Key	*TR*	*TF*	*KF*		*KF*	*Count*	*Key*	*TR*	*TF*	*KF*	
(0, 5)	0	0	0	Wait	1	x′	(1, 5)	1	0	0	Timer start
(1, 5)	1	0	0	Wait	1	x	(1, j)	1	0	0	Timer restart
(1, j)	1	0	0	Wait	1	x	(1, k)	1	0	0	Timer restart
(1, k)	1	0	0	Wait	1	x	(1, l)	1	0	0	Timer restart
(1, l)	1	0	0	Wait	1	x	(1, 5)	1	0	0	Timer restart
(1, 5)	0	1	0	–	0	0	(1, 5)	0	0	0	Timer reset
(1, j)	0	1	0	–	0	0	(1, j)	0	0	0	Timer reset
(1, k)	0	1	0	–	0	0	(1, k)	0	0	0	Timer reset
(1, l)	0	1	0	–	0	0	(1, l)	0	0	0	Timer reset

Note: Count x′ initial count register value, = x means counts greater than x′ and less than a compare register value set for 1 second time out. TR = 0 means timer stopped. TR =1 means timer running after loading a value in compare register for capture time out after 1 second. TF = 1 means timer compare time out. KF = 1 key press event. KF = 0 means key reset or inactive.

Example 6.9

The C codes from Table 6.1 can be written as follows.

```
# define true 1
# define false 0
```

```
# define initialState '05000'
# define state1 '15100'
# define state2 '1j100'
# define state3 '1k100'
# define state4 '1l100'
# define state5 '15010'
# define state6 '1j010'
# define state7 '1k010'
# define state8 '1l010'
# define state9 '15000'
# define state10 '1j000'
# define state11 '1k000'
# define state12 '1l000'
 vcid Key5FSM ( ) {
 char [ ] state;
 initialState = "05000"
 while (true) { /* An infinite loop */
 /*-------------------------------------------------------------------------------------- */
   /* function display ("x") shows character x on the screen and function cursor_next ( ) moves the
 cursor position to next when keyiing in an SMS text message. SWI is software interrupt instruction */
 Switch (State) {
 /*************************************************************************************/
 initialState: if ((KF == 1) && Count == 0)) {
                SWI timerstart; /* Execute Interrupt routine to start the timer */
                display ("5"); State = State1;}
      break;
 /*************************************************************************************/
 State1: if ((KF == 1) && Count == x)) {
                SWI timerRestart; /* Execute Interrupt routine to restart the timer */
                display ("j"); State = State2;}
      break;
 /*************************************************************************************/
 State2: if ((KF == 1) && Count == x)) {
                SWI timerRestart; /* Execute Interrupt routine to restart the timer */
                display ("k"); State = State3;}
      break;
 /*************************************************************************************/
 State3: if ((KF == 1) && Count == x)) {
                SWI timerRestart; /* Execute Interrupt routine to restart the timer */
                display ("k"); State = State4;}
      break;
 /*************************************************************************************/
 State4: if ((KF == 0) && Count == x)) {
```

```
                SWI timerRestart; /* Execute Interrupt routine to restart the timer */
                display ("l"); State = State5;}
        break;
/*****************************************************************************/
State5: if ((KF == 0) && Count == 0)) {
                SWI timerReset; /* Execute Interrupt routine to reset and stop the timer */
                display ("5"); cursor_next ( ); State = State9;}
        exit ( );
/*****************************************************************************/
State6: if ((KF == 0) && Count == 0)) {
SWI timerReset; /* Execute Interrupt routine to reset and stop the timer */
 display ("j"); cursor_next ( ); State = State10;}
        exit ( );

/*****************************************************************************/
State7: if ((KF == 0) && Count == 0)) {
                SWI timerReset; /* Execute Interrupt routine to reset and stop the timer */
                display ("k"); cursor_next ( ); State = State11;}
        exit ( );
/*****************************************************************************/
State8: if ((KF == 0) && Count == 0)) {
                SWI timerReset; /* Execute Interrupt routine to reset and stop the timer */
                display ("l"); cursor_next ( ); State = State12;}
        exit ( );

/*****************************************************************************/
}
/*-------------------------------End of Switch-case ------------------------------*/
        } /* End of While infinite loop */
} /* End of Key5FSM */
```

FSM model assumes the finite number of states and reduces the programming tasks to the following: (i) coding for each state transition function and each output function; (ii) knowing the time period taken by the process at each state transition function and between each state, when programming for real time. The FSM model is appropriate for one process at a time, for the sequential flows from one state to the next state and for the controlled flow of the program. When using the FSM model, a state table representation becomes very handy while coding for state machine.

6.4 MODELING OF MULTIPROCESSOR SYSTEMS

6.4.1 Multiprocessor Systems

A large complex program can be partitioned into the tasks or sets of instructions (or processes or threads) and the ISRs. The tasks and ISRs can run concurrently on different processors and by some mechanism the tasks can communicate with each other.

Example 6.10

1. Assume a large program has four tasks: task 1, task 2, task 3 and task 4. It has four ISRs: ISR_A, ISR_B, ISR_C and ISR_D. Assume a processor PA is statically scheduled to run task 2, task 4, ISR_B and ISR_D. Processor PB is statically scheduled to run task 1, task 3, ISR_A and ISR_C. Figure 6.11(a) shows the scheduling on the processors.
2. Assume a large program has four tasks: task 1, task 2, task 3 and task 4. It has 4 ISRs: ISR_A, ISR_B, ISR_C and ISR_D. Assume a processor has dual core with one core statically scheduled to run the tasks and the other the ISRs. ISRs send the messages to the tasks running on the other core. Figure 6.11(b) shows the scheduling on a dual-core processor.

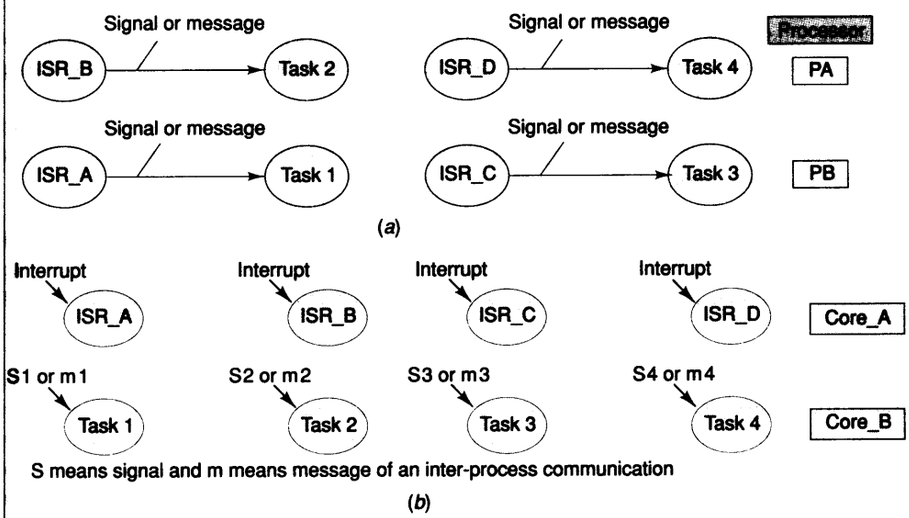


Fig. 6.11 (*a*) Static scheduling of tasks and interrupt service routines on two processors (*b*) Static scheduling on two processor cores

The problem is how to partition the program into tasks or sets of instructions between the various processors, and then how to schedule the instructions and data over the available processor times and resources so that there is optimum performance. Should there be static scheduling for running one task on one processor? Then, suppose one processor finishes computations earlier than the other. What is the performance cost? Performance cost is more if there is idle time left from the available. What is the performance cost if one task needs to send a message to another and the other waits (blocks) till the message is received? Following are the problems in modeling the processing of instructions in a multiprocessor system.

1. Partitioning of processes, instruction sets and instruction(s).
2. Concurrent processing of *processes* on each processor.
3. Static scheduling by the compiler, analogous to scheduling in a superscalar processor. (Each superscalar processor has multiple processing units in parallel.)
4. When superscalar units are present in a processor, it means two or more pipelines of instructions are executed in parallel. Pipeline has a number of stages (3 to 9) and different instructions are at different

stages. Problem is then not only of scheduling of concurrent processing instructions on different processors but also scheduling of concurrent processing instructions on each superscalar unit and pipeline in the processor.

5. Hardware scheduling issue, for example, whether static scheduling of hardware (processors and memories) is feasible or not (it is simpler and its use depends on the types of instructions when it does not affect the system performance).
6. Static scheduling issue (e.g., when the performance is not affected and when the processing actions are predictable and synchronous).
7. Synchronizing issues, synchronization means the use of interprocessor or process communications (IPCs) such that there is a definite order (precedence) in which the computations are fired on any processor in a multiprocessor system (IPC is a message or signal to another process or processor so that it can proceed further. Section 7.9 will describe the IPC in detail).
8. Dynamic scheduling issues (e.g., the performance is affected when there are interrupts and when the services to the tasks are asynchronous. It is also relevant when there is pre-emptive scheduling as that is also asynchronous).
9. Scheduling of the instructions, SIMDs (single instruction multiple data), MIMDs (multiple instructions and multiple data) and VLIWs (very large instruction words within each process) and scheduling them for each processor.

There are several methods of scheduling and synchronizing the execution of instructions, SIMDs, MIMDs and VLIWs in the system. In a multiprocessor system, scheduling is done after analysing the scheduling and synchronizing options for the concurrent processing and scheduling of instructions, SIMDs, MIMDs and VLIWs.

Consider two processors, PA and PB, interfaced with the memory in a system. *Case 1*: Processors share the same address space through a common bus, called tight coupling between processors. *Case 2*: Processors have different autonomous address spaces (like in a network) as well as shared data sets and arrays, called loose coupling. Figure 6.12(a) and (b) show both the cases. *Case 3*: Processors share the memories in alternative bus architecture, for example, three-dimensional mesh, ring, torrid or tree in place of a shared bus between the different tightly coupled processors. Processors process concurrently as follows:

1. One way of concurrent processing is to schedule each task so that it is executed on different processors and synchronize the tasks by some interprocessor communication mechanism.
2. The second way is, when an SMID or MIMD or VLIW instruction has different data (e.g., different coefficients in Example 6.5), each task is processed on different processors (tightly coupled processing) for different data. This is analogous to the execution of a VLIW in TMS320C6, a Texas Instruments DSP series processor. It employs two identical sets of four units and a VLIW instruction word can be within 4 and 32 bytes. It has instruction level parallelism when a compiler schedules such that the processors run the different instruction elements at the different units in parallel.
   Note: The compiler does *static scheduling for VLIWs.* Static scheduling is one in which a compiler compiles such that the codes are run on different processors or processing units as per the schedule decided and this schedule remains static during the program run even if a processor waits for others to finish the scheduled processing.
3. An alternate way is that a task instruction is executed on the same processor or different instructions of a task can be done on different processors (loosely coupled). A compiler schedules the various instructions of the tasks among the processors at an instance.

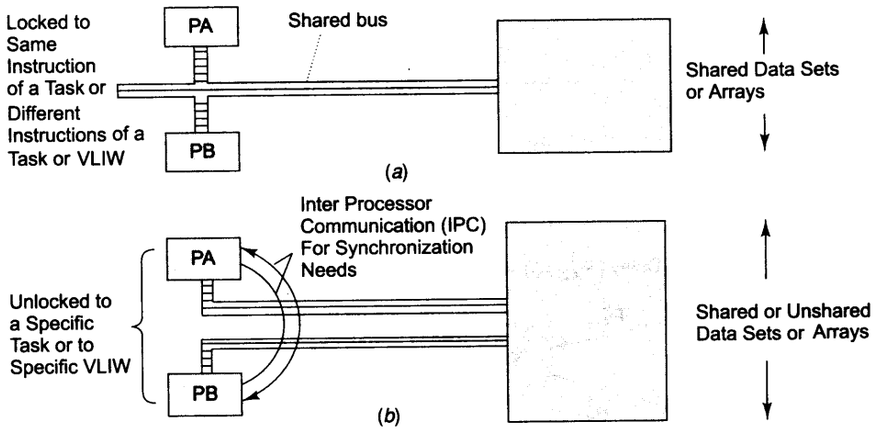


Fig. 6.12 (*a*) Tightly coupled processors sharing the same address space while processing multiple tasks (*b*) Loosely coupled processors having separate autonomous address spaces as in a network as well as shared address space for data sets and arrays

6.4.2 Model of Unfolding SDFGs into Homogeneous SDFGs

An SDFG models the delays as well as the number of inputs and outputs (Section 6.2.3). The edges directed to a circles are assumed to have a physical memory buffer and till the buffer has the data, the computations do not fire. When there is only one token at the input, and one at the output, an SDFG is called homogenous SDFG (HSDFG). Figure 6.13(a) shows a modeling of computations by an SDFG. Figure 6.13(b) shows an HSDFG representation after unfolding the SDFG in Figure 6.13(a). The dot and label over the edge show delayed two number input tokens at vertex Y.

For example, suppose that the outputs from vertex X′ (a set of computations) is a and input to Y′ (another set of computations) is also a. An SDFG can therefore unfold into a HSDFG. An SDF graph can be unfolded into one or more HSDFGs. Two vertices can be connected by two or more edges in the HSDF graph. An HSDF graph will naturally have more vertices and edges than an SDFG because only one token is permitted at a vertex.

When there is an indefinitely long data sequence, SDFG-based modelling and the consequent unfolding into the HSDF graphs helps. For example, HSDFGs applied to the computations of a fast Fourier transform or for coding voice data. An HSDF graph can also effectively model an IPC (interprocessor communication) graph. All computations are static scheduled in HSDFG execution at each vertex (firing elements for the computations and creating another set of output tokens). Let there be a sequence of computations that are fired at the vertices. Let *precedence* in a directed graph define the computations order by which the vertices are placed first, then next, and then next to next. A sequence on one processor among the set of processors can be delayed at the arcs. Input from another processor (initial token) can also be delayed. A SDF model program then translates into a number of parallel concurrent or sequential model programs using HSDFGs.

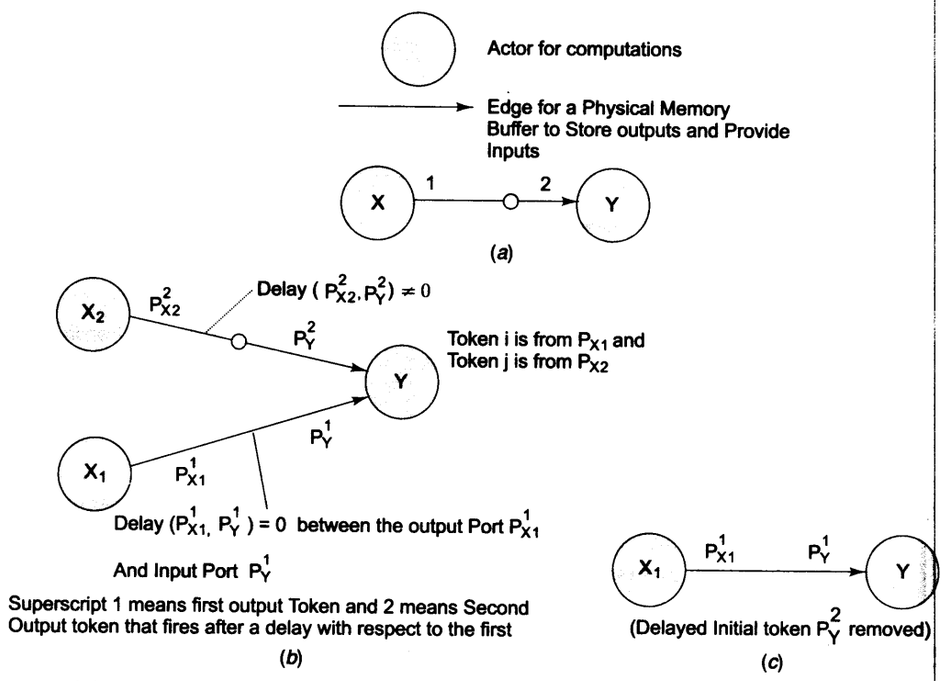


Fig. 6.13 (*a*) A modeling of computations by an SDFG. The dot and label over the edge show delayed two number input tokens at vertex Y (*b*) A homogeneous SDFG representation after unfolding the SDFG (*c*) An APEG representation from an HSDFG after removing the delayed edge

Multiprocessor system computations and their firing instances can be modeled. Modeling simplifies the programming, scheduling and synchronizing of the processes. HSDFG models are like an SDFGs but have the feature that there is only one token that delays along an edge (arc or arrow) because there is only one token at input, and one at output.

6.4.3 Model of Unfolding HSDFGs into APEGs

Acrylic precedence is a precedence of vertices in a directed graph such that there are no delays at the arcs. If initial tokens (delays) are taken off from an HSDF graph, an acrylic precedence expansion graph (APEG) is obtained.

APEGs are important for scheduling in multiprocessor systems. An APEG not only has along the arc, starting inputs identical to the output from a previous vertex, but also no delaying for the token. Hence, the execution is smooth along the arc with no interprocessor communication time. An APEG-based algorithm becomes the simplest to schedule such that the precedence constraints in the algorithm remain the same as before. Figure 6.13(c) shows a corresponding APEG that is a graph with no *delays*. It drives from a HDFG [Figure 6.13(b)] or SDFG [Figure 6.13(a)].

A task-level concurrent process as well as an IPC graph can be modeled using APEGs and HSDFGs. A thread running on one processor modeled as APEG can pass a control to another by blocking itself or by sleeping, but the sequence and process flow along the APEG remain intact. Example 6.11 explains this.

Example 6.11

Let V'_1, V'_2, and V'_3 be the computation vertices assigned to processor PA. Let V''_1, V''_2, V''_3 be the computation vertices assigned to processor PB concurrently processing with PA. An IPC is needed when algorithm (or set of computations) V''_3 cannot proceed till there is a message (token) from V'_2. Let IPC be between V''_3 and V'_2. This synchronizes the processes at PA and PB through the IPC. Figure 6.14 shows one APEG and one HSDFG with an IPC to PB from PA.

APEG models are such that there are no delays during execution at any stage in an APEG or chain of APEGs. Complex problems are therefore first modelled as the SDFGs, then SDFGs are unfolded into HSDFGs and HSDFGs are separated into APEGs. Processing is as per precedence constraints between the APEGs. APEG-based algorithms become the simplest to schedule but precedence constraints in the algorithm among its APEGs remains the same as before.

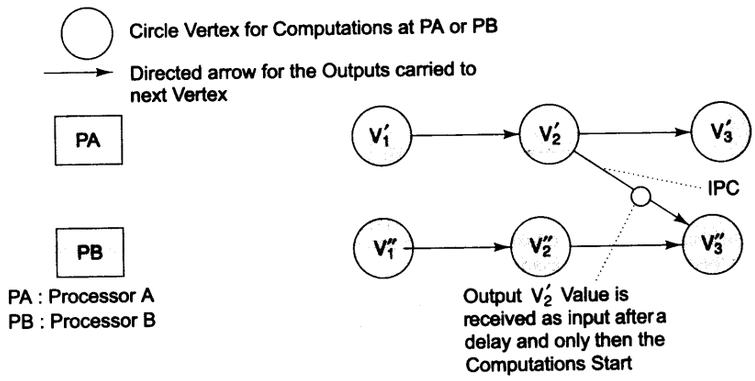


Fig. 6.14 A two-processor system with one acrylic precedence expansion graph and one homogeneous synchronous data flow graph with an IPC to PB from PA

6.4.4 Applications of the Graphs to Multiprocessor Systems: Partitioning and Scheduling

When there are multiple processors in parallel, the partitioning of a program is done as follows.

1. There are minimum number of IPCs so that the total time of IPC delays (waiting periods) minimized.
2. There is load balancing. Each processor has the least waiting time by sharing the processing load.
3. The performance cost minimizes. Performance cost means the execution time required (i) for computations for the tokens and delays at the edge (communication time), (ii) the computation time before firing (computations) by a vertex (transition) and (iii) context switch time.

Consider Figure 6.15 vertices. At each vertex computations occur such that the precedence constraints maintain (remain intact). The graph of a program thus partitions into the functions or tasks or threads. One of the three following strategies can schedule a program for running.

1. Each task or function is executed on an assigned processor. Each task or function is executed on different processors at different periods. Instructions of four different tasks are partitioned on two processors. Instructions of four different tasks are partitioned and scheduled on two processors differently in different periods [Figures 6.15(a) to (d)] show these four partitioning and scheduling strategies].

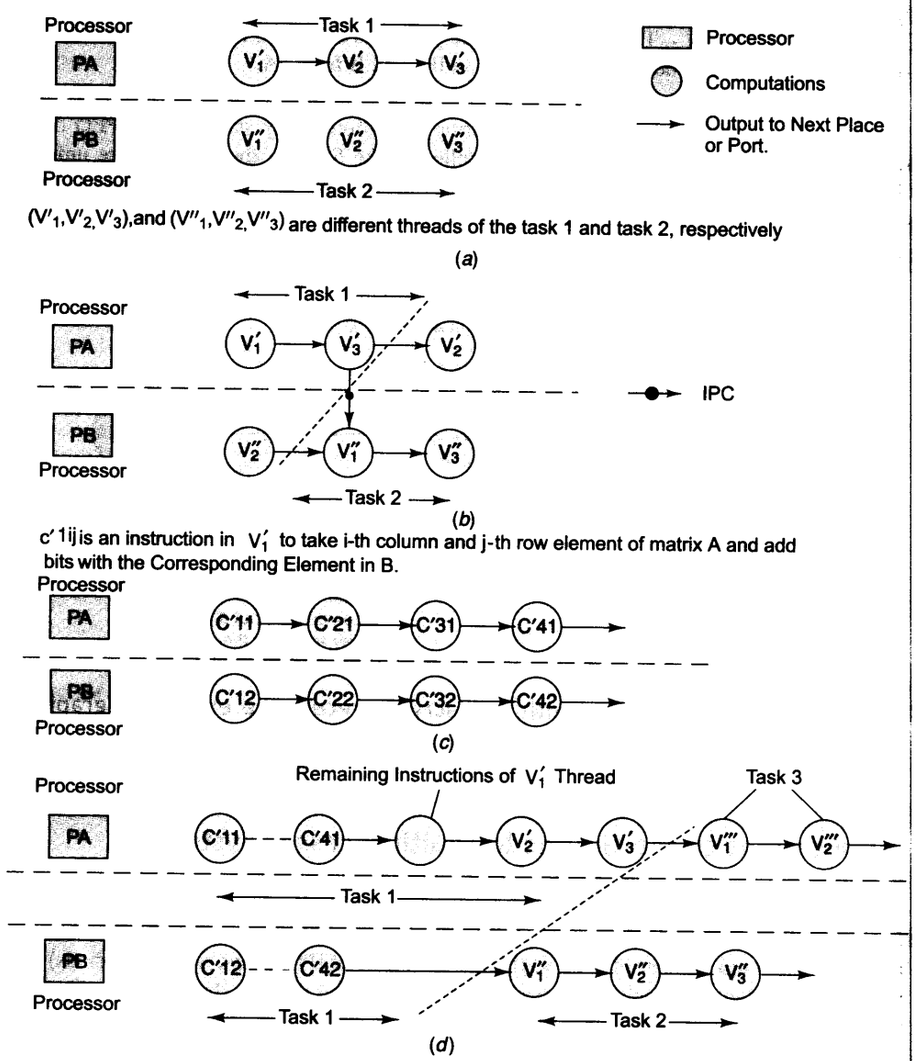


Fig. 6.15 (*a*) Each task or function is executed on an assigned processor (*b*) Each task or function is executed on different processors at different periods (*c*) Instructions of four different tasks are partitioned on two processors (*d*) Instructions of four different tasks are partitioned and scheduled on two processors differently during different periods

2. Each set of data is partitioned in a VLIW instruction and is executed on the different processors, which execute the same program. Consider a matrix addition process. Each row can be added on a different processor when the data of the rows are partitioned among the processors. Such data partitioning is preferred when processing a DSP-VLIW.

A combined partitioning is done both at the data level as well as the task (or function) level. Different functions themselves may run concurrently on different processors but at the micro or atomic-level data is partitioned and the instructions are run.

Partitioning and scheduling of vertices can be done in a number of ways. (i) Each task or function is executed on an assigned processor. (ii) Each task or function is executed on different processors at different periods. (iii) Instructions of four different tasks are partitioned on two processors. (iv) Instructions of four different tasks are partitioned and scheduled on two processors differently in different periods. (v) Data partitioning in case of SIMDs, MIMDs and VLIWs.

6.5 UML MODELLING

Recapitulate Section 5.5. The concepts used in object-oriented language are also used in software designing. Object-oriented designing is also done as before.

1. Object-oriented design is done when there is a need for reusability of the defined software components as object or set of objects (reusable components). The new component can be abstracted from the existing. New components and object designs are created by the object inheritances and polymorphs. There is information encapsulation within a designed component or object.
2. A designed component object is also characterized by its identity (a reference to it that holds its state and behaviour), by its state (its designs for data, property, fields, attributes and algorithms) and by its behaviour (method or methods that can manipulate the state of the design).
3. New object designs are created from the instances of a designed class. Class defines the state, attributes, operations and behaviour of a design concept. It has internal user-level fields for its state and behaviour. It defines the ways of using the designs.
4. A designed class can then create many component objects (designs) by copying the group and making designs functional. Each design is a functional design. Each object design can interface with other designs to process the states as per the defined behaviour.
5. A set of classes then gives the complete software design for a system.

UML is a unified (common) modeling language for any general system for which object-oriented analysis and design are feasible and which can be abstracted by models. Unification in UML means its common applicability to many designs or processes. We can then model the following by a similar set of diagrams: (i) software visualizing, (ii) data design(s), (iii) algorithms design(s), (iv) software design(s), (v) software specifications, (vi) software development process, (vii) an industrial process.

UML is a language for modeling. Details of the language can be learnt from a standard textbook. [For example, "The Unified Modeling Language User Guide" by Grady Booch, James Rambaugh and Ivar Jacobson, Addison Wesley, 1999.] UML features and its applications in designing of embedded systems can be understood from the following brief description.

Figure 6.16(a) to (f) shows representations of six basic UML elements: class, package, stereotype, object, anonymous object and state. Table 6.2 gives a list of these and their description.

A conceptual design modeling can follow the UML approach. A conceptual design can use the user, object, sequence, state, class and activity diagrams. Table 6.3 gives UML 'class', 'state', 'sequence', 'collaboration' and 'object' diagrams.

UML allows the SpecCharts and StateCharts: SpecCharts (specification charts) is another language for specifications and charts. It allows state machines to use sequential programs to model the state actions. StateChart is a language for implementing the activity diagram, FSM states and state transitions, concurrency, synchronization, timing and behavioural hierarchy. The message sequence charts are first prepared and from these the StateChart, to show an activity diagram. For example, StateChart can model two concurrent activities of two FSMs. Its models along with its StateCharts-like features provide implementation of the exception-handling (trapping and interrupting) routines easily. UML sequence diagrams may also use the StateChart substrates, or models created by the StateChart language.

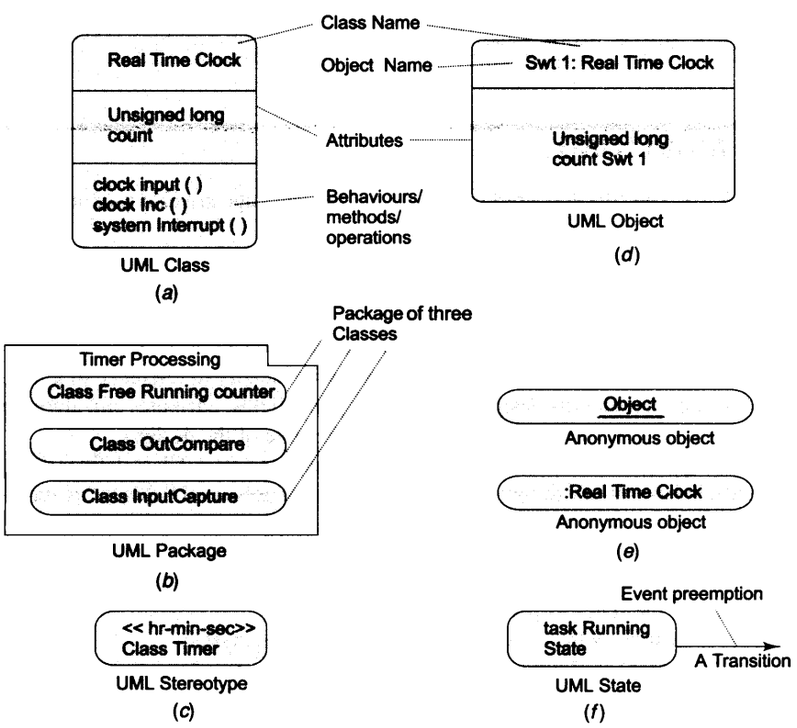


Fig. 6.16 Representation of unified modeling language basic elements (*a*) class (activeClass and abstract or inactive class) (*b*) package (*c*) stereotype (*d*) object (*e*) anonymous object and (*f*) state

UML is a powerful modeling language for: (i) software visualizing, (ii) data design, (iii) algorithms design, (iv) software design, (v) software specifications and (vi) software development process. UML basic elements are class, package, stereotype, object, anonymous objects and state. UML modeling is by class diagrams, state diagrams, object diagrams, sequence diagrams and collaboration diagrams.

Table 6.2 UML Basic Elements

Modeling Diagram	*What does it model and show?*	*Exemplary Diagrammatic Representation*
Class	Class defines the states, attributes and behaviour. A class can also be an active or abstract class.	Rectangular box with divisions as shown in Figure 6.16(a) for class names for its identity, attributes and behaviours (operations or methods or routines or functions).
Abstract class	A class in general may be abstract when either one or more states, operations or behaviour not completely defined, being in an abstract stage, or when it is not for creating objects but only a class, which extends, implements the abstract behaviours (methods) and specifies the abstract attributes (fields or properties) that class can create the object.	Rectangular box with divisions for class names for its identity, attributes and operations, but with prefix abstract with each abstract behaviour and attribute.
Object	An instance of a class that is a functional entity formed by copying the states, attributes and behaviour from a class.	Rectangular box with object identity followed by semicolon and class identity as shown in Figure 6.16(d).
Active object	An active class defines an active object instance of an active class. A process or thread is equivalent to the active object in UML, because active object posts the signals like thread and can wait before starting or resuming the operations using the methods.	Rectangular box with object identity followed by semicolon and class identity, but with prefix active with object identity.
Active class	An active class means a thread class that has a defined state, attributes, behaviours and behaviours for the signals. Active class in addition, defines the control by signal behaviours (for a signalling object, which can be posted and for which it may wait before starting or resuming). Thus there is control on the class behaviour.	Rectangular box with thick border lines and inner divisions for the class names for the identity, attributes and behaviours (operations and signals), but with prefix active with class identity.
Signal	An object, which is sent (posted) from one active class (active object) to another active class, which waits for start or resumption. Signal object behaviour defines the behaviour (operation method) of the interprocess communication. [Signal (Section 4.2.2) is software instruction or method (function), which generates interrupt.] Signal object has attributes (parameters). Attribute may be just a flag of 1-bit.	Signal identity within two pairs of starting and closing signs followed by class identity (Similar to stereotype).
Stereo-type	An unpacked collection of elements (attributes or behaviours) that is repeatedly used.	Rectangular box with stereotype identity name given within the two pairs of starting and closing signs followed by the class identity as shown in Figure 6.16(c).

(*Contd*)

Modelling Diagram	*What does it model and show?*	*Exemplary Diagrammatic Representation*
Anonymous object	An object without identity.	Rectangular box with no object identity before the semicolon and class identity as shown in Figure 6.16(e).
Package	A packed collection of classes and objects.	A rectangular box with inner boxes for each class with name for class-identity. Package name is given over the top of box as shown in Figure 6.16(b).
State	A state.	Rounded rectangle with state name for its identity and with an arrow from the box. The arrow indicates a transition as shown in Figure 6.16(f).

Table 6.3 UML Diagrams

Modeling Diagram	*What does it model and show?*	*Representation*
State diagram	State diagram shows a model of a structure for its start, end, in-between associations through the transitions and shows event labels (or condition) with associated transitions.	A dark circular mark shows the starting point, arrows show the transitions. A label over the arrow shows the condition or event, which fires the transition. A dark rectangular mark within a circle shows the end [Figure 6.17(a)].
Class diagram	Class diagrams show how the classes and objects of a class relate hierarchical associations and object interactions between the classes and objects.	Rectangular boxes show the classes and arrows with unfilled triangles at the end show the class hierarchy. Classes in the hierarchy can be joined using a line. Start and end numbers on a line show the number of objects of a class associates with the number of objects of the other [Figure 6.17(b)].
Object diagram	Object diagram defines the static configuration of the system. It also gives the relationship among the consequent objects.	Refer Figure 6.17(c).
Sequence diagram	Sequence diagram visualizes the interactions between the objects. Sequence diagrams also specify the sequences of states.	Rounded rectangles for states and rectangular boxes with object identity and class connects by arrows. Vertical axis pointing downward shows progressively the time [Figure 6.18(a) and (b)[1]].
Collaboration diagram	Collaboration diagram visualizes the concurrent sequences of states or object interactions.	Horizontal or vertical axis pointing right or downwards shows progressively the time and a parallel set of sequences show concurrency. Conditions or events can be labelled on the arrow [Figure 6.18(c)[2]].

[1] Figure 6.18 shows the UML sequence diagrams. Figure 6.18(a) shows sequence of interaction between the states, (b) shows the sequence diagram (e.g., automatic chocolate-vending machine sequences of states).

[2] Figure 6.18(c) shows the collaboration diagram (concurrent multiprocessing).

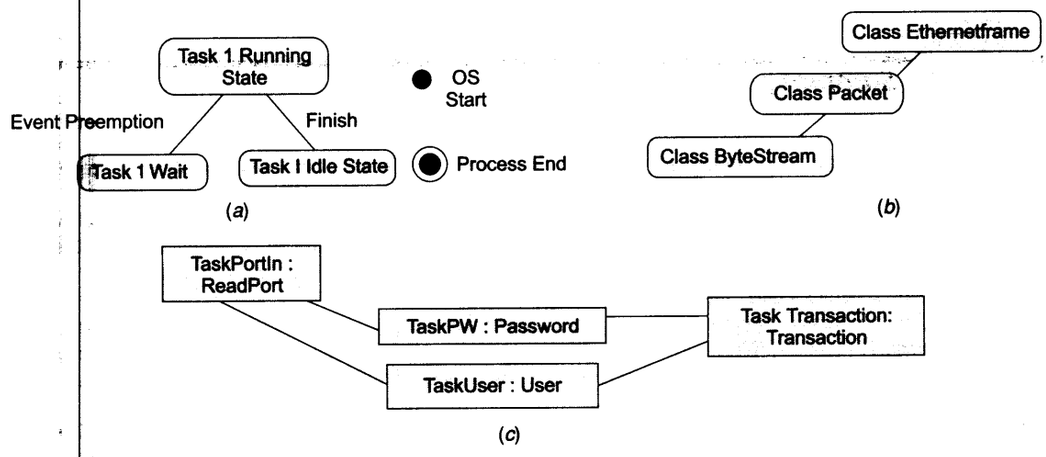


Fig. 6.17 Unified modeling language diagram: (*a*) state diagram (*b*) class diagram (*c*) object diagram

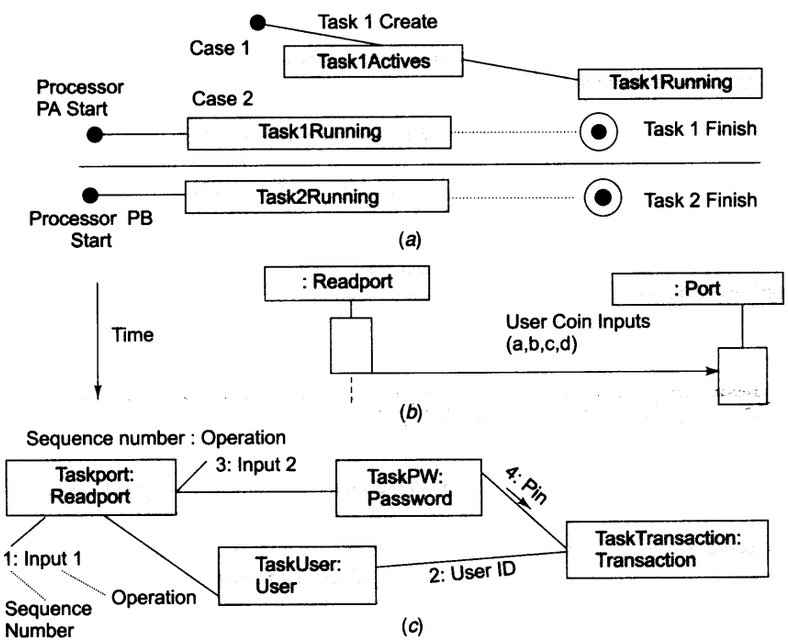
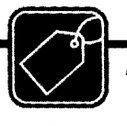


Fig. 6.18 Unified modeling language diagrams: (*a*) sequence of interaction between the states (*b*) sequence diagram (e.g., automatic chocolate-vending machine sequences of states (*c*) collaboration diagram (e.g., multiprocessing concurrently processing system)

Summary

- Important models for programming are multiple function calls, polling for events, function queuing, sequential functions, data flow graph, CDFG state machine, concurrent processing and OOP.
- A standard design practice is using a model or set of models during the development process for software. Software implementation becomes greatly simplified by the use of DFGs because programming tasks are thereby reduced to the following. Coding for each process represented by a circle using the data input from incoming arrow(s) and generating data output to the outgoing arrow(s).
- In a DFG model, there is a single data-in point and a single data-out point with a process or set of processes that are represented by circle(s). When the assignment to an input is fixed in a DFG, it is also called ADFG. Programming complexity is minimized by modeling a program in terms of as many DFGs as possible and the use of as many ADFGs as possible.
- Another important concept of program modeling is the CDFG for program design and analysis. The CDFGs represent the controlled decision at the nodes and program paths (DFGs) that are traversed consequently from the nodes after the decisions.
- Program modeling can be done by the FSM and state machine models.
- The FSM model is appropriate for one process at a time, for the sequential flows from one state to the next state and for a controlled flow of the program. The FSM is found to easily model the states in processes in the systems.
- The multiprocessor system uses two or more processors for faster execution of the following: (i) program functions, (ii) tasks or (iii) single instruction multiple data instructions or (iv) multiple instructions multiple data instructions or (v) very long instruction words. The VLIWs in the DSP instructions can be completed at high speed. Modelling of multiprocessor system uses SDFG and HSDFG representations in which there is unfolding of the SDFG so that there is only one token which delays along its edge and/or an APEG in which there are no delays.
- Models are used for partitioning, load balancing, scheduling, synchronization and resynchronization during the program flow on the multiple processors. This gives minimum total performance costs (processing delays).
- UML is used for modeling the object-oriented programs. UML specifies the following basic elements: class, package, stereotype, object, anonymous objects and state. UML specifies the class diagrams, state diagrams, object diagrams, sequence diagrams and collaboration diagrams.



Keywords and their Definitions

ADFG	:	Acrylic DFG model when the assignment to the input is fixed.
APEG	:	HSDFG with no delays and vertices arranged in the precedence order.
Concurrent processing	:	When several processes execute a set of instructions such that each can proceed further by passing or exchanging messages or signals or tokens.
CDFG	:	Modeling by representing the controlled decision at the nodes and program paths (DFGs) that are traversed consequently from the nodes after the decisions.
DFG	:	The code for each process is represented by a circle and the data inputs to the process are by incoming arrow(s) and the generation of data output is through the outgoing arrow(s).
Finite state machine	:	A model in which there are finite states. After a given set of inputs, a state changes according to the state transition function.

HSDFG	:	Representation in which there is unfolding of the SDFG so that there is only one token, which may delay along its edge.
Interprocess communication	:	During concurrent processing, IPC is a message or signal or token to another waiting process or processor to proceed further.
Load balancing	:	Partitioning and scheduling of threads (set of instructions) and instructions such that each processor shares the processing load in a multiprocessor system.
Model	:	It is a representation by which problem, process, design or analysis can be easily understood and the problem becomes simplified after modelling.
Multiprocessor system	:	A system that uses two or more processors or that uses dual cores or multiple cores for faster execution of the (i) program functions, (ii) tasks or (iii) single instruction multiple data instructions or (iv) multiple instructions multiple data instructions or (v) very long instruction words.
Partitioning	:	Partitioning the graphs into parts, with each part scheduled on the processors as per the scheduling strategy adopted.
Performance cost	:	Time taken in waiting for execution at a vertex or at a sub-graph or micro thread.
Resynchronization	:	Repeating synchronization by suitable mathematical analysis, reducing the number of IPCs and thus the delays caused at the processors waiting for the IPCs in a multiprocessor system.
Scheduling	:	Allocation of different vertices or sub-graphs on different processes.
SDFG	:	A DFG representation in which input(s) delays are also shown. Circles (vertices) are the actors where computations take place. Nodes in a DFG edge (arc or arrow) has dots for the delays and labels the number of inputs and outputs.
State transition function	:	A process or state of codes that carry a program state from one to another.
UML	:	A language used for modelling the (i) software visualizing, (ii) data designs, (iii) algorithms design, (iv) software designs, (v) software specifications and (vi) software development process.
Total performance cost	:	Total of all performance costs. This will be the minimum if the load on the processors is balanced and there is appropriate partitioning and scheduling.

Review Questions

1. Why does program complexity increase with a reduced number of DFGs and increasing decision nodes?
2. Explain with one example each, APEG, SDFG and HSDFG.
3. Why do you unfold SDFGs into as many HSDFGs as feasible and then HSDFGs into as many APEGs as possible?
4. How does concurrent processing help in VLIW instruction execution at high speed?
5. How will you schedule an SIMD instruction on two processors?
6. How will you schedule an MIMD instruction on two processors?
7. How will you schedule an VLIW instruction on two processors?
8. What do you mean by completely dynamic scheduling and completely static scheduling in a multiprocessor system?
9. What do you mean by load balancing? How do you achieve it by combined partitioning?
10. How is an anonymous object denoted in UML?
11. What are the features of UML?

Practice Exercises

12. Tabulate various program models and give two application examples of each.
13. How will the DFG for FIR filter modify as CDFGs?
14. Draw a CDFG to incorporate decision nodes at the loop start and loop end to limit the summation terms up to n = 10 for Equation as follows: $y_n = \Sigma\, a_i.x.\ (n–i) + \Sigma\, b_l.y.\ (n–l)$ used in an IIR filter.
15. Draw an FSM model of an automatic chocolate-vending machine program. The machine permits only one type of coin, Rs 1, one chocolate at a time and one chocolate is cost is Rs 8.
16. Give the program model of master and slave robots in robot orchestra.
17. Give the program model of a digital camera.
18. Draw multiprocessor system for the cases: (i) tightly coupled to the memory; (ii) loosely coupled; (iii) coupled by mesh; (iv) ring-coupled; (v) torrid-coupled and (vi) tree-like coupling.
19. How do you solve the following problems: 'How can a program be partitioned into tasks or sets of instructions between the various processors? How can then the instructions and data be scheduled over the available processor times and resources so that there is an optimum performance'?
20. Assume that four processes are scheduled to run on two processors. A program is partitioned in such a way that with each 10,000 ns each process schedules 10 times on each processor. What will be the minimum number of contexts switching/microsecond?
21. How will you create and display SMS message T9 keypad of a mobile phone? Use the states, FSM model and state tables for all keys 0, 1 to 9 with T9 keypad. Use Examples 6.7 to 6.9 as templates.
22. Draw the 'class diagram', 'state diagram', 'sequence diagram', 'collaboration diagram' and 'object diagram' for AVCM in Section 1.10.2.

Interprocess Communication and Synchronization of Processes, Threads and Tasks

7

The following important points have been discussed in carlier chapters.

1. **Software embedded in a system can be highly complex as an application program has a number of functions, ISRs, threads, multiple physical and virtual device drivers, and several program objects that may be sequentially or concurrently processed on a single processor or multiple processors.**
2. **The system tasks may have vastly different functionalities, priorities, response-time constraints, latencies and deadlines.**

LEARNING OBJECTIVES

We will discuss the following with examples.

1. *Processes or threads or tasks are controlled by an OS, which enables their running concurrently in a system.*
2. *The tasks and their states.*
3. *The tasks and task-control-blocks, thread-stacks and process-control-blocks.*
4. *The context and context switching in multiprocessing, multitasking and multithreading system.*
5. *The distinction between functions, ISRs and tasks in order to understand the finer details of processing of each during a program run.*

We will learn the uses of the following IPCs (inter-process communication functions) and devices.

1. Semaphore to communicate occurrence of an event at one process to another which waits for the event to proceed further.
2. Semaphore as mutex or counting semaphore and understanding of the P and V semaphores.
3. Problem and solution of the data that have to be shared between multiple tasks.
4. Mutex in solving the shared data problem and application in running the critical section codes.
5. Solution of the priority inversion problem and deadlock situation when using a semaphore.
6. *Signal* by a process to force a running process to interrupt and start a signal handler function (ISR) or process.
7. *Queue* in which messages are inserted by a process and communicated to other which are waiting for messages.
8. *Mailbox* to communicate a message from one process to another which waits for the message to proceed further.
9. *Pipe* device to communicate the bytes for messages from one process to another which takes the messages.
10. *Socket* as a bi-direction device to communicate the bytes as a stream or as per the protocol from one socket address in a process to another socket address in another process which can be local or remote.
11. *Remote procedure call* from a process to call a function or method in another process which is remote as per the protocol from one address in a process to another address in another process which can be local or remote.

An OS provides mechanism of the IPCs to enable processes to synchronize and transfer the signals and messages. The OS also provides functions for the process, memory, IOs, device, time and event management. The OS also provides interrupt-handling mechanism. The OS also provides scheduling mechanism for the processes or tasks or threads. Chapter 8 will describe these mechanisms.

Chapters 9 and 10 will describe with exemplary RTOSes. The RTOSes also provide for handling the task-priorities and real-time constraints.

7.1 MULTIPLE PROCESSES IN AN APPLICATION

7.1.1 Process

Application program can be said to consist of a number of processes [Figure 7.1(a)] and each process runs under the control of an OS (Section 1.4.6). Meaning and the basic concept of process can be understood as follows:

1. A process consists of sequentially executable program (codes) and *state*-control by an OS.
2. The *state* during running of a process is represented by the information of process state (created, running, blocked or finished), process structure—its data, objects and resources, and process control block (PCB) [PCB explanation follows later].
3. A process runs on scheduling by OS (kernel), which gives the control of CPU to the process. Process runs instructions and the continuous changes of its state takes place as the PC changes. [PC is program counter or instruction pointer to point to the current instruction of running program.]

Process is that unit of computation, which is controlled by some process at the OS for scheduling that lets it execute on the CPU and by some process at OS for resource management that permits use of system memory and other system resources such as network, file, display or printer.

Process is defined as a computational unit that processes on a CPU and whose state changes under the control of kernel of an OS. It has a state, which at an instance defines by the *process status* (running, blocked or finished), *process structure*—its data, objects and resources and *process control block.*

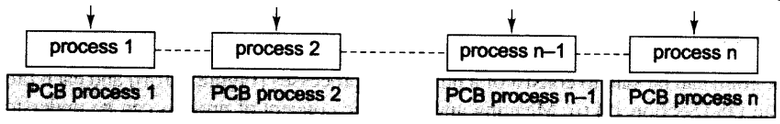
Example 7.1

Consider a mobile phone device (Section 1.5.5). The device-embedded software is highly complex. It has a number of functions, ISRs, threads, multiple physical and virtual device drivers and several program objects that must be concurrently processed on a single processor. The OS assumes the application-embedded software as consisting of a number of processes. Exemplary processes at the device are as follows: (i) *Voice encoding and convoluting process*: the device captures the spoken words through a speaker, and generates the digital signals after analog to digital conversions, and does the digits encoding and convoluting using a CODEC; (ii) *Modulating process*; (iii) *display process*; (iv) GUIs (graphic user interfaces) and (v) *Key-input process* for provisioning of user keypad interrupts.

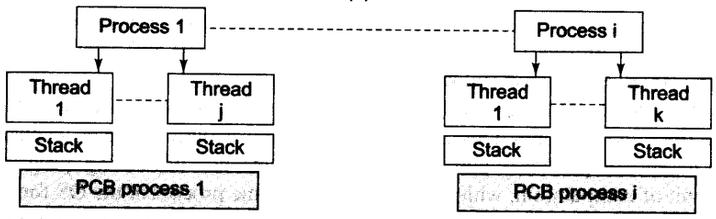
Process Control Block PCB is a data structure having the information using which the OS controls the process state. The PCB stores in the protected memory addresses at kernel. The PCB consists of following information about the process state.

1. Process ID, process priority, parent process (if any), child process (if any) and address to the next process PCB, which will run next.
2. Allocated program memory address blocks in physical memory and in secondary (virtual) memory for the process codes.
3. Allocated process-specific data address blocks.
4. Allocated process heap (data generated during the program run) addresses.
5. Allocated process stack addresses for the functions called during running of the process.
6. Allocated addresses of the CPU register-save memory as a process context represents by CPU registers, which include the PC and SP [These register contents (process context) load into the CPU registers from the memory when the process starts running, and the addresses of CPU register-save memory saves the registers on context switch to another process].

Units of computation, execution of codes in which is controlled by the OS scheduler, inter-process communication, resource-manager, system-memory and other system-resources (such as network, file, display or printer) access control mechanisms and are processed concurrently.

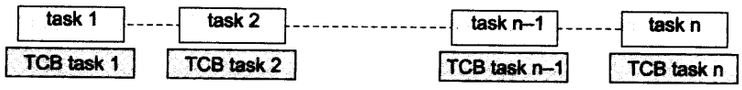


(*a*)



A thread is a process or sub-process within a process that has its own program counter, its own stack pointer, and stack, its own priority-parameter for its scheduling by a thread-scheduler, and its own variables that load into the processor registers on context switching and is processed concurrently along with other threads.

(*b*)



Tasks are embedded program computational units that run on a CPU under the state-control using a task control block. The tasks are processed concurrently

(*c*)

Fig. 7.1 (*a*) Processes (*b*) Threads (*c*) Tasks

7. Process-state signal mask [when mask is set to 0 (active) the process is inhibited from running and when reset to 1, the process is allowed to run].
8. Signals (and messages) dispatch table (for the process IPC functions).
9. OS-allocated resources' descriptors [e.g., file descriptors for open files, device descriptors for open (accessible) devices, device-buffer addresses and status, socket-descriptor for open socket].
10. Security restrictions and permissions.

The present CPU registers, which include PC and SP are called context and save on the PCB-pointed process stack and register-save memory addresses. Then the running process stops. Other process CPU registers now load and that process runs. This also means that the context has switched to another process.

7.2 MULTIPLE THREADS IN AN APPLICATION

Application program can be said to consist of a number of threads or a number of processes and threads [Figure 7.1(b)]. Meaning and basic concept of thread can be understood as follows:

1. A thread consists of sequentially executable program (codes) under state-control by an OS.
2. The state information of a thread is represented by *thread-state* (started, running, blocked or finished), *thread structure*—its data, objects and a subset of the process resources and *thread-stack*.
3. A thread is a lightweight entity.

[*Note:* A process is considered as a heavyweight process and a kernel-level controlled entity. A process can have codes in the secondary memory from which the pages can be swapped into the physical primary memory during running of the process. The process may therefore have process structure with the virtual memory map, file descriptors, user-ID and so on. A thread can be considered a lightweight process and a process-level controlled entity. [*Note:* What the structure is, however, depends on the OS.]

A thread is a process or subprocess within a process that has its own PC, its own SP and stack, its own priority parameter for its scheduling by thread-scheduler, its variables that load into the processor registers on context switching. It has its own signal mask at the kernel. The signal mask when unmasked allows the thread to activate and run. When masked, the thread is put into a queue of pending threads. A thread stack is at a memory address block allocated by the OS. When a function in a thread in OS is called, the calling function state is placed on the stack top. When there is return the calling function takes the state information from the stack top.

A multiprocessing OS runs more than one process. When a process consists of multiple threads, it is called multithreaded process. A thread can be considered as daughter process. A thread defines a minimum unit of a multithreaded process that an OS schedules onto the CPU and allocates the other system resources.

A process structure consists of data for memory mapping, file description and directory. Different threads of a process may share a common process structure. Multiple threads can share the data of the process.

Process can be allocated program memory address blocks in the physical memory as well as in the secondary (virtual) memory for the process codes. Memory mapping means mapping of the running program logic addresses with the physical addresses where the pages of the process codes load. A map called virtual memory map is used for memory mapping. A thread need not possess this data.

How does a task differ from a thread? Thread is a concept used in Java or Unix. A thread can either be a subprocess within a process or a process within an application program. To schedule the multiple processes, there is the concept of forming thread groups and thread libraries. A task is a process and the OS does the multitasking. Task is a kernel-controlled entity while thread is a process-controlled entity. A task is analogous to a thread in most respects. A thread does not call another thread to run. A task also does not directly call another task to run. Both need an appropriate scheduler. Multithreading needs a thread-scheduler. Multitasking needs a task-scheduler. There may or may not be task groups and task libraries in a given OS.

Example 7.2

Consider a mobile phone device (Section 1.5.5). *Display_process* can have multiple threads. A thread *Display_Time_Date* can be for displaying clock time and date. A thread *Display_Battery* can be for displaying battery power. A thread *Display_Signal* can be for displaying signal power for communication with the mobile service provider. A thread *Display_Profile* can be for displaying silent or sound-active mode. A thread *Display_Message* can be for displaying unread message in the inbox. A thread *Display_Call Status* can be for displaying call status; whether dialing or call waiting. *Display_Menu* can be for displaying menu. These threads can share the common memory blocks and resources allocated to the *Display_Process*. A display thread is now the minimum computational unit controlled by the RTOS. Each thread has independent parameters—ID, priority, PC, SP, CPU registers and its present status.

A thread is a process or subprocess within a process that has its own PC, its own SP and stack, its own priority parameter for its scheduling by thread-scheduler. Thread is a concept in Java and Unix and it is a lightweight subprocess or process in an application program. The thread can share a process structure. It has a thread stack at the memory. It has a unique ID. It has states in the system as follows: starting, running, blocked and finished.

7.3 TASKS

Task is the term used for the process in the RTOSes for the embedded systems. (For example, VxWorks and μCOS-II are the RTOSes, which use the term task.) A task is similar to a process or thread in an OS. Some OSes use the term task and some use the term process. Figure 7.1(c) shows the application software consisting of a number of tasks.

1. A task consists of a sequentially executable program (codes) under a state-control by an OS.
2. The state information of a task is represented by the task state (running, blocked or finished), task structure—its data, objects and resources and task control block (TCB).

An application program can also be defined as a program consisting of the tasks and task behaviours in the various states. The task states are controlled by some process at the OS for scheduling that allows it to execute on the CPU and by some process at OS for resource-management that allows it to use the system memory and other system resources such as network, file, display or printer.

Embedded software for an application may consist of a number of tasks and each task run needs a control of the state by OS. Assume that there is only one CPU in a system. Each task is independent in that it takes control of the CPU when scheduled by a scheduler at the OS. The scheduler controls and runs the tasks. A task is an independent process. No task can call another task. [It is unlike a C (or C++) function, which can call another function.] The task can send signal(s) or message(s) that can let another task run waiting for that signal or message. The OS can block a running task and let another task gain access of the CPU to run the servicing codes.

Task is defined as embedded program computational unit that runs on a CPU under the state-control of kernel of an OS. It has a state, which at an instance defines by *status* (running, blocked, or finished), *structure*—its data, objects and resources and control block.

Example 7.3

Consider an ACVM (Section 1.10.2). The ACVM-embedded software is highly complex and the OS schedules to run the application-embedded software as consisting of a number of tasks. Exemplary tasks at the ACVM are as follows: (i) *Task User Keypad Input*: the keypad gets the user input. (ii) *Task Read-Amount* for reading the inserted coins amount. (iii) *Chocolate delivery task*; delivers the chocolate and signals the machine to get ready for the next input of the coins. (iv) *Display Task*. (v) GUI_Task (for graphic user interfaces). (vi) *Communication task* for provisioning the AVCM owner access to the machine status and information.

7.4 TASK STATES

Figure 7.2(a) shows a task and its states. Task has state, which includes its status at a given instance in the system. It can be one of the following state: idle (created), ready, running, blocked and deleted (finished). It is

in the ready state again after finish when it has infinite waiting loop—an important feature in embedded system design. Multitasking operations are by context switching between the various tasks.

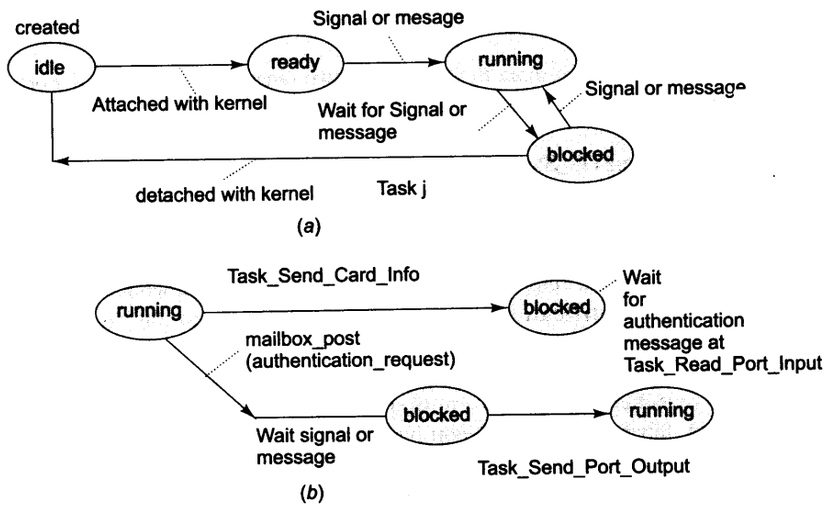


Fig. 7.2 (*a*) Task and its states (*b*) States of the task Task_Send_Card_Info in Example 7.4

A task can be considered to be in one of the five states. What the states can be, however, depends on the ROS. Five states are as follows.

1. *Idle (created) state:* The task has been created and memory allotted to its structure. However, it is not ready and is not schedulable by the kernel.
2. *Ready (active) state:* The created task is ready and is schedulable by the kernel but not running at present as another higher priority task is scheduled to run and has the system resources at this instance.
3. *Running state:* Executing the servicing codes and getting the system resources at this instance. It will run till it needs some IPC (input) or starts wait for an event or till it pre-empts by another higher priority task than this task.
4. *Blocked (waiting) state:* Execution of the servicing codes suspends after saving the needed parameters into its context. It needs some IPC (input) or waiting for an event or waiting for higher priority task to block. For example, a task is pending while it waits for an input from the keyboard or file. The scheduler then puts it in the blocked state.
5. *Deleted (finished) state:* The created task has memory de-allotted to its structure. It frees the memory. Task has to be re-created.

A created and activated task will be in one of the three states, ready, running and blocked.

Example 7.4

Consider a smart card (Section 1.10.3). When it is inserted into a card reader host machine, it gets the radiation and charges up.

Step 1: Let the main program first run an OS function *OS_initiate* ( ). This enables use of the RTOS functions.

Step 2: The main program runs an OS function OS_Task_Create ( ) to create a task, Task_Send_Card_Info. The task is for sending card information to the host. The task is allocated memory for the stack. The Task has a TCB using which the OS controls the task. The task state is idle state. Let this task be of high priority.

Step 3: OS_Task_Create ( ) runs two more times to create two other tasks, Task_Send_Port_Output and Task_Read_Port_Input and both of them are also in idle state. Let these tasks be of middle and low priorities, respectively.

Step 4: The functions for starting OS_Start ( ) and for initiating n system clock interrupts OS_Ticks_Per_Sec ( ) run. The system switches from the user mode to the supervisory mode every 1/60 seconds if n = 60. All three task states will be made in ready_state by an OS function.

Step 5: The OS runs a function, which makes the Task_Send_Card_Info state as running. The Task_Send_Card_Info runs an OS function mailbox_post (authentication_request), which sends the server identification request through IO port to the host using the task Task_Send_Port_Output.

Step 6: The Task_Send_Card_Info runs a function mailbox_wait ( ), which makes the task state as blocked and the OS switches context to another task Task_Send_Port_Output and then to Task_Read_Port_Input for reading IO port input data.

Step 7: When the mailbox gets the authentication message from the host server, the OS switches context to Task_Send_Card_Info and the task comes to the running state again.

Figure 7.2(b) shows the task Task_Send_Card_Info states in different steps.

7.5 TASK AND DATA

Figure 7.3 shows a task and its data including its context and TCB. A task has the following data specific to a task, which saves at the TCB.

1. Each task has an ID just as each function has a name. The ID is of one byte and is called the index of the task if a typical OS assigns each ID a number between 0 and 255.
2. Each task may have a *priority parameter*. The priority, if between 0 and 255, is represented by a byte (usually, the higher the value, the lower the priority of that task).
3. Each task has its independent (distinct from other tasks) values of the following at an instant: (i) PC (memory address from where it runs if granted access to the CPU) and (ii) SP (memory address from where it gets the saved CPU registers and parameters, which includes registers for the task PC and pointer to task stack-top after the scheduler grants access to the CPU). These two values are the part of its context of a task.

Context Each task has a context (CPU registers and parameters, which includes registers for the task PC and pointer to the called function stack-top). This reflects the CPU state just before the OS blocks one task and initiates another task into the running state. The context thus continuously updates during the running of a task, and the context is saved before switching occurs to another task.

Context Switch Only after saving these registers and pointers does the CPU control switch to any other process or task. The context must retrieve on transfer of program control to the CPU back for running the same task again, on the OS unblocking its state and allowing it to enter the running state again. The context-switching action must happen each time the scheduler blocks one task and runs another task.

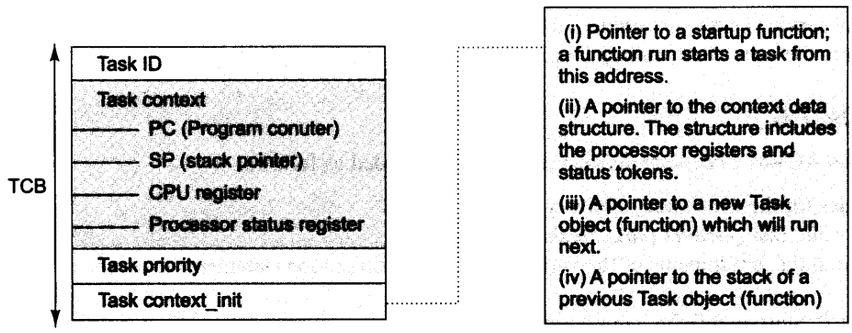


Fig. 7.3 A task and its data including its context and task control block

Each task also has an initial context, *context_init*. The *context_init* has the initial parameters of a task. The parameters of *context_init* are as follows: (i) Pointer to a start-up function: a function run starts a task from this address. (ii) Pointer to the context data structure: the structure includes the processor registers and status tokens. (iii) The task context may also include a pointer to a new task object (function) which will run next. (iv) It may also include a pointer to the stack of a previous task object (function).

Example 7.5

Consider an ACVM (Section 1.5.2). After the *Task Read-Amount* (for reading the inserted coins amount) gets the required cost of the chocolate, it send an IPC (a signal or message) to let the OS context switch and start the *Chocolate delivery task.*

Chocolate delivery task delivers the chocolate, sends an IPC for *Display task* to display 'thank you and visit again' and sends another IPC to the machine ready for the next input of the coins.

There are context switches from *Task Read-Amount* to *Chocolate delivery task*, from *Chocolate delivery task* to *Display task* and from *Display task* to *Task User Keypad Input.*

7.5.1 Task Control Block

Each task has a TCB. TCB is a memory block. Figure 7.3 showed the TCB data for a task. The TCB is a data structure having the information using which the OS controls the task state. The TCB stores in the protected memory area of the kernel. The TCB consists of the following information about the task. It stores the current instant PC information (to indicate the address of the next instruction to be executed for this task), memory map, signal (message) dispatch table, signal mask, task ID, CPU state (registers, task PC and task SP) and kernel stack (for executing system calls and so on). [*Note:* (i) The TCB is similar to the process control block (PCB) and (ii) TCB data structure can vary from one OS to another.]

7.6 CLEAR-CUT DISTINCTION BETWEEN FUNCTIONS, ISRs AND TASKS BY THEIR CHARACTERISTICS

7.6.1 Task Coding in Endless Event-Waiting Loop

Each task may be coded such that it is in endless event-waiting loop to start with. An event loop is one that keeps on waiting for an event to occur. On the start event, the loop starts from the first instruction of the loop.

Execution of service codes (or setting a token that is an event for another task) then occurs. At the end, the task returns to the start event waiting loop.

Example 7.6

Consider an ACVM *Chocolate delivery task*. It can be coded as follows.

```
/* The codes for the Chocolate_ delivery_ task */
    static void Task_Deliver (void *taskPointer) {
    /* The initial assignments of the variables and pre-infinite loop statements that execute once only*/
.
while (1) { /* Start an infinite while-loop. */
/* Wait for an event indicated by an IPC from Task Read-Amount */
.
/* Codes for delivering a chocolate into a bowl. */
.
/* Send message through an IPC for displaying "Collect the nice chocolate. Thank you, visit again" to
    the Display Task*/
/* Resume delayed Task Read-Amount */
    }; /* End of while loop*/
/ * End of the Task_Deliver function */
```

7.6.2 Distinction between Function, ISR and Task

When there are multiple devices, functions, ISRs and program objects, the embedded software can be modelled as consisting of multiple tasks and each task is scheduled by the kernel schedule and uses IPCs for synchronization. Threads are used in embedded Linux- or Unix-based applications. Threads are used in Java. Functions are subunits of the processes or tasks or ISRs or another function. Functions and ISRs do not have analogue of PCB or TCB. They have only a stack. Function has no associated scheduler-like tasks scheduler or thread scheduler at the kernel. ISR has associated interrupt handler at the kernel. Table 7.1 summarizes the characteristics of functions, ISRs and tasks.

1. *Function* is used in any routine for performing a specific set of actions as per the arguments passed to it and which runs when called by a process or task or thread or from another function. Functions run by nesting. Function runs after the previous context saving and after retrieving the context from a common stack.
2. An *ISR* is a function, which executes on interrupts. An ISR executes on an event and pending ISRs run as per priority-based scheduling. ISR can post the events or signals or messages. ISRs run as per the hardware-based interrupt-handling mechanism. ISRs may or may not run by nesting. ISR runs after the context saving and after retrieving the context from a common stack in case of nesting.
3. A *task* is a function, which executes on scheduling. A task can wait as well as post the events or signals or messages. The tasks run after saving of the previous context at the SP pointed address in task TCB and the context switching to new context at the new task SP pointed address in TCB. The tasks run as per the task scheduling and IPC management mechanism of the OS.

Recall that Section 5.4.6 explained the re-entrant function. Each task must be either reentrant function or must have a way to solve the shared data problem. Section 7.7 will explain the shared data problem and use of semaphores.

Table 7.1 Characteristics of the Functions, Interrupt Service Routines (ISRs) and Tasks

	Function	*ISR*	*Task*
1.	***Uses:*** Function is used in any routine or process or task for running specific set of codes for performing a specific set of actions as per the arguments passed to it.	ISR is used for running a specific set of codes for performing a specific set of actions. ISR has code, which runs once and for servicing the interrupt-call only.	Task is used for running specific set of codes for performing a specific set of actions. Task has codes in an endless waiting loop.
2.	***Calling source***: A call to run a function is from another function or process or thread or task.	An interrupt call for running an ISR can be from hardware or software at any instance. All interrupt source calls for running the ISRs are independent.	A call to run the task is from the system (OS). A OS preemptive scheduler can allow another higher priority task to execute after blocking the present one. It is the RTOS (kernel) only that controls the task scheduling.
3.	***Context save***: Each function code run by changes in program counter instantaneous value. There is a stack. On the top of which the program counter value (for the code left without running) and other values (called functions' context) must save when calling another function or on interrupt and start of ISR. When there is return from a function to the function, which called it, the program counter restores from stack top to that value where the code left earlier [Figure 7.4(a)]. The stack of the functions in a process, thread or task is at the common memory block when the different functions execute.	Each ISR is an event-driven function code. The code run by changes in program counter instantaneous value. ISR has a stack for the program counter instantaneous value and other values that must save before allowing another ISR to execute. The stack need not be at a distinct memory block when different ISRs execute and the ISR stack is at a common memory block when there is nesting. (This is similar to the stack that associates with the functions.) Processor hardware may or may not provision for allowing the ISRs to execute in the nested mode.	Each task code run by change in program counter instantaneous value. Each task has a distinct task stack at the distinct memory block for the context (program counter instantaneous value and other CPU register values in task control block) that must save when blocking from its running state due to an interrupt or preemption by another higher priority task. Each task has a distinct process structure (TCB) at the distinct memory block.
4.	***Response and synchronization***: A function calls another function and there is nesting of one another [Figure 7.4(b)]. There is a hardware mechanism for sequential nested mode synchronization between the functions directly without the control of scheduler or OS.	There is a hardware mechanism for responding to an interrupt for the interrupt source calls and there is, according to the given OS kernel feature, a synchronizing mechanism for the ISRs. (Refer to the next chapter; Figures 8.1(a) to (c) and 8.4.	According to the given OS kernel feature, there is a task-responding and synchronizing mechanism. The kernel functions are used for task synchronization because only the kernel calls a task to run at a time. When a task runs and when it blocks it is fully under the control of the OS.[1]

(Contd)

Assume OSSemPost ( ) is an OS function for IPC by posting a semaphore and assume OSSemPend ( ) is another IPC function for waiting the semaphore. Let *sdispT* is the binary semaphore posted from *Chocolate delivery task* and taken by a *Display task* section for displaying the thank you message. Let *sdispT* initial value = 0. The following will be the codes.

```
static void Task_Deliver (void *taskPointer) {
.
while (1) {
.
/* Codes for delivering a chocolate into a bowl. */
.
OSSemPost (sdispT) /* Post the semaphore sdispT. This means that OS function increments sdispT in
    corresponding event control block. sdispT becomes 1 now. */
.
    };
static void Task_Display (void *taskPointer) {
.
while (1) {
.
OSSemPend (sdispT) /* Wait for the semaphore sdispT. This means that task waits till sdispT is posted
    and becomes 1. When sdispT becomes 1, the wait is over, an OS function runs to decrement sdispT
    in corresponding event control block, sdispT becomes 0 now, and Task then runs further the
    following code*/
/* Code for display "Collect the nice chocolate. Thank you, visit again" */
.
    };
```

1. Semaphore provides a mechanism to allow section of the task code wait till another notifies an action (finish running of a section of the codes at a task or ISR). It provides a way of signalling an event occurrence. It provides a way of signalling taking of a note of the event. Semaphore can be used as a signalling or notifying variable (token).
2. Semaphore increments when posted (sent or released) by a task or ISR instruction and decrements when accepted or taken by the waiting task section.
3. A waiting task section is notified to start on sending the semaphore. A waiting task section starts on taking the semaphore.

7.7.2 Use of a Semaphore as Resource Key and for Critical Section

OS provides for the use of a single semaphore as a resource key and for running of the codes in critical section. A task *A*, when getting access to a resource (e.g., printer file or network or section of codes called critical section or printer) notifies to the OS to have taken the semaphore (take notice). [An OS function, e.g., OSSemPend ( ) runs to notify. The OS returns the semaphore as taken (accepted) by decrementing the semaphore from 1 to 0.] Now, the task *A* accesses the resource (e.g., accesses the file, or network or runs the section of codes).

The task *A*, after completing access to a resource (e.g., memory buffer or file or network, or critical section) it notifies to the OS to have posted that semaphore (post notice). [An OS function, e.g., OSSemPost ( ) runs to notify. The OS returns the semaphore as released by incrementing the semaphore from 0 to 1.]