11. Internet-enabled embedded devices and their network protocols
12. Wireless protocols for mobile and wireless networks

3.1 IO TYPES AND EXAMPLES

A serial port is a port for serial communication. Serial communication means that over a given line or channel one bit can communicate and the bits transmit at periodic intervals generated by a clock. A serial port communication is over short or long distances.

A parallel port is a port for parallel communication. Parallel communication means that multiple bits can communicate over a set of parallel lines at any given instance. A parallel port communicates within the same board, between ICs or wires over very short distances of at most less than a meter.

A serial or parallel port can provide certain special features and sophistication (Section 3.4) by using a processing element.

Ports can interconnect by wireless. Wireless or mobile communication is serial communication but without wires, can be over a short-range personal area network as well as long-range wireless network, and transmission takes place by using carrier frequencies. The carrier modulates the serial bits before transmission in air [Sections 3.5 and 3.13]. A receiver demodulates and retrieve the serial bits back.

Serial and parallel ports of IO devices can be classified into following IO types: (i) Synchronous serial input (ii) Synchronous serial output (iii) Asynchronous serial UART input (iv) Asynchronous serial UART output (v) Parallel port one bit input (vi) Parallel one bit output (vii) Parallel port input (viii) Parallel port output. Some devices function both as input and as output; for example, a modem.

3.1.1 Synchronous Serial Input

The part 1 in Figure 3.1(a) shows a synchronous input serial port. Each bit in each byte and each received byte is in synchronization. Synchronization means separation by a constant interval or phase difference [part 2 in Figure 3.1(a)]. If clock period equals T, then each byte at the port is received at input in period 8 T. The bytes are received at constant rates. Each byte at the input port separates by 8 T and data transfer rate for the serial line bits is 1/T bps [1 bps = 1-bit per second]. The sender, along with the serial bits, also sends the clock pulses SCLK (serial clock) to the receiver port pin. The port synchronizes the serial data-input bits with clock bits.

The serial data input and clock pulse-input are on same input line when the clock pulses either encode or modulate serial data input bits suitably. The receiver detects clock pulses and receives data bits after decoding or demodulating.

When a separate SCLK input is sent, the receiver detects at the middle, positive or negative edge of the clock pulses that indicate whether data-input is 1 or 0 and saves the bits in 8-bit shift register. The processing element at the port (peripheral) saves the byte at a port register from where the microprocessor reads the byte.

Synchronous serial input is also called master output slave input (MOSI) when the SCLK is sent from the sender to the receiver and slave is forced to synchronize sent inputs from the master as per the master clock inputs. Synchronous serial input is also called master input slave output (MISO) when the SCLK is sent to the sender (slave) from the receiver (master) and the slave is forced to synchronize sending the inputs to master as per the master clock's outputs.

Synchronous serial input is used for interprocessor transfers, audio inputs and streaming data inputs.

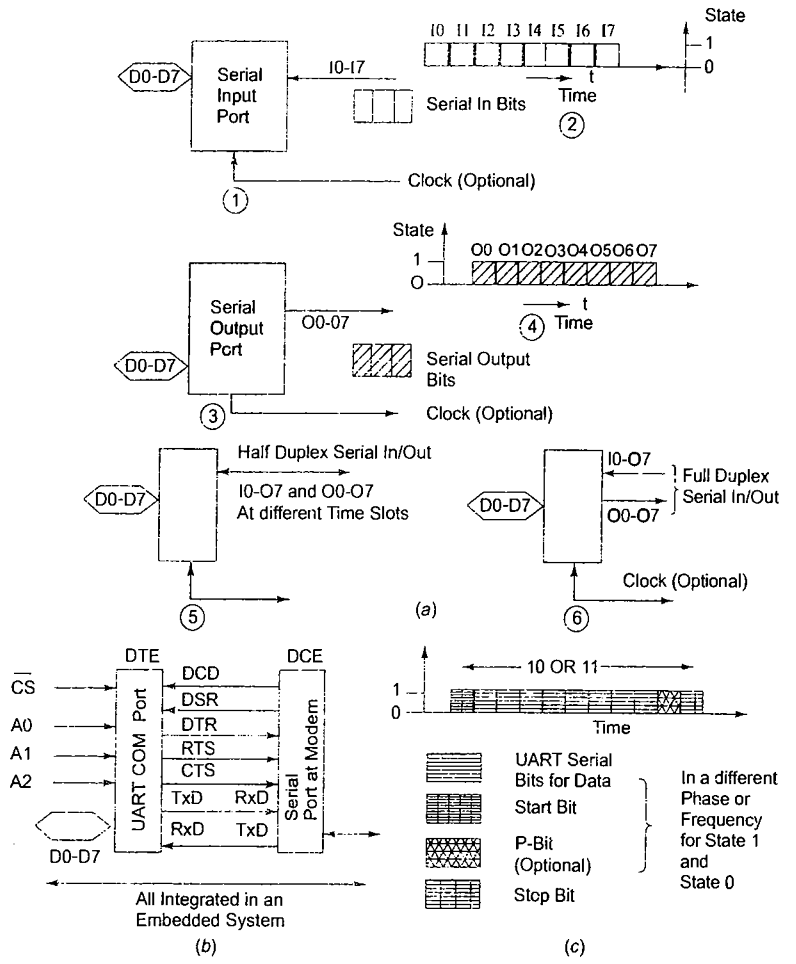


Fig. 3.1 (*a*) Input serial port, Output Serial port, Bi-directional half-duplex serial port, and Bi-directional full-duplex serial port (*b*) Handshaking signals at COM port in computer and (*c*) a UART serial port bits

3.1.2 Synchronous Serial Output

The part 3 in Figure 3.1(a) shows a synchronous output serial port. Each bit in each byte is in synchronization with a clock. The bytes are sent at constant rates [part 4 in Figure 3.1(a)]. If the clock period equals T, then the data transfer rate is 1/T bps. The sender sends either the clock pulses at SCLK pin or the serial data output and clock pulse-input through same output line when the clock pulses either suitably modulate or encode the serial output bits.

The processing element at the port (peripheral) sends the byte through a shift register at the port to which the microprocessor writes the byte.

Synchronous serial output is used for interprocessor transfers, audio outputs and streaming data outputs.

3.1.3 Synchronous Serial Input–Output

The part 5 in Figure 3.1(a) shows a synchronous serial input–output port. Each bit in each byte synchronizes with the clock input and output. The bytes are sent or received at constant rates as shown in parts (2) and (4) in Figure 3.1(a)]. The IOs are on same IO line when the clock pulses suitably modulate or encode the serial input and output, respectively. If the clock period equals T, then the data transfer rate is 1/ T bps. The processing element at the port (peripheral) sends and receives the byte at a port register to or from which the microprocessor writes or reads the byte.

Synchronous serial input/outputs are also called master input slave output (MISO) and master output slave input (MOSI), respectively.

They are used for interprocessor transfers and streaming data. The bits are read from or written on magnetic media such as a hard disk or on optical media such as a CD by using devices with serial synchronous IO ports.

The part 6 in Figure 3.1(a) shows the IO synchronous port when input and output lines are separate.

3.1.4 Asynchronous Serial input

Figure 3.1(b) shows the asynchronous input serial port line, denoted by RxD (receive data). Each RxD bit is received in each byte at fixed intervals but each received byte is not in synchronization. The bytes can separate by variable intervals or phase differences. Figure 3.1(c), on the right side, shows the starting point of receiving the bits for each byte, indicated by a line transition from 1 to 0 for a period T. When a sender shifts after every clock period T, then a byte at the port is received at input in period 10·T or 11·T. The time of 2 T is due to use of additional bits at the start and end of each byte. An addition time of 1T is taken when a P-bit is sent before the stop bit.

The bit transfer rate (for the serial line bits) is (1/ T) baud per second but different bytes may be received at varying intervals. The word 'Baud' is taken from a German word for raindrop. Bytes pour from the sender like raindrops at irregular intervals. The sender does not send the clock pulses along with the bits.

The receiver detects *n* bits at the intervals of T from the middle of the first indicating bit. n = 0, 1, ..., 10 or 11, finds out whether the data-input is 1 or 0 and saves the bits in an 8-bit shift register. The processing element at the port (peripheral) saves the byte at a port register, from where the microprocessor reads the byte.

Asynchronous serial input is also called UART input if the serial input is according to the UART protocol (Section 3.2.3). Asynchronous serial input is used for keypad and modem inputs.

3.1.5 Asynchronous Serial Output

Figure 3.1(b) shows the asynchronous output serial port line, denoted by TxD (transmit data). Each bit in each byte is sent at fixed intervals but each output byte is not in synchronization (it is separated by a variable interval or phase difference). The Figure 3.1(c) shows the starting point of sending the bits for each byte, which is indicated by a line transition from 1 to 0 for a period T. The sender port of TxD does not send clock pulses along with the bits.

The sender transmits bytes at the minimum intervals of n T. Bits start from the middle of the start indicating bit, where n = 0, 1, ..., 10 or 11 and sends the bits through a 10- or 11-bit shift register [Figure 3.1(c)]. The processing element at the port (peripheral) sends the byte at a port register to where the microprocessor writes the byte.

Asynchronous serial output is also called UART output if the serial output is according to a UART protocol (Section 3.2.3). Asynchronous serial output is used for modem and printer inputs.

3.1.6 Parallel Port

A parallel port can have one or multibit input or output and can be bi-directional IO.

(i) One bit input, output and IO

(ii) Eight or more bit input, output and IO

Section 3.3 will describe parallel device ports in detail.

3.1.7 Half Duplex and Full Duplex

The part 5 in Figure 3.1(a) on the left side shows the IO serial port (bi-directional half-duplex serial port). Half duplex means that at any point communication can only be one way (input or output) on a bi-directional line. An example of half-duplex mode is telephone communication. On one telephone line, we can talk only in the half-duplex mode. The part 6 in Figure 3.1(a) shows the separate input and output serial port lines. Full duplex means that the communication can be both ways simultaneously. An example of the full duplex asynchronous mode of communication is communication between the modem and computer through the TxD and RxD lines [Figure 3.1(b)].

There are two types of communication ports for IOs: serial and parallel. Serial line port communication is synchronous when a clock of the master device controls the synchronization of the bits on the line. Serial line port communication is asynchronous when clocks of the sender and receiver are independent and bytes are received, not necessarily at constant phase differences. Serial communication can be full duplex, which means simultaneously communication both ways, or half duplex, which means one way communication.

3.1.8 Examples of Serial and Parallel Port IOs

Table 3.1 gives a classification of IO devices into various types. It also gives examples of each type.

Table 3.1 Examples of various types of IO devices

IO Device Type	*Examples*
Serial synchronous input	Inter-processor data transfer, reading from CD or hard disk, audio input, video input, dial tone, network input, transceiver input, scanner input, remote controller input, serial IO bus input, reading from flash memory using SDIO (Secure Data Association IO) card
Serial synchronous output	Inter-processor data transfer, multiprocessor communication, writing to CD or hard disk, audio output, video output, dialer output, network device output, remote TV Control, transceiver output, and serial IO bus output, writing to flash memory using SDIO card
Serial asynchronous input	Keypad controller serial data-in, mice, keyboard controller data in, modem input, character inputs on serial line [also called UART (universal receiver and transmitter) input when according to UART mode]

(*Contd*)

IO Device Type	*Examples*
Serial asynchronous output	Output from modem, output for printer, the output on a serial line [also called UART output when according to UART mode]
Parallel port single bit input	(i) Completion of a revolution of a wheel, (ii) achieving preset pressure in a boiler, (iii) exceeding the upper limit of the permitted weight over the pan of an electronic balance, (iv) presence of a magnetic piece in the vicinity of or within reach of a robot arm to its end point and (v) filling a liquid up to a fixed level
Parallel port single bit output	(i) PWM output for a DAC, which controls liquid levei, temperature, pressure, speed or angular position of a rotating shaft or a linear displacement of an object or a d.c. motor control (ii) pulses to an external circuit
Parallel port input	(i) ADC input from liquid level measuring sensor or temperature sensor or pressure sensor or speed sensor or d.c. motor rpm sensor (ii) Encoder inputs for bits for angular position of a rotating shaft or a linear displacement of an object
Parallel port output	(i) LCD controller for multiline LCD display matrix unit in a cellular phone to display on screen the phone number, time, messages, character outputs or pictogram bit-images or e-mail or web page (ii) print controller (iii) stepper motor coil driving output bits

3.2 SERIAL COMMUNICATION DEVICES

3.2.1 Synchronous, Iso-synchronous and Asynchronous Communications from Serial Devices

Synchronous Communication When a byte (character) or frame (a collection of bytes) of data is received or transmitted at constant time intervals with uniform phase differences, the communication is called ***synchronous***. Bits of a data frame are sent in a fixed maximum time interval. ***Iso-synchronous*** is a special case when the maximum time interval can be varied.

An example of synchronous serial communication is frames sent over a LAN. Frames of data communicate, with the time interval between each frame remaining constant. Another example is the inter-processor communication in a multiprocessor system. Table 3.2 gives a synchronous device port bits.

Figure 3.1(a) part 2 showed the serial IO bit format and serial line states as a function of time. Two characteristics of synchronous communication are as follows:

1. Bytes (or frames) maintain a constant phase difference. It means they are synchronous, that is, in synchronization. There is no permission for sending either the bytes or the frames at random time intervals; this mode therefore does not provide for handshaking *during* the communication interval. [Handshaking means that the source and destination first exchange the signals between them before they communicate the data bits.] The master is the one whose clock pulses guide the transmission and slave is the one which synchronizes the bits as per the master clock.
2. A clock ticking at a certain rate must always be there to serially transmit the bits of all the bytes (or frames). The clock is not always implicit to the synchronous data receiver. The transmitter generally transmits the clock rate information in the synchronous communication of the data.

Table 3.2 Synchronous device port bits

S.No.	*Bits at Port*	*Compulsory or Optional*	*Explanation*
1.	Sync code bits or bi-sync code bits or frame start and end signaling bits	Optional	A few bits (each separated by interval ΔT) as Sync code for frame synchronization or signaling precedes the data bits[1]. There may be inversion of code bits after each frame. Flag bits at start and end are also used in certain protocols
2.	Data bits	Compulsory	m frame bits or 8 bits transmit such that each bit is at the line for time ΔT or each frame is at the line for time $m.\Delta T$ [2]
3.	Clock bits	Mostly not optional	Either on a separate clock line or on a single line such that the clock information is also embedded with the data bits by an appropriate encoding or modulation

[1]Reciprocal of ΔT is the transfer rate in bit per second (bps).
[2]m may be a large number. It depends on the protocol.

Figure 3.2 gives ten methods by which synchronous signals, with the clocking information, are sent. (i) There are two separate lines for the data bits and clock. The parallel-in serial-out (PISO) and serial-in parallel-out (SIPO) are used for transmitting and receiving the signals for data, respectively. (ii) There is a common line and the clock information is encoded by modulating the clock with the stream of bits. (iii) There are preceding and succeeding additional synchronizing and signaling bits. There are five common methods of encoding the clock information into a serial stream of the bits: (a) Frequency Modulation (FM (b) Mid Frequency Modulation (MFM) (c) Manchester coding (d) Quadrature amplitude modulation (QAM) (e) Bi-phase coding. The synchronous receiver separates serial bits of the message as well as synchronizing clock.

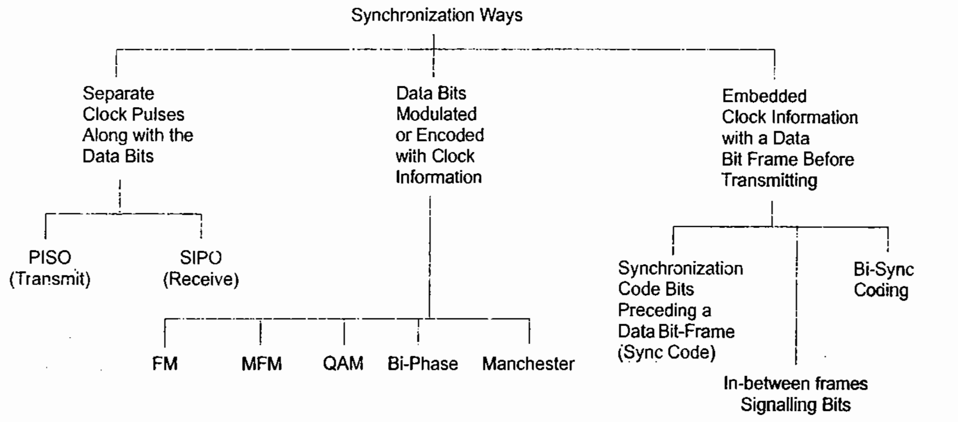

Synchronization Ways
Separate Clock Pulses Along with the Data Bits
Data Bits Modulated or Encoded with Clock Information
Embedded Clock Information with a Data Bit Frame Before Transmitting
PISO (Transmit)
SIPO (Receive)
FM
MFM
QAM
Bi-Phase
Manchester
Synchronization Code Bits Preceding a Data Bit-Frame (Sync Code)
Bi-Sync Coding
In-between frames Signalling Bits

Fig. 3.2 Ten ways by which the synchronous signals with the clocking information transmit from a master device to slave device

Asynchronous Communication When a byte (characters) or frame (a collection of bytes) of data is received or transmitted at variable time intervals, communication is called *asynchronous*. Voice data on the line is sent in asynchronous mode. Over a telephone line the communication is asynchronous. Another example is keypad communication.

An example of mode of asynchronous communication is RS232C communication between the UART devices (Section 3.2.2).

UART communication (Section 3.2.3) for asynchronous data is used for the transfer of information between the keypad or keyboard and computer.

Two characteristics of asynchronous communication are as follows:

1. Bytes (or frames) need not maintain a constant phase difference and are asynchronous, that is, not in synchronization. Bytes or frames can be sent at variable time intervals. This mode therefore facilitates in-between handshaking between the serial transmitter port and serial receiver port.
2. Though the clock must tick at a certain rate to transmit bits of a single byte (or frame) serially, it is *always implicit* to the asynchronous data receiver. The transmitter *does not* transmit (neither separately nor by encoding using modulation) along with serial stream of bits any clock rate information in asynchronous communication. The receiver clock is thus not able to maintain identical frequency and constant phase difference with the transmitter clock.

When a device sends data using a serial communication frame, it may not be as simple as shown in Figures 3.1(a) and (b) or as given in Table 3.2. It can be complex and has to be as per the protocol, which is followed by transmitting and receiving devices during communication between them.

Example 3.1

An IBM personal computer has two COM ports (communication ports), COM1 and COM2. These have 8 bytes at IO addresses 0x3F8 and 0x2F8.

Figure 3.1(b) showed COM port handshaking signals besides TxD and RxD. When a modem connects, it detects a carrier signal on the telephone line. A modem sends *data carrier detect* DCD signal at time t_0. A modem then communicates *data set ready* (DSR) signal at time t_1 when it receives the bytes on the line. The receiving end responds at time t_2 by data terminal ready (DTR) signal. After DTR, *request to send* (RTS) signal is sent at time t_3 and the receiving end responds by *clear to send* (CTS) signal at time t_4. After the response CTS, the data bits are transmitted by modem from t_5 to the receiver terminal at successive intervals [Figure 3.1(c)]. Between two sets of bytes sent in asynchronous mode, the handshaking signals RTS and CTS can again be exchanged. This explains why the bytes do not remain synchronized during asynchronous transmission.

A communication system may use the following protocols for synchronous or asynchronous transmission from a device port: RS232C, UART, HDLC, X.25, Frame Relay, ATM, DSL and ADSL. These are protocols for networking the physical devices in telecommunication and computer networks. Ethernet and token ring are protocols used in LAN networks. There are a number for protocols for serial communication. RS232C, UART and HDLC are described in Sections 3.2.2 to 3.2.4.

The protocols in embedded network devices such as bridges, routers, embedded Internet appliances use bridging, routing, application and web protocols. Internet enabled embedded systems use *application* protocols — HTTP (hyper text transfer protocol), HTTPS (hyper text transfer protocol Secure Socket Layer), SMTP (Simple Mail Transfer Protocol), POP3 (Post office Protocol version 3), ESMTP (Extended SMTP), TELNET (Tele network), FTP (file transfer protocol), DNS (domain network server), IMAP 4 (Internet Message Exchange Application Protocol) and Bootp (Bootstrap protocol) and others (Section 3.11).

Embedded wireless appliances use wireless protocols— IrDA, Bluetooth, 802.11 and others (Section 3.13).

Synchronous, iso-synchronous and asynchronous are three ways of communication. Clock information is transmitted explicitly or implicitly in synchronous communication. The receiver clock continuously maintains constant phase difference with the transmitter clock. HDLC is a data link protocol for computer networks and telecommunication devices. RS232C and UART are asynchronous mode communication standard.

3.2.2 RS232C/RS485 Communication

(i) RS232C RS232C communication is between DTE (computer) COM (communication) port and DCE (modem) port. DTE stands for 'Data Terminal Equipment'. DCE stands for 'Data Communication Equipment'. RS232C is an interfacing signal standard between DCE and DTE.

Figure 3.1(b) showed the interfacing (handshaking) signals on a RS232C port. The receive data and transmit data signals from RS232C port are RxD and TxD, respectively. RS232C port serial RxD and TxD bits are asynchronous and follow UART protocol (Section 3.2.3). Receiver end voltage level from – 3 to – 25 V denotes logic 1 and voltage level from +3 to + 25 V denotes logic 0. Transmitter end voltage level from – 5 to – 15 V denotes logic 1 and voltage level from + 5 to + 15 V denotes logic 0.

Example 3.2

The IBM personal computer two COM ports (communication ports) called COM2 and COM1, have IO addresses 0x2F8-0x2FF and 0x3F8-0x3FF, respectively. The COM port is RS232C port. It has TxD and RxD serial output and input. It has handshaking signals $\overline{DCD}$, $\overline{DSR}$, $\overline{DTR}$, $\overline{RTS}$ and $\overline{CTS}$.

RS232C port in a computer is used upto 9600 baud/s asynchronous serial transmission rate with UART mode communication. Generally baud rates are set at 300, 600, 1200, 4800 and 9600. When transmitting upto 0.25 m or 1 m on cable (untwisted) the maximum baud rate can be 115.2 k or 38.4 k baud/s, respectively.

RS232C port is used for keyboard serial communication at 1200 baud/s asynchronous serial transmission rate with UART mode communication at IBM PC COM port. The signals used are $\overline{RTS}$, $\overline{CTS}$, TxD and RxD for keypad communication. A mice port also is RS232C COM port in the computer. (New mice nowadays use USB in place of COM port).

Example 3.3

A mobile smart phone has a Bluetooth device for personal area wireless network. A Bluetooth device is capable of emulating DCE serial port, which can now communicate in UART mode.
A computer on the other hand has a serial port called COM port (Example 3.1). The mobile device is placed on a cradle. The mobile device port data-pins connect the cradle pins. The cradle connects to the computer or laptop COM port. The mobile and computer serial ports then communicate. The data (for example, pictures or address book data) between them synchronize between COM and emulated serial at Bluetooth device.

RS232C standard provides for UART serial port asynchronous mode communication. A different set of voltage levels are prescribed for the 0s and 1s in RS232C standard.

(ii) RS485 RS485, now called EIA-485 is a protocol for physical layer in case of two wire full or half duplex serial connection between multiple points. Transmission is at 35 Mbps upto 10 meter and 100 Kbps up to 1.2 km. Electrial signals are between + 12 V and –7 V. Logic 1 is +ve and 0 is reverse polarity. Difference in potential defines logic 1 and 0. A converter is used to convert RS232C bits to RS485 and another for vice versa.

3.2.3 UART

Figure 3.1(b) showed handshaking signals of RS232C port and UART serial bits in the output to a serial line device. The UART mode is as follows:

1. A line is in non-return to zero (NRZ) state. It means that in the idle state the logic state is 1 at serial line.
2. The start of serial bits is signaled by $1 \rightarrow 0$ transition (negative edge) on the line for a period equal to reciprocal of baud rate. The baud rate is preset at both receiver and transmitter. The receiver detects the start bit at middle of the interval, logic 0 state of the transmitter start bit.
3. UART bits, when sending a byte, consist of start bit, 8 data bits (for example, for an ASCII character or for a command word), option programmable bit (P-bit) and stop bit, each during the interval δT. When sending or receiving a byte, the logic states during interval $10\ \delta T$ or $11\ \delta T$ are as shown in Figure 3.1(c) as a function of time t. A bit period, δT is equal to the reciprocal of baud rate, the rate at which the bits from UART transmitter are sent. One extra bit before the stop bit is programmable bit P and is called TB8 at the transmitter and RB8 at the receiver.
4. The data bits in certain specific cases can be 5 or 6 or 7 instead of 8.
5. The stop bit can be for a minimum interval of $1.5\ \delta T$ or $2\ \delta T$ instead of δT in certain specific cases.
6. Optional programmable bit (P-bit) can be used for parity detection or can be used to specify the purpose of the serial data bits that are before the P-bit. For example, P can specify bits as the bits of a control or command word when P = 1 and data bits when P = 0. Bit P can specify the address of receiver when P = 1 and data when P = 0 so that only the addressed receiver wakes up and receives the data in the subsequent data transfers. When P is used as address/data specification, it provides a means to interface a number of UART devices through a common set of TxD and RxD lines and form a UART bus.

UART 16550 includes a 16-byte FIFO buffer and is nowadays used more commonly as compared to the original IBM PC COM port, which had an 8-bit register at UART port and was based on 8250 and did not include the FIFO buffer.

UART serial port communication is usually either in 10 bits or in 11 bits format: one start bit, 8 data bits, one optional bit and one stop bit. UART communication can be full duplex, which is simultaneously both ways, or half duplex, which is one way. It is an important communication mode.

3.2.4 HDLC Protocol

When data are communicated using the physical devices on a network, synchronous serial communication may be used. **HDLC** (High Level Data Link Control) is an International Standard protocol for a data link network. It is used for linking data from point to point and between multiple points. It is used in telecommunication and computer networks. It is a bit-oriented protocol. The total number of bits is not necessarily an integer multiple of a byte or a 32-bit integer. Communication is full duplex.

Table 3.3 gives the synchronous network device port bits in an HDLC protocol. The reader may refer to a standard textbook, for example, "*Data Communications, Computer Networks and Open Systems*" by Fred Halsall from Pearson Education (1996) for details of HDLC and its field bits.

Table 3.3 Format of bits in synchronous HDLC protocol-based network device

S.No.	*Bits[1] at Port*	*Present Compulsorily or Optionally*	*Explanation*
1	Frame start and end signaling flag bits	Compulsory	Flag bits at start as well as at end are (01111110)
2	Address bits for destination	Compulsory	8-bits in standard format and 16-bits in extended format
3a	Control bits Case 1: information frame	Compulsory as per case 1 or 2 or 3	First bit 0, next 3 bits N(S), next bit P/F[2] and last 3 bits N(R) in standard format N(R)[3] and N(S) = 7 bits each in extended format
3b	Control bits case 2: supervisory frame	—	First two bits (10), next 2 bits RR[3] or RNR or REJ or SREJ, next bit P/F and last 3 bits N(R) in standard format. N(R)[4] and N(S)[4] = 7 bits each in extendd format
3c	Control bits Case 3: unnumbered frame	—	First two bits (11), next 2 bits M[5], next bit P/F and last 3-bit remaining bits for M. [8 bits are immaterial after M bits in extended format]
4	Data bits	Compulsory	m frame bits transmit such that each bit is at the line for time ΔT or, each frame is at the line for time m.ΔT and also there is bit stuffing.[6]
5	FCS (Frame check sequence) bits	Compulsory	16-bits in standard format and 32 in extended format
6	Frame end flag bits	Compulsory	Flag bits at end are also (01111110)

[1] Bits are given in order of their transmission or reception.
[2] P/F = 1 and P means when a primary station (Command device) is polling the secondary station (receiving device). P/F = 1 and F means when receiving device has no data to transmit. Usually it is done in last frame.
[3] RR, RNR, REJ and SREJ are messages to convey 'Receiver ready,' 'Receiver not ready,' 'Reject,' and 'Selective reject'. REJ or SREJ is a negative acknowledgement (NACK). NACK is sent only when the frame is rejected.[*A child cries only when milk is not given on need, else it remains silent!*] 'Reject' means that the receiver received a frame out-of-sequence: it is rejected and a repeat transmission of all the frames from the point of frame rejection is requested using REJ. 'Selective reject' means that a frame is received out-of-sequence; it is to be rejected and a selective repeat transmission is requested for this frame using SREJ.
[4] N(R) and N(S) means received (earlier) and sending (now) frame sequence numbers. These are modulo 8 or 128 in standard or extended format frame, respectively.
[5] M five bits are for a command (or response) from a transmitter. Examples of a command are *reset*, *disconnect* or *set a defined mode type*; examples of a response are a message from the receiver for a disconnect mode accepted, frame rejected, command rejected, and for an unnumbered acknowledgement.
[6] When five 1s transmit for the data, one 0 is stuffed additionally. This prevents misinterpretation by receiver the data bits as flag bits (01111110).

3.2.5 Serial Data Communication using the SPI, SCI and SI Ports

Microcontrollers have internal devices for SPI or SCI or SI as explained below. Each device has separate registers for control, status, serially received data bits and transmitting serial bits. Each device is programmable as described below. The device can be used in programmed IO modes or in interrupt driven reception and transmission.

Synchronous Peripheral Interface (SPI) Port Figure 3.3(a) shows an SPI port signals. Figure 3.3(b) shows SPI port in 68HC11 and 68HC12 microcontroller. It has full-duplex feature for synchronous communication. There are signals SCLK for serial clock, MOSI and MISO output from and input to master.

Section 3.11. Figure 3.3(b) shows programmable features and DDR feature of Port D. An SPI feature is rogrammable rates for clock bits, and therefore for the serial out of the data bits down to the interval of).5 µs for an 8 MHz crystal at 68HC11.

SPI is also programmable for defining the occurrence of negative and positive edges within an interval of oits at serial data *out* or *in*. It is also programmable in the open-drain or totem pole output from a master to a lave and for device selection as master or slave. This can be done by a signal to hardware input SS (slave elect when 0) pin. In the hardware the slave select pin connects to '1' at the *master* SPI device and to '0' at he *slave*. Defining SPI as slave or master can also be done by software. Programming a bit at the device ontrol register does this.

68HC12 provides SPI communication device operations at 4 Mbps. SPI device operates up to 2 Mbps in 8HC11.

erial Connect Interface (SCI) Port Figure 3.3(c) shows an SCI port programmable features and DDR ort bits in 68HC11/12. SCI is a UART asynchronous mode port. Communication is in full-duplex mode for the SCI transmission and receiver. SCI baud rates are fixed as prescaling bits. Rate not programmable separately for ndividual serial *in* and *out* lines. A baud rate can be selected among 32 possible ones by the three-rate bits and wo prescaling bits. The SCI receiver has a *wake up* feature and is programmable by RWU (Receiver wakeup navailable) bit. It is enabled if RWU (1st bit of SCC2, Serial Communication Control Register 2) is set, and is isabled if RWU is reset. If RWU if set, then the receiver of a slave is not interrupted by the succeeding bytes. SCI as two control register bits, TB8 and RB8. RWU feature helps in inter-processor communication, and SCI is lefined for transmission and for reception using the SCC2 bits. UART communication, when programmed by control bits, is in 11-bit format. A number of processors can communicate on the bus in UART mode by RWU, when RB8 and TB8 bits are set.

There are separate hardware devices at 68HC11 for synchronous and asynchronous communications. These re SPI and SCI, respectively. 68HC12 provides two SCI communication devices that can operate at two ifferent clock rates. Standard baud rates can be set up to 38.4 kbps. There is only one SCI and standard baud ates in 68HC11 can be set up to 9.6 kbps only.

erial Interface (SI) Port Figure 3.3(d) shows an SI port. SI is a UART mode asynchronous port interface. t also functions as USRT (universal synchronous receiver and transmitter). SI is therefore a synchronous–synchronous serial communication port called USART (universal synchronous–asynchronous receiver and ransmitter) port. It is an internal serial IO device in 8051. There is an on-chip common hardware device alled SI in Intel 80196. Its features are as follows: programmable-rates register after loading the 14 bits at BAUD_RATE register twice. SI operates in one of the following ways:

(i) Half-duplex synchronous mode of operation, called mode 0. When a 12 MHz crystal is at 8051 and is attached to the processor, the clock bits are at the intervals of 1 µs.

(ii) Full-duplex asynchronous serial communication, called mode 1 or 2 or 3. Using a timer, the baud rate varies according to the programmed timer bits in modes 1 and 3. Using SMOD bit at SFR called PCON, when mode 2 is used, the baud rate is programmable at two rates only. It is 1/64 or 1/32 of oscillator frequency at 8051. TB8 and RB8, when using 11-bit format, provide the 10th bit for error-detection or for indicating whether the sent data byte is a command or data for the receiving SI device.

Most microcontrollers have internal serial communication SPI and SCI or SI-like devices for serial communication. The IBM personal computer has two UART chips for the two COM ports. Table 3.4 gives the features of internal serial ports in select microcontrollers.

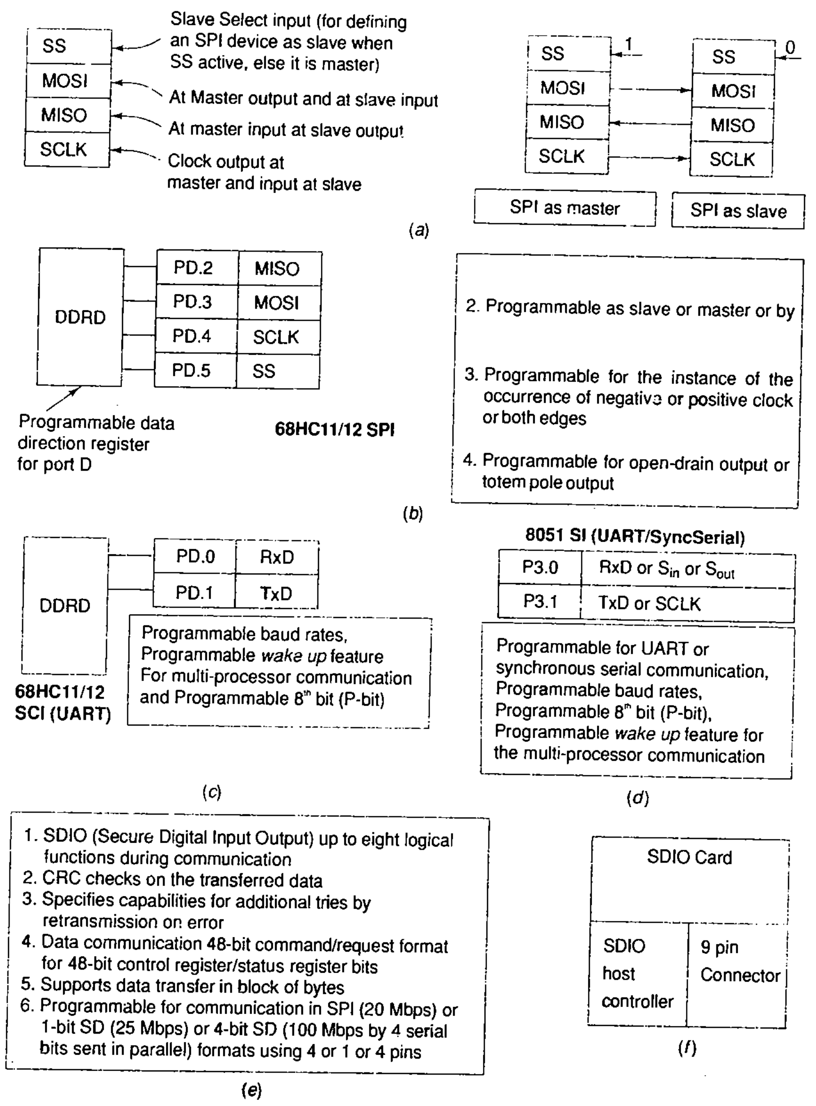

SS
MOSI
MISO
SCLK
Slave Select input (for defining an SPI device as slave when SS active, else it is master)
At Master output and at slave input
At master input at slave output
Clock output at master and input at slave
1
0
SPI as master
SPI as slave
(a)
DDRD
PD.2 MISO
PD.3 MOSI
PD.4 SCLK
PD.5 SS
Programmable data direction register for port D
68HC11/12 SPI
2. Programmable as slave or master or by
3. Programmable for the instance of the occurrence of negative or positive clock or both edges
4. Programmable for open-drain output or totem pole output
(b)
DDRD
PD.0 RxD
PD.1 TxD
Programmable baud rates, Programmable wake up feature For multi-processor communication and Programmable 8th bit (P-bit)
68HC11/12 SCI (UART)
(c)
8051 SI (UART/SyncSerial)
P3.0 RxD or Sin or Sout
P3.1 TxD or SCLK
Programmable for UART or synchronous serial communication, Programmable baud rates, Programmable 8th bit (P-bit), Programmable wake up feature for the multi-processor communication
(d)
1. SDIO (Secure Digital Input Output) up to eight logical functions during communication
2. CRC checks on the transferred data
3. Specifies capabilities for additional tries by retransmission on error
4. Data communication 48-bit command/request format for 48-bit control register/status register bits
5. Supports data transfer in block of bytes
6. Programmable for communication in SPI (20 Mbps) or 1-bit SD (25 Mbps) or 4-bit SD (100 Mbps by 4 serial bits sent in parallel) formats using 4 or 1 or 4 pins
(e)
SDIO Card
SDIO host controller
9 pin Connector
(f)

Fig. 3.3 (*a*) SPI port signals (*b*) SPI port programmable features and DDR at port D in 68HC11/12 (*c*) SCI port in 68HC11/12 (*d*) SI port in 8051 (*e*) SDIO communication features (*f*) SDIO card structure

Table 3.4 Processor with internal serial ports in microcontrollers

Features	*Intel 8051 and Intel 8751*	*Motorola M68MC11E2*	*Intel 80196*
Synchronous serial port (half or full duplex)	Half	Full	Half
Asynchronous UART port (half or full duplex)	Full	Full	Full
Programmability for 10 as well as 11 bits per byte from UART	Yes	Yes	Yes
Separate un-multiplexed port pins for synchronous and UART serial ports	No	Yes (Separate 4 Pins)	No
Synchronous serial port as a master or slave definition by software or hardware	Software	Hardware and software	Software
UART serial port programmability as a transmitter or receiver and for additional bit for parity or RWU or control or other purpose.	Yes	Yes	Yes
Synchronous serial port registers	SCON. SBUF and TL-TH 0-1	SPCR. SPSR and SPDR	SPCON SPSTAT BAUD_RATE and SBUF
UART serial port registers	SCON. SBUF and TL-TH 0-1 (Time 2 in 8052)	BAUD. SCC1 SCC2. SCSR SCIRDR and SCITDR	SPCON SPSTAT BAUD_RATE and SBUF
Uses internal timer or uses separate programmable BAUD rate generator.	Timer	Separate	Separate as well as the Timer

Microcontrollers have internal devices of three types SPI, SCI and SI. SPI is synchronous master slave mode serial full duplex communication. SCI is UART asynchronous transmitter-receiver mode serial full duplex communication. SI is synchronous half duplex and asynchronous full duplex UART serial communication.

3.2.6 Secure Digital Input Output (SDIO)

Secure Digital (SD) Association created a new flash memory card format, called SD format. It is an association of over 700 companies started from 3 companies in 1999. This SDIO card for SD format IOs [Figure 3.3(e)] has become a popular feature in handheld mobile devices, PDAs, digital cameras and handheld embedded systems. SD card size is just 0.14 × 2.4 × 3.2 cm. SD card [Figure 3.3(f)] is also allowed to stick out of the handheld device open slot, which can be at the top in order to facilitate insertion of the SD card.

SDIO is an SD card with programmable IO functionalities such that it (a) can be used upto eight logical functions, (b) can provide additional memory storage in SD format, and (c) can provide IOs using protocols in systems such as IrDA adapter, UART 16550, Ethernet adapter, GPS, WiFi, Bluetooth, WLAN, digital camera, barcode or RFID code reader.

Figure 3.3(e) shows an SDIO communication device port features. It supports SPI (Section 3.2.5), 4-bit and 1-bit SD formats. Both SPI and SD formats specify that there should be interrupt handling of the IOs and also the CRC checks on transferred data, and specifies capabilities for more tries on error. SDIO card [Figure 3.3(f)] has 9 pins. Six pins are for communication using SPI or SD. A processing element function is used as SDIO host controller to process the IOs. The controller may include SPI controller to support SPI mode for the IOs and supports the needed protocol functionality internally. Maximum clock rate supported for SPI is 20 MHz for a maximum of 20 Mbps data transfers. There is an optional 4-bit SD mode, which uses 4 data lines. Maximum clock rate supported is 25 MHz for maximum 100 Mbps SD bit data transfer in 4-bit SD mode. Four serial bits simultaneously transmits at four times clock rate on 4 SD lines in this mode. Four-bit SD mode is compromise between serial and parallel bits communication to enhance serial transfer rate four times. In 1-bit SD mode, with 25 MHz clock the maximum data transfer rate is 25 Mbps and one serial bit transmits at 1 line only.

SDIO card has a control section called function 0. It necessarily uses function 1 and optionally uses functions between 2 and 7 (depending on the application in devices used such as Bluetooth, PHS, GPS or digital camera). Each function has PCMIA (Personal Computer Manufacturer Interface Adapter) defined card information structure and registers, for example, ID number, function enable bit, supported bus width (1 or 4), voltage, power needs, clock rates and interrupt enable bit. Each function's specifications for the register bits and protocols have been defined in SD standard. A standard device driver can therefore be written. A new function can also be defined.

Data communication is in 48-bit command/request format for 48-bit control register/ status register bits and supports data transfer of blocks of bytes. For single byte transactions, SDIO card may also include a UART 16550 mode communication over the SD bus.

SDIO is an SPI based 9 pin connector card, which supports SPI as well as 1-bit SD or 4-bit SD communication. SDIO supports 8 logic functions. SDIO functions include IOs with several protocols, for example, IrDA adapter, UART 16550, Ethernet adapter, GPS, WiFi, Bluetooth, WLAN, digital camera, barcode or RFID code reader.

3.3 PARALLEL DEVICE PORTS

The parallel port of devices transfers number of bits over the wires in parallel. Parallel wires capacitive effect reduces the length up to which parallel communication can be done. High capacitance results in delay for the bits at the other end undergoing transition from 0 to 1 or from 1 to 0. High capacitance can also result in noise and cross talk (induced signals) between the wires. Therefore, parallel port carries the bits upto short distances, generally within a circuit board or IC.

Figure 3.4(a) shows the parallel input, output, and bi-directional device ports. Figure also shows a device-interfacing circuit with the processor and system buses. Parallel port inputs I0 to I7 may be to a keypad controller. Parallel port outputs O0 to O7 may be output bits to LCD display output controller. BR_i and BR_0 are the input and output data buffers at bi-directional IO port.

A device port connects to the address bus signals, A_i and A_j through a port address decoder. $\overline{IORD}$ and $\overline{IOWR}$ are additional control signals for a port/device read and write, respectively, in case of an 80x86 processor, which has IO mapped IOs. The memory read and write signals, $\overline{RD}$ and $\overline{WR}$ are used in the processor with memory mapped IOs [Section 2.2.2].

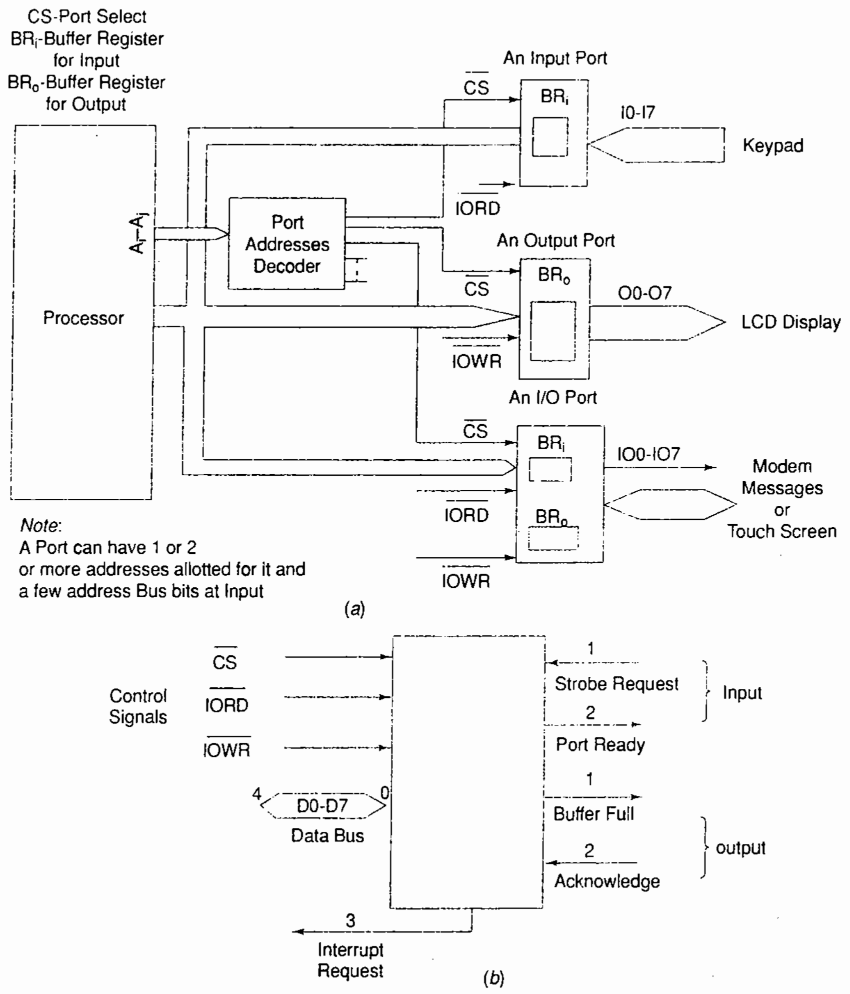


Fig. 3.4 (*a*) Parallel input port, output port, and a bi-directional port for connecting the device (*b*) The handshaking signals when used by the IO ports

Example 3.4

IBM personal computer has a parallel port with a 25 pin connector. There are 8 IO pins, 5 input pins for status signals (four active high S3 to S6, one active low $\overline{S7}$) from external device port (for example, printing device port) and 4 output pins for control signals (one active high C2 and three active low $\overline{C0}$, $\overline{C1}$ and $\overline{C3}$. The 8 pins are ground pins (Pins at 0 V). The status pins and control pins are provided for handshaking between peripheral and computer.

Figure 3.4(b) shows the handshaking signals. An external input device to the device port makes a strobe request, *STROBE*, after it is ready to send the byte and the system IO device sends the acknowledgement, *PORT READY* when BR$_i$ (receiving buffer) is empty.

An external output receiving device sends the message *ACKNOWLEDGE* when the IO device port ends the *BUFFER FULL* signal. The processor is sent the *INTERRUPT REQUEST* message when BR$_o$ transmitting buffer is empty not full (available for next write) or when the receiving buffer is full (available for next read). This enables the processor to interrupt and retransmit next byte(s) in next cycle or receive the byte(s) from input using the appropriate service routines for output or input from the port, respectively.

Example 3.5

Intel 8255 is a programmable peripheral interface (PPI) chip. A PPI device has four addresses, three for the ports and one for the control word. There are three 8-bit ports: port A, B and C. Port C can also be programmed to function in bit set-reset mode. Each bit of this port can be set to 1 or reset to 0 by an appropriate control word. Alternatively, the ports can be grouped as Group A (Port A and Port C upper four bits) or Group B (Port B and Port C lower four bits).

1. In mode 0 programming for a group, each port group does not use handshaking signals.
2. Mode 2 programming is used for port A as input as well as output. In mode 2 programming for the group A, port A uses handshaking signals, STROBE, PORT READY, BUFFER FULL, ACK and INTERRUPT and port A functions as a bi-directional IO port.
3. Mode 1 programming is either for port as input or as output. In mode 1 programming for the group A or B, port A or B uses only one of the two handshaking signal pairs, either (STROBE, PORT READY) or (BUFFER FULL, ACK) plus one INTERRUPT signal.

The following characteristics are taken into consideration when interfacing a device port.

1. A device port may have multi-byte data input buffers and data output buffers. Suppose there is an eight-byte buffer. Assuming that a device (as in the 80196 microcontroller) can generate three interrupts, one on receiving a byte, one on receiving the fourth byte and one when the buffer is full, then the deadline for servicing these interrupts increases up to eight times compared to the case when there is a single byte register instead of buffer.
2. A port may have a DDR (Data Direction Register) (as in the 68HC11 microcontroller). This is an advantage since each bit of the port is now programmable. It can be set as input or output. DDR programs the port bits.
3. Port LSTTL-driving capability and port-loading capability are important characteristics. A port may be an OD (open drain) port. It has zero driving capability unless the drain connects the positive supply voltage. If the given port has OD gates, an appropriate pull-up resistance or transistor is connected to each port pin to provide the driving capability. The drain or collector connects to the supply voltage to provide the pull-up.
4. If a given port is *quasi bi-directional* (as in 80196), then the port pins have limited driving capability, which suffices for a period of one or a few clock cycles and drives a LSTTL gate for that period. When this device port connects to more than one LSTTL, then an appropriate pull-up circuit will be required for the port pins.
5. There may be multiple or alternate functionality in the port pins; for example, 80196 input port pins. Each pin of P2 has an alternative use as multi-channel analog input facility for 8 analog inputs. Another example is 8051 two ports P0 and P2. These port bits also have an alternate function in that they bring out when needed the internal multiplexed buses for the external program and memories whenever the

internal memory is insufficient. Each pin of P3 in 8051 has multiple uses. These are used during serial communication, timer/counter signals, interrupt-signals, and $\overline{RD}$ and $\overline{WR}$ control signals for external memories. 68HC11 ports B and C are of 8 bits each and have alternative uses for the port pins in it. One of the alternate functions is to bring out the internal address and data buses, respectively.

6. A port may have provision for multiplexed output to connect to multiple systems or units.
7. A port may have provision for demultiplexed inputs from multiple systems or units.

A parallel device port can have parallel inputs, parallel outputs, bi-directional and quasi-bi-directional IOs. A parallel device port can have handshaking pins. A parallel device port can also have control pins for control-signal outputs to external circuit and status pins for inputs of status signals to external circuits.

3.3.1 Parallel Port Interfacing with Switches and Keypad

A 16 keys keypad has many applications. A mobile smart phone device has 16 keys and four menu: select up, down, left, right keys. Assume that an IO device has two ports, A and C. The device has a processing element which functions as a keypad-controlling device (controller).

Figure 3.5(a) shows how a set of switches or a keypad of 16 keys and four menu-select keys can interface to the device. Four bits of an 8-bit input port A (A_4-A_7) can be used for the four menu select keys. Assume that the idle state logic state equals 1. The 16 keys can be considered as arranged in four rows and four columns. The other four bits of A (A_0-A_3) are inputs from sense lines from four rows. Assume that the idle state logic state is equal to 1. The four bits of output port C (C_0-C_3) are output to sense lines in four columns.

The processing element in device activates for polling the output port C ten times each second and sends C_0-C_3 = 0000; after a wait it reads D_0-D_7 and A_4-A_7. The processing element computes the code of the pressed key and generates a status signal when a key is found pressed. From the bit pattern found at A_0-A_3, the processing element computes 7-bit ASCII code of the pressed key at that instance and can output that code at D_0-D_6. It also outputs D_7 = 1 when a specific key is found pressed, else D_7 = 0. The processing element also processes the bounces when a key is pressed. This takes care of bouncing effects. The processing element is thus functioning as a keypad controller, as it is keypad specific.

Example 3.6

A mobile phone keypad is smart and is called T9 keypad. Nine keys are used to enter not only the numbers but also text of messages. The processing element is programmed as a state machine to compute the ASCII code to be sent. A state machine generates the states. For example, a key marked as number 5 is in state (0, 5) in reset state, which is also its idle state. The key-state undergoes transition to state (1, 5) when it is pressed first time. When it is pressed second time within 1 s, the key state becomes (1, j). This state corresponds to character j. If it is pressed third time within 1 s, the key-state becomes (1, k). The state of the key changes in a cyclic fashion. (1, 5) → (1, j) → (1, k) → (1, l) → (l, 5) → (1, j), ... The transition of a key state occurs only if it is found pressed within 1 s of the previous transition, and the appropriate action takes place as per the state. The processing element computes the ASCII code from the read value of A_0-A_3 and key state at an instance. After processing is over or after 1s, the key-state resets to (0, 5).

Two key states simultaneously or separately undergoing transitions can define a transition to another state. For example, when there is transition to (1, j) state after another key state is (1, #), then (1, j) undergoes another transition to (1, j), and when that key state is (0, #) it remains at (1, j).

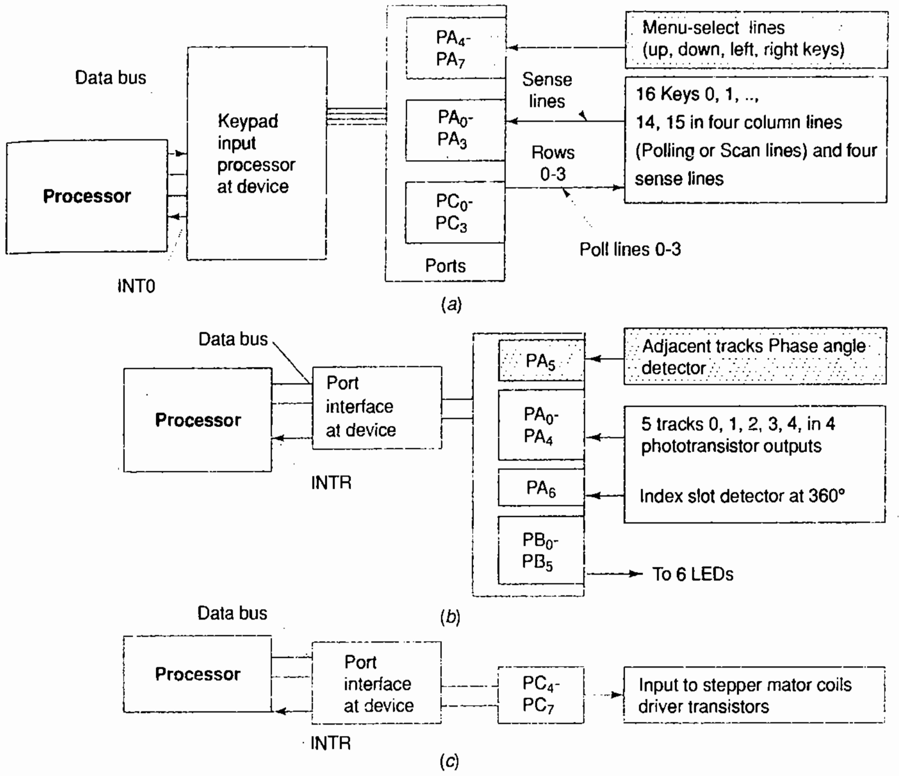

Data bus
Processor
Keypad input processor at device
INTO
PA4-PA7
PA0-PA3
PC0-PC3
Ports
Sense lines
Rows 0-3
Poll lines 0-3
Menu-select lines (up, down, left, right keys)
16 Keys 0, 1, .., 14, 15 in four column lines (Polling or Scan lines) and four sense lines
(a)
Data bus
Processor
Port interface at device
INTR
PA5
PA0-PA4
PA6
PB0-PB5
Adjacent tracks Phase angle detector
5 tracks 0, 1, 2, 3, 4, in 4 phototransistor outputs
Index slot detector at 360°
To 6 LEDs
(b)
Data bus
Processor
Port interface at device
INTR
PC4-PC7
Input to stepper motor coils driver transistors
(c)

Fig. 3.5 (*a*) Parallel input port A and a four bit output port C used for interfacing a set of 16 keys in keypad and four menu select keys (*b*) Parallel input port A connected to an encoder circuit which senses the rotated or linear position of a moving shaft and port B connected to 6 LEDS (*c*) Four bit *parallel output port* C connected to a stepper motor

A parallel device having a number of input and output bits can be used to find the code of the pressed key in a matrix of keys. A keypad controller has a processing element to compute the code of the pressed key and to generate a status signal when any key is found pressed. A mobile phone keypad controller processes the states of the keys to enable application of same keypad for dialing as well as editing SMS messages.

3.3.2 Parallel Port Interfacing with Encoders

Encoder is a device that measures angular or linear position of a rotating or moving shaft. It has application in robots and industrial plants. A rotatory-angle encoder has multiple tracks on a rotating disk. Each track has half of the segments transparent and half opaque. A linear encoder has a multi-slotted plate. A set of n infrared (IR) LED and phototransistor pairs generate n-bit inputs for a port. The encoder connects to parallel port, as shown in Figure 3.5(b).

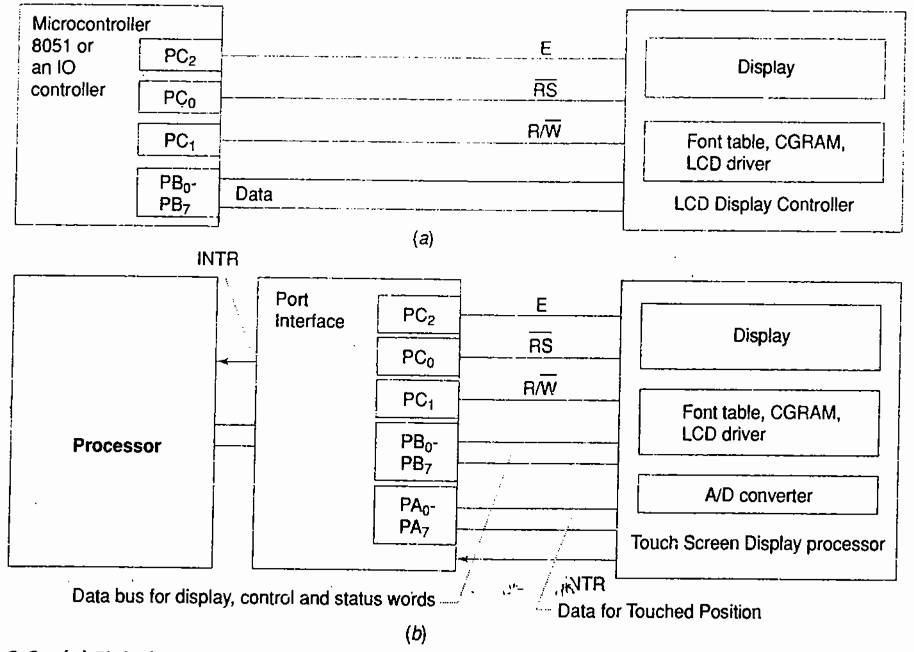
3.3.3 Parallel Port Interfacing with Stepper Motor

A stepper-motor rotates by one step angle when its four coils are given currents in a specific sequence and that sequence is altered. For example, assume that currents at an instance equal + i, 0, 0, 0 in four coils X, X', Y, Y'. The motor rotates by one step when the currents change to 0, + i, 0, 0. The sequences at intervals of T are changed as follows: 1000, 0100, 0010, 0001, 1000, 0100, ... [The bits in the nibble (set of 4 bits) rotate by right shift.] Here 1 corresponds to + i. The motor thus rotates n step angles in interval of (n.T). The sequences are changed to rotate the motor in the opposite direction, as follows: 0001, 00010, 0100, 1000, 0001, 0010, ... [The bits in the nibble (set of 4 bits) rotate by left shift.] Alternately, the coils are given the currents in the sequence of 1100, 0110, 0011, 1001, 1100, 0110, ..., or 0011, 0110, 1100, 1001, 0011, 0110, ... The motor rotates (n/2) steps in interval equals to (n.T/2). T is the period of clock pulses that drives the motor by change of coil currents to the next sequence.

The coils connect to parallel port 4 output pins, as shown in Figure 3.5(c). Alternatively, a processing element called stepper-motor deriver can be used. The driver is given two outputs from the port: clock pulses and a rotating direction bit r. For example, if r = 1, motor rotates clockwise and if r = 0 then motor rotates anti-clockwise. The motor rotates as long as clock pulses are given at the output PC_4-PC_7.

3.3.4 Parallel Port Interfacing with LCD Controller

An LCD controller has a processing element that needs three control signals as inputs and 8 input/output bits for parallel set of 8 IO bits. Eight-bit *parallel output port B* pins PB_0-PB_7 connect LCD controller, as shown in Figure 3.6(*a*). LCD controller also connects to one bit PC_0 at an output port for RS (register select) signal.



Microcontroller 8051 or an IO controller
PC2
PC0
PC1
PB0-PB7
E
RS
R/W
Data
Display
Font table, CGRAM, LCD driver
LCD Display Controller
(a)
INTR
Processor
Port Interface
PC2
PC0
PC1
PB0-PB7
PA0-PA7
E
RS
R/W
Display
Font table, CGRAM, LCD driver
A/D converter
Touch Screen Display processor
INTR
Data bus for display, control and status words
Data for Touched Position
(b)

Fig. 3.6 (*a*) Eight bit parallel output port B connected to an LCD controller (*b*) 8-bit parallel output port B and 8-bit parallel input port A connected to a touch screen control circuit

When RS is reset as 0, PB_0-PB_7 communicates a control word to control register of the LCD controller. When RS is set as 1, PB_0-PB_7 communicates data to the LCD controller.

The LCD controller also connects to a one bit PC_1 at output port for $R/\overline{W}$ (read/write). PC_1 is set to 1 when status register of LCD controller is read from PB_0-PB_7. PC_1 is reset to 0 when writing into LCD controller the PB_0-PB_7 bits. The processing element generates all signals required for LCD displays.

The LCD controller is sent control words and data words for initialization and programming PB_0-PB_7 bits, PC_0 and PC_1 outputs for each word to LCD controller. The controller then has to be enabled by sending 1 at E pin. It connects to one bit PC_2 at output port for E (enable). There is an interval in which the controller may be in disabled state. During this interval, it cannot accept instructions or data through the output of control word or data port pins. For example, a control instruction is to clear display. The internal processing element has to clear the bytes at all the N addresses in N characters LCD display. Assume that in a typical LCD, it is 150 µs. When the first 1 is written at E, then 0 is written and a 150 µs delay program is called in-between; the E output creates a negative going pulse at LCD controller. It disables sending of any control word or data for a period of 150 µs.

LCD controller has M displayed character ROM addresses. M = 128 for 128 ASCII codes. For each distinct ASCII character, there is a 64-bit graphic. The LCD controller has an internal CGRAM (character graphic RAM). For each ASCII character, 8 bytes are sent from font table ROM to CGRAM address. CGRAM has N addresses. N = 64 when 64 characters are displayed at the LCD. An address changes by incrementing or decrementing the cursor position to the previous or next address on screen. By sending appropriate control words followed by data, the LCD controller is programmed to display up to 64 characters on the screen.

A parallel device having 8 output data and 3 bits for E, $\overline{RS}$ and $R/\overline{W}$ can be used to connect to an LCD controller.

3.3.5 Parallel Port Interfacing with Touchscreen

Touchscreen is an input device cum LCD display device. It is also interfaced through IO port B functioning as data bus far display, control and status words to an LCD display device controller. The interface uses an additional input port A for a byte, which corresponds to the address of the position touched on the screen.

A touchscreen is either resistive or capacitive. On touching at a position on the screen, there is change in resistance or capacitance, which depends on the touched position. A touch can be finger or stylus. The stylus is about one-fifth thinner than a pencil and about half of the length of the pencil. The resistance or capacitance is a part of a bridge circuit that generates an analog voltage. An 8-bit ADC is given an input from a bridge circuit and the 8-bit ADC output connects to 8-bit input port A.

Eight-bit *parallel IO port B* pins PB_0-PB_7, E, $\overline{RS}$ and $R/\overline{W}$ and eight-bit *parallel input port* A connect to touch screen circuit ports as shown in Figure 3.6(b). An interrupt signal INTR is issued whenever the screen is touched.

Example 3.7

A PocketPC has a touchscreen. The touchscreen device facilitates GUIs. It can display menus as well as a virtual keypad. Using the keypad on screen and stylus, a set of characters can be entered for creating or editing SMS messages, e-mails, or other files. The stylus is held like a pencil and is used to touch the virtual keypad and then the device selects the menu and commands on the screen.

A parallel device having 8 input data bits from an ADC and 8 IO data bus and 4 bits for INTR, E, $\overline{RS}$ and R/$\overline{W}$ can be used to connect a port interface with a processing element to a touchscreen. The ADC generates input bits for the port from the analog signal, which is as per the touched position on the screen.

3.4 SOPHISTICATED INTERFACING FEATURES IN DEVICE PORTS

A device port may not be as simple as the one for a stepper motor port or for a serial line UART. Nowadays, a complex embedded system has highly sophisticated IO devices, for example, SDIO card (Section 3.2.6), IO devices with fast serialization and de-serialization of data, fast transceiver, and real time video processing system. The following are the few sophisticated interfacing device and port features.

1. Let the operation voltage level expected for logic state 1 = 5 V (TTL or CMOS). The Schmitt trigger circuit has a property that when a transition from 0 to 1 occurs, only if the voltage level exceeds 2/3 of the 5 V level is there a transition to 1. Similarly, when a transition from 1 to 0 occurs, only if the voltage level lowers below 1/3 of the 5 V level is there a transition to 0. Hence, the Schmitt trigger circuit eliminates noise as large as 2/3 of 5 V, or 3.3 V, when it is superimposed at an input line to the device. One great advantage of the in-built Schmitt trigger circuit at the port is conditioning of the signal by noise elimination. Otherwise, a device port input will need an external chip for Schmitt trigger-based noise elimination. Such a device is used in transceivers for repeating systems, which are used in long distance communication.
2. When a device port is waiting for instructions, power management can be done at the gates of the device. Lately, a new technology called DataGate (from Xilinx) has been developed for use at ports. DataGate is a programmable ON/OFF switch for power management; DataGate makes it possible to reduce power consumption by reducing unnecessary toggling of inputs when these are not in use. The great advantage of an inbuilt DataGate-like circuit at a device port is reduced power dissipation when the device port is operated at fast speeds. Such a device is extremely useful in systems connected to a common bus and there is a need to control unnecessary input toggling. For example, in a bus interface unit, the input signals should activate only when the input has to be passed to the circuit. As the number of bus interfaces in the system grows, the demand to prevent needless switching of input signals increases.
3. Earlier, port interfaces used to be either open drain CMOSs or TTLs or RS232Cs. (*i*) Nowadays, a system may be required to operate at a voltage lower than 5 V. [Recall Section 1.3.1.] Low Voltage TTL (LVTTL) and Low Voltage CMOS (LVCMOS) gates may be used at the device ports for 1.5 V IO. (*ii*) Nowadays, a system may be required to operate using advanced IO standard interfaces. Examples are High Speed Transreceiver Logic (HSTL) and Stub-series Terminated Logic (SSTL) standards. HSTL is used for high-speed operations; SSTL is used when the buses are to be isolated from relatively large stubs.
4. A device connects to a system bus and also to IO bus when it is networked with other devices. Device and bus-impedances during an IO should match. Else, line reflections occur. Recent developments make it feasible to match these dynamically. For example, a new technology, called, XCITE (Xilinx Controlled Impedance Technology) can be used. The great advantage of an inbuilt device for dynamically matched impedances is that when resistors are replaced with digitally, dynamically controlled and matched impedances in the devices, there are no line reflections and therefore no missing bits or bus faults.

5. An IO device may consist of multiple gigabit (622 Mbps to 3.125 Gbps) transceivers (MGTs). Special support circuitry is needed for this rate. Rocker IO ™ serial transreceivers are examples of circuits that provide support circuitry at this rate.
6. A device for an IO may integrate a SerDes (serialization and de-serialization) subunit. SerDes is a standard subunit in a device where the bytes placed at 'transmit holding buffer' serialize on transmission, and once the bits are received these de-serialize and are placed at the 'receiver buffer'. Once the device SerDes subunit is configured, serialization and de-serialization is done automatically without the use of the processor instructions. The great advantage of the SerDes unit is that these operations are fast when compared to operations without SerDes. [A device for IO may integrate a DAA (direct access arrangement using analog IOs along with one master and seven slave CODECs) or McBSP (multi channel buffered serial port with high speed communication) subunit when serializing.
7. Recently, multiple IO standards have been developed for IO devices. A support to the multiple IO standards may be needed in certain embedded systems. A technology, Flexible Select IO ™ -Ultra technology, supports over 20 single-ended and differential IO signaling standards. Advantages of multiple standard device ports are obvious.
8. An IO device may integrate a digital Physical Coding Sublayer (PCS). Analog audio and video signals can then be pulse code modulated (PCM) at the sublayer. The PCS sublayer directly provides codes from analog inputs within the device itself. The codes are then saved in the device data buffers. The advantage of an inbuilt PCS at device port is that there is then no need of external PCM coding. Besides, these operations are performed in the background as well as fast. It improves the system's performance when there are multimedia inputs at the device.
9. A device for IO may integrate an analog unit Physical Media Attachment (PMA) for connecting direct inputs and outputs of voice, music, video and images. The great advantage of inbuilt PMA is that the device directly connects to physical media. PMA is needed for real-time processing of video and audio inputs at the device.

Nowadays, IO devices have sophisticated features. Schmitt trigger inputs are used for noise elimination. Devices with low voltage gates and devices using power management by preventing unnecessary toggling at the inputs are used for sophisticated applications. Dynamically controlled impedance matching is a new technology and it eliminates line reflections when interfacing the devices. The SerDes subunit serializes and deserializes outputs and inputs in the devices. A port may have DAA, McBSP, PCS and PMA subunits for analog IOs for video and audio devices.

3.5 WIRELESS DEVICES

Wireless devices have become very common in recent years for serial transmission of bits.

Wireless devices use infrared (IR) or radio frequencies after suitable modulation of data bits. IrDA (Section 3.13.1), Bluetooth (Section 3.13.2), WiFi, 802.11 WLAN (Section 3.13.3) and ZigBee (Section 3.13.4) have become popular protocols for wireless communication of data bits from a source to the receiver.

An IR source communicates over a line of sight and the receiver phototransistor is used for detecting infrared rays. Example of applications of IR communication includes handheld TV remote controllers and robotic systems. IR devices use IrDA protocol.

Radio frequencies communicate over short and long distances. The transmitter and receiver use antennae to transmit and receive signals and modulator and demodulators to carry the data bits using RF frequencies. Mobile GSM wireless devices use 890–915 MHz, 1710–1785 MHz, or 1850–1910 MHz bands. Mobile CDMA wireless devices use 2 GHz carrier frequencies. Bluetooth and ZigBee wireless devices (Sections 3.13.2 and 3.13.4) use 2.4 GHz or 900 MHz frequencies.

The number of frequency bands is limited, while a large number of devices may need to communicate. Therefore, time and frequency division multiplexing are used. An innovative method is radio frequency hopping over a wider spectrum, as in Bluetooth devices. The transmitted carrier frequencies hop among different channels at a given hopping rate. The transmitter modulates the data bits as per protocol specifications. The receiver tunes to these hopped carrier frequencies at a given hopping rate and in the same hopping sequence as the ones used by the transmitter. The receiver demodulates and detects the data bits as per physical-layer protocol used for transmitting.

Several wireless devices network use FHSS or DSSS transmitters and receivers. Popular protocols are IrDA, Bluetooth, 802.11 and ZigBee.

3.6 TIMER AND COUNTING DEVICES

Most embedded systems need a timing device.

3.6.1 Timing Device

A timer device is a device that counts the regular interval (δT) clock pulses at its input. The counts are stored and incremented on each pulse. It has output bits (in a count register or at the output pins) for the period of counts. The counts multiplied by interval δT gives the time. The (counts–initial counts) × δT interval gives the time interval between two instances when the present count bits are read and the initial counts are read. It has an input pin (or a control bit in a control register) for resetting to make all count bits = 0. It has an output pin (or a status bit in status register) for output when all count bts equal0 after reaching the maximum value, which also means timeout on the overflow.

3.6.2 Counting Device

A counting device is a device that counts the input for events that may occur at irregular or regular intervals. The counts gives the number of input events or pulses since it was last read.

Blind Counting Synchronization A counting device may be a free running (blind counting) device with a prescaler for the clock input pulses and for comparing the counts with the ones preloaded in a compare register. The prescalar can be programmed as p = 1, 2, 4, 8, 16, 32, ..., by programming a prescaler register. It divides the input pulses as per the programmed value of p. It has an output pin (or a status bit in the status register) for output when all count bits equal 0 after reaching the maximum value, which also means after timeout or on overflow. The counter overflows after p × 2^n × δT interval. It can have an input pin (or a control bit in control register) for enabling an output when all count bits equal count preloaded in the compare register. At that instance, a status bit or output pin also sets in and an interrupt can occur for event of comparison equality. This device is useful for the alarm or processor interrupts at preset instances or after preset intervals with respect to another event from another source.

The counting device may be the free running (blind counting) device with a prescalar for the clock input pulses, for comparing the counts with the ones preloaded in a compare register as well as for capturing counts on an input event. This device functions are similar to the above, but there is an addition input pin for sensing an event and for saving the counts at the instance of that event. At this instance, a status bit can also set in and a processor interrupt can occur for the capture event.

The above device is useful for alarm generation and processor interrupts at the preset times as well as for noting the instances of occurrences of the events and processor interrupts for requesting the processor to use the captured counts on the events. Alarm generation can be synchronized with the input capture events. Writing counts into the compare register does this. Counts in the register are set equal to capture register counts plus additional counts, which define the interval after which an alarm is to be generated.

A blind counting free running counter with prescaling, compare and capture registers has a number of applications. It is useful for action or initiating a chain of actions, and processor interrupts at the preset instances as well as for noting the instances of occurrences of the events and processor interrupts for requesting the processor to use the captured counts on the events for future actions.

3.6.3 Timer cum Counting Device

A timer cum counting device is a counting device that has two functions. (1) It counts the input due to the events at irregular instances and (2) It counts the clock input pulses at regular intervals. An input or a status bit in the timing device register controls the mode as timer or counter. The counts gives the number of input events or pulses since it was last read. It has an output pin (or a status bit in status register) for output when all count bits equal 0 after reaching the maximum value, which also means timeout or overflow interrupts to the processor.

Table 3.5 lists twelve uses of a timer device. It also explains the meaning of each use.

Table 3.5 Uses of Timer Device

S.No.	*Applications and Explanation*
1.	Real Time Clock Ticks (functioning as system heart beats). [Real time clock is a clock that once the system starts it, does not stop and can't be reset. Its *count value* can't be reloaded. *Real time endlessly flows and never returns!*] Real Time Clock is set for ticks using prescaling bits and rate-set bits in appropriate control registers. Section 3.8 gives the details.
2.	Initiating an event after a preset delay time. Delay is as per *count-value* loaded.
3.	Initiating an event (or a pair of events or a chain of events) after a comparison between the preset time with counted value. Preset time is loaded in a Compare Register. [It is similar to presetting an alarm.]
4.	Capturing the *count-value* at the timer on an event. The information of *time* (instance of the event) is thus stored at the *capture register.*
5.	Finding the time interval between two events. *Counts* are captured at each event in the capture register and read. The intervals are thus found out. A service routine does the counts read on interrupt.
6.	Wait for a message from a queue or mailbox or semaphore for a preset time when using an RTOS. There is a predefined waiting period before RTOS lets a task run without waiting for the message. (Section 7.4)

(Contd)

S.No.	Applications and Explanation
7.	Watchdog timer. It resets the system after a defined time. Section 3.7 gives details.
8.	Baud or Bit Rate Control for serial communication on a line or network. Timer timeout interrupts define the time of each baud.
9.	Input pulse counting when using a timer, which is ticked by giving non-periodic inputs instead of the clock inputs. The timer acts as a counter if, in place of clock inputs, the inputs are given to the timer for each instance to be counted.
10.	Scheduling of various tasks. A chain of software-timer interrupts and RTOS uses these interrupts to schedule the tasks.
11.	Time slicing of various tasks. A multitasking or multiprogrammed operating system presents the illusion that multiple tasks or programs are running simultaneously by switching between programs very rapidly, for example, after every 16.6 ms. This process is known as *context switch*. RTOS switches after preset time-slice from one running task to the next. Each task can therefore run in predefined slots of time.
12.	Time division multiplexing (TDM). Timer device is used for multiplexing the input from a number of channels. Each channel input is allotted a distinct and fixed-time slot to get a TDM output. [For example, multiple telephone calls are the inputs and TDM device generates the TDM output for launching it into the optical fibre.]

A timing device has number of states and Table 3.6 gives the states.

Table 3.6 States in a timer

S.No.	States
1.	Reset State (initial count equals 0)
2.	Initial Load State (initial count loaded)
3.	Present State (counting or idle or before start or after overflow or overrun)
4.	Overflow State (count received to make count equal 0 after reaching the maximum count)
5.	Overrun State (several counts received after reaching the overflow state)
6.	Running (Active) or Stop (Blocked) state
7.	Finished (Done) state (stopped after a preset time interval or timeout)
8.	Reset enabled/disabled State (enabled resetting of count equal 0 by an input)
9.	Load enabled/disabled State (reset count equals initial count after the timeout)
10.	Auto Re-Load enabled/disabled State (enabled count equals initial count after the timeout)
11.	Service Routine Execution enable/disable State (enabled after timeout or overflow)

At least one hardware timer device is a must in a system. It is used as a system clock. Let number of system clock ticks needed before a system interrupt occurs equals *numTicks*. The hardware timer gets the input from a clock-out signal from the processor and activates the system clock tick as per the *numTicks* preset at the hardware timer. On each system clock tick, the user-mode task interrupts and the system takes control. The system enables the privileged mode actions and the CPU context switches as per the preset state of the system. The system control actions are performed by operating system (software).

Figure 3.7 shows hardware timer control bits (and signals) and status flags. *Control bits* are as per the hardware signals and corresponding bits at control register. Control bits (or signals) can be of nine types. These are: (i) Timer enable (to activate a timer). (ii) Timer start (to start counting at each clock input). (iii) Timer stop (to stop counting) from the next clock input. (iv) Prescaling bits (to divide the clock-out frequency signal from the processor). (v) Up count Enable (to enable counting up by incrementing the count value on each clock input) (vi) Down count Enable (to decrement on a clock input). (vii) Load enable (to enable loading of a value at a register into the timer). (viii) Timer-interrupt enable (to enable interrupt servicing when the timer outs (overflows) and reaches *count value* equals 0) (ix) Time out enable [to enable a signal when the timer overflows (reaches count equals 0)] to another device.

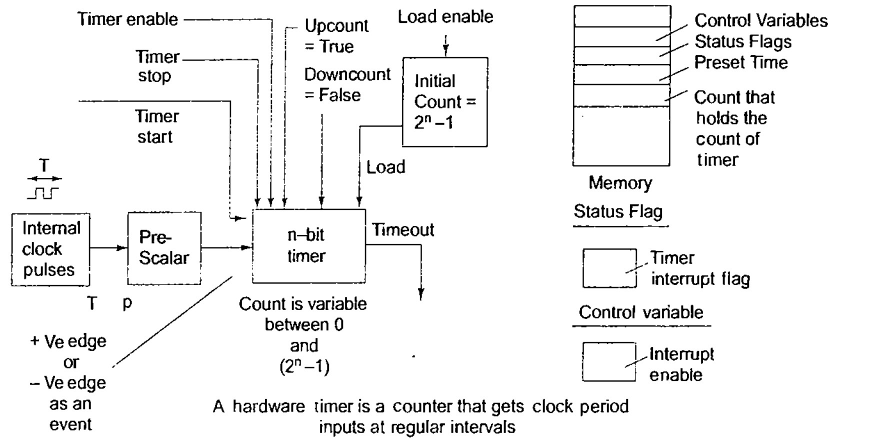


Fig. 3.7 Signals, clock-inputs, control bits and status flags at registers or memory in a hardware timer device

Status flag is as per the corresponding hardware signal time-out from the hardware timer. This flag and signal set when the timer all bits (*count value*) reach to 0.

Table 3.7 lists ten forms of the timers for the uses listed in Table 3.5. Software timer (SWT) is an innovative concept.

The system clock or any other hardware-timing device ticks and generates one interrupt or a chain of interrupts at periodic intervals. This interval is as per the *count-value* set. Now, the interrupt becomes a clock input to an SWT. This input is common to all the SWTs that are in the list of activated SWTs. Any number of SWTs can be made active put in a list of active SWTs. Each SWT will set a status flag on its timeout (*count-value* reaching 0). Figure.3.7 shows the control bits and status bits in an SWT. SWT *control bits* are set as per the *application*. There is no hardware input or output in an SWT. A flag sets when the SWT count-value reaches 0 after reading the maximum. Table 3.8 lists all the variables of SWT. It includes the control-bits and status flags. SWT thus has similar control variables and flags as in the hardware timer or counter.

SWT actions are analogous to that of a hardware timer. While there is physical limit (1, 2 or 3 or 4) for the number of hardware timers in a system, SWTs can be limited by the number of interrupt vectors provided by the user. Processors (microcontrollers) also define the interrupt vector addresses of two or four SWTs.

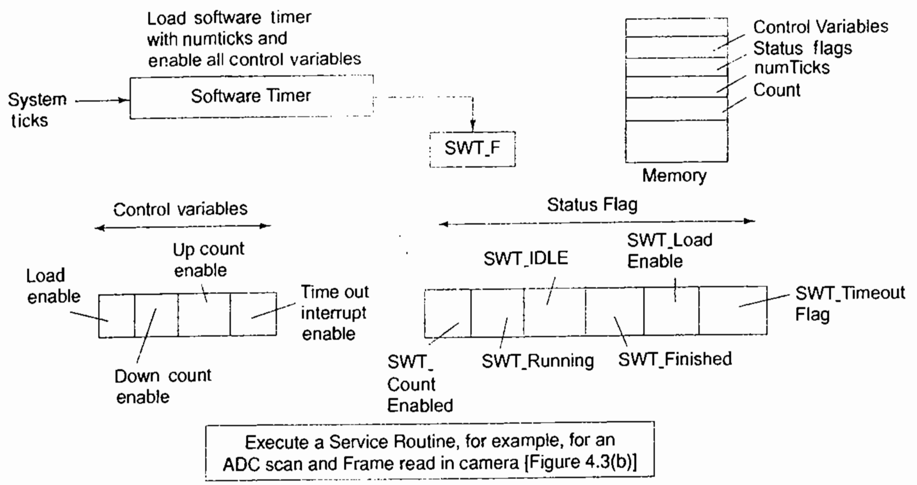


Fig. 3.8 Control bits, status flags and variables of a software timer

Table 3.7 Ten forms of a timer

S.No.	*Types*
1.	Hardware internal timer
2.	Software timer (SWT)
3.	User software-controlled hardware timer.
4.	RTOS-controlled hardware timer. An RTOS can define the clock ticks per second of a hardware timer at a system. [Refer to function OS_Ticks (N) in Section 9.2.1.]
5.	Timer with periodic time-out events (auto-reloading after overflow state). A timer may be programmable for auto-reload after each time-out.
6.	One Shot Timer. (No reload after the overflow and finished state.) It triggers on event-input for activating it to running state from its idle state. It is also used for enforcing time delays between two states or events. On the event or reaching a state one shot timer starts and after the time out another state or event occurs.
7.	Up count action Timer. It is a timer that increments on each count-input from a clock.
8.	Down count action timer. It is a timer that decrements on each count-input.
9.	Timer with its overflow status bit (flag), which auto-resets as soon as interrupt service routine starts running.
10.	Timer with overflow-flag, which does not auto reset.

Timing devices are needed for a number of uses in a system. (i) There can only be a limited number of hardware timers present in the system. A system has at least one hardware timer. The system clock is configured from this. A microcontroller may have 2, 3 or 4 hardware timers. One of the hardware timer ticks forms the inputs from the internal clock of the processor and generates the system clock. Using the systems clock or internal clock, the number of software timers can be drven. These timers are programmable by the device driver programs. (ii) A software timer is software that executes and increases or decreases a count-variable (count value) on an interrupt on a timer output or on a real-time clock interrupt. The software timer also generates interrupt on overflow of count-value or on finishing value. Software timers are used as virtual timing devices. There are a number of control bits and a time-out status flag in each timer device.

Table 3.8 Variables for control bits and status in a software timer

S.No.	*32 or 16 or 8 or 1-bit variables*
1.	Reset Value 32/16/8
2.	Initial Load Value (numTicks) 32/16/8
3.	Count-value (Preset value) 32/16/8
4.	Maximum Value 32/16/8
5.	Minimum Value 32/16/8
6.	Timer run enable bit
7.	Timer interrupt enable bit
8.	Timer reset enable bit
9.	Timer load enable bit
10.	Timer reload (after finished state) enable bit
11.	Overflow-flag

3.7 WATCHDOG TIMER

Watchdog timer is a timing device that can be set for a preset time interval, and an event must occur during that interval else the device will generate the timeout signal. For example, we anticipate that a set of tasks must finish within 100 ms. The watchdog timer disables and stops in case the tasks finish within 100 ms. The watchdog timer generates interrupts after 100 ms and executes a routine that runs because the tasks failed to finish in the anticipated interval. A software task can also be programmed as a watchdog timer (Section 9.3.3). A microcontroller may also provide for the watchdog timer.

The watchdog timer has a number of applications. One application in a mobile phone is that the display is turned off in case no GUI interaction takes place within a specified time. The interval is usually set at 15, 20, 25, or 30 s in a mobile phone. This saves power.

Another application in a mobile phone is that if a given menu is not selected by a click within a preset time interval, another menu can be presented or a beep can be generated to invite user's attention.

An application in a temperature controller is that if a controller takes no action to switch off the current within the preset time, the current is switched off and a warning signal raised, indicating controller failure. Failure to switch off current may cause a boiler in which water is heated to burst.

Example 3.8

68HC11 microcontroller has a watchdog timer in the hardware. There are two registers, CONFIG (system configuration control register) and COPRST (computer operating properly and processor reset on failure). They are for programming the interrupts of watchdog timer. CONFIG has a bit, NOCOP. It configures when the processor writes the configuration word at address 0x003F. NOCOP is the 2nd bit of CONFIG. If this bit is reset to 0, the COP facility is enabled. [COP means computer (68HC11) operating properly watchdog timer. The COP watchdog timer provides for keeping a watch on execution time of the user program.]

When user program takes a longer time in a routine than planned or expected the user provides for storing at desired intervals; first, the 0x55 and then the 0xAA at the computer-reset control register COPRST. By keeping a watch means that as soon as the watchdog timer overflows (time outs), the program counter is reset according to 16 bits at the lower and higher bytes that are preloaded at addresses 0xFFFA and

0xFFFB, respectively. If these 16 bits are same as the bits in 0xFFFE and 0xFFFF, then the microcontroller executes instructions, which are same as when it resets on power up or else it executes the routine at the 16-bit address fetched from 0xFFFE and 0xFFFF whenever there is failure within the watched time interval.

The 0th and 1st bit of the option register, OPTION, at the address 0x0039 are the CR_1 and CR_0 bits. If NOCOP resets (0) and CR_1-CR_0 = 0–0, the watchdog timer time out occurs after every 2^{16} pulses. As T = 0.5 μs for the processor when the E clock output is 2 MHz, the WDT time-out occurs at every 16.384 ms ($2^{16} \times 0.5$ μs) unless the user software stores at desired intervals before a time out, first the 0x55 and then the 0xAA at the computer reset control register COPRST. This means user program resets the watchlog timer by itself after finishing the watched section of the program. [After 2^{15} pulses if CR_1-CR_0 = 0–1, 2^{14} pulses for 1–0, 2^{13} pulses for 1–1].

A watchdog timer has a number of applications and is a timing device such that it is set for a preset time interval and an event must occur during that interval else the device will generate a timeout signal and interrupt for the failure to get that event in the watched time interval.

3.8 REAL TIME CLOCK

Real time clock (RTC) is a clock that causes occurances of regular interval interrupts on its each tick (timeout). An interrupt service routine executes on each timeout (overflow) of this clock. This timing device once started never resets or is never reloaded with another value. Once it is set, it is not modified later. The RTC is used in a system to save the current time and date. The RTC is also used in a system to initiate return of control to the system (OS) after the preset system clock periods.

Example 3.9

(*i*) Assume that a hardware timer of an RTC for calendar is programmed to interrupt after every 5.15 ms. Assume that at each tick (interrupt) a service routine runs and updates at a memory location. Within one day (86400 s) there will be 2^{24} ticks, the memory location will reach 0x000000 after reaching the maximum value 0xFFFFFF. Within 256 days there will be 2^{32} ticks, the memory location will reach 0x00000000 after reaching the maximum value 0xFFFFFFFF. Note that battery must be used to protect the memory for that long period.

(*ii*) Assume that an RTC has to implement using a software timer. Assume that a hardware 16-bit timer ticks from processor clock after 0.5 μs. It will overflow and execute an overflow interrupt service routine after 2^{15} μs = 32.768 ms. The interrupt service routine can generate a port bit output after every time it runs and can also call a software routine or sends a message for a task. If n = 30, the RTC initiated software will run every 30 × 32.768 ms, which is close to 1 s.

(*iii*) A real time clock timer for interrupts at regular intervals is present in a microcontroller. 68HC11 has a register called the Pulse Accumulator Control Register, PACTL and two lowest significance bits, RT_1-RT_0 (1st and 0th). PACTL is *write* only. If the RT_1-RT_0 pair is 00, an interrupt can occur after 2^{13} pulses of the E clock. If the E clock pulses are of 2 MHz and thus T is 0.5 μs, the interrupts from a real time clock occur after each 4.096 ms. If the RT_1-RT_0 pair is 01, an interrupt can occur after 2^{14} pulses of the E clock, that is, after each 8.192 ms. If the RT_1-RT_0 pair is 10, the interrupt can occur after 2^{15} pulses of the E clock, that is after each 16.384 ms. If the RT_1-RT_0 pair is 11, an interrupt can occur after each 2^{16} pulses of the E clock, that is, after each 32.768 ms. The real time clock is based on a free running counter in 68HC11. RT_1-RT_0 bits control its rate of ticking.

The interrupts from a real time clock are disabled or enabled by I bit in clock control (CC) register. The interrupts from real time clocks are also locally masked by the 6th bit, RTI in timer interrupt mask register2, TMASK2. This bit is set to unmask and reset to mask the real time clock interrupts. If RTI and I bits permit the interrupt request for real time clock timeout then the microcontroller fetches the lower and higher bytes of the interrupt servicing routine address from the addresses 0xFFF0 for higher byte) and 0xFFF1 (for lower byte). This is the vector address for real time clock interrupts in 68HC11. The interrupt service routine must clear (0) the RTIF, which is interrupt flag for the real time clock interrupts. The RTIF is a bit in timer interrupt flag register2, TFLG2. The TFLAG2 is at address 0x0025. It is set by each interrupt from the real time clock interrupt and therefore it must be cleared in order to enable next interrupt before returning from the corresponding service routine and before the next real-time clock-interrupt occurs.

A real time clock (RTC) provides system clock and it has a number of applications. It is a clock that generates system interrupts at preset intervals. An interrupt service routine executes on each tick (timeout or overflow) of this clock. This timing device once started is generally never reset or never reloaded to another value.

3.9 NETWORKED EMBEDDED SYSTEMS

Each specific IO device may be connected to others using specific interfaces; for example, an IO device connects and is interfaced to an LCD controller, keyboard controller or print controller using specific interface. Bus communication simplifies the number of connections and provides a common protocol for interconnecting different or same type of IO devices.

Any device that is compatible with a system's IO bus can be added to the system (assuming an appropriate device driver program is available), and a device that is compatible with a particular IO bus can be integrated into any system that uses that type of bus. This makes systems that use IO buses very flexible, as opposed to direct interconnections between the processor and each IO device, and it allows system support to many different IO devices (depending on the needs of its users), and it also allows users to change the IO devices that are attached to system as their needs change.

The main disadvantage of an IO bus (and buses in general) is that each bus has a fixed bandwidth that must be shared by all the devices, which connect to the bus. Even worse, electrical constraints (wire length and transmission line effects) cause buses to have less bandwidth than using the same number of wires to connect just two devices. Essentially, there is a trade-off between interface simplicity and bandwidth sharing. Assume that bandwidth of a bus is 200 Mbps. If the bus communicates two devices simultaneously then it does so by 100 Mbps communication by each.

IO devices communicate with the processor through an IO bus, which is separate from the memory bus that the processor uses to communicate with the memory system. Embedded systems connected internally on the same IC or systems at very short, short and long distances, and can be networked using the followings types of IO buses, each functioning according to specific protocols.

1. Using a serial IO bus allows a computer or controller or embedded system to interface network with a wide range of IO devices without having to implement a specific interface for each IO device. When

the IO devices in the distributed embedded systems are networked at long distances of 25 cm and above, all can communicate through a common serial bus. A serial bus has very few lines. Sections 3.10.1 to 3.10.5 describe the serial bus communication protocols.

2. Using a parallel IO bus allows a computer or controller or embedded system to interface with a number of internal systems at very short distances without having to implement a specific interface for each IO device. Section 3.11 describes the parallel bus communication protocols.
3. Using the Internet or intranet, a computer, controller or embedded system's IO device can interface globally and can network with other systems or computers and a wide range of devices in the distributed systems. Section 3.12 describes these systems.
4. Using a wireless protocol allows a handheld computer, controller or embedded system IO device to interface and network with a number of handheld system IO devices at short distances up to 100 m using a wireless personal area network (WPAN) protocol, without having to implement a specific wireless interface for each IO device. Section 3.13 describes wireless bus communication protocols.

Embedded systems are distributed and networked using a serial or parallel bus or wireless protocol software and appropriate hardware.

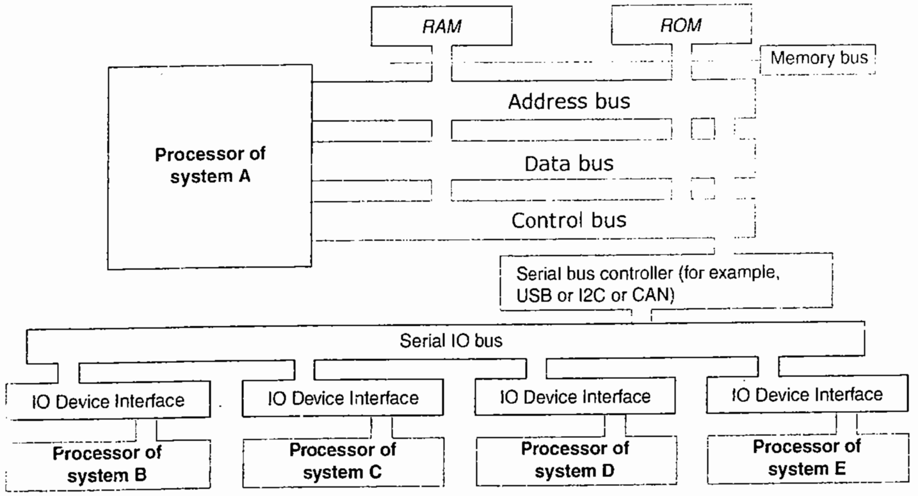


Fig. 3.9 A processor of embedded system connected to system memory bus and networked to other systems through a serial bus

3.10 SERIAL BUS COMMUNICATION PROTOCOLS

Figure 3.9 shows a processor of embedded system connected to system memory bus and networked to other systems through a serial bus. Sections 3.10.1 to 3.10.5 describe popular serial buses.

3.10.1 I^{2}C Bus

Assume that there are number of device circuits in a number of processes in a plant, one IC each for measuring temperatures and pressures. These ICs mutually network through a common synchronous serial bus. I^{2}C (Inter IC connect) bus is a popular bus for these circuits. There are three I^{2}C bus standards: Industrial 100 kbps I^{2}C, 100 kbps SM I^{2}C, and 400 kbps I^{2}C. The I^{2}C was originally developed at Philips Semiconductors.

The I^{2}C Bus has two lines that carry its signals— one line is for clock and one is for bidirectional data. There is a protocol for I^{2}C bus. Figure 3.10(a) shows the signals during a transfer of a byte when using I^{2}C bus.

Each device has an address using which the data transfers take place. The master can address 127 other slaves at an instance. It has a processing element functioning as a bus controller or a microcontroller with I^{2}C bus interface circuit. Each slave can also optionally have an I^{2}C bus controller and processing element. A number of masters can also connect to the bus. However, at any instance, there can be only one master, which is one that initiates a data transfer on SDA (serial data) line and which transmits the SCL (serial clock) pulses. From the *master* or *slave*, a data frame has fields beginning from start bit as per Table 3.9. Figure 3.10(b) shows the format of the bits at the I^{2}C bus.

Table 3.9

Field and its length	*Explanation*
First field of 1-bit	It is start bit similar to the one in a UART.
Second field of 7 bits	It is called the address field. It defines the slave address being sent the data frame (of many bytes) by the master.
Third field of 1 control bit	It defines whether a read or write cycle is in progress.
Fourth field of 1 control bit	Next bit defines whether the present data is an acknowledgement (from the slave)
Fifth field of 8 bits	It is used for IC device data bits.
Sixth field of 1-bit	It is a negative acknowledgement bit (NACK) from the master. If active, then acknowledgement after a transfer is not needed from the slave, else acknowledgement is expected from the slave.
Seventh field of 1-bit	It is a stop bit like in a UART.

The disadvantage of this bus is the time taken by algorithm in the master hardware that analyses the bits through I^{2}C in case the slave hardware does not provide for the hardware that supports it. Some ICs support the protocol and some do not. In that case, interface circuits for those ICs are also required. Also, there are open collector drivers at the master. Therefore, a pull-up resistance of 2.2 K or an active circuit for pull up of line to logic 1 for on each line is essential.

I^{2}C is a serial bus for interconnecting ICs. It has a start bit and a stop bit like in a UART. It has seven fields for the start, 7-bit address, defining a read or write, defining a byte as an acknowledging byte, data byte, NACK and end.

3.10.2 CAN Bus

Number of devices and controllers are located and are distributed in a car. An automobile uses number of distributed embedded controllers, including those for the brakes, engine, electric power, lamps, inside temperature control, air-conditioning, gate, front dash board display, meter display panel and cruising control. Embedded controllers must network through a bus. CAN (controller area network) bus is a standard bus in distributed network. It is mainly used in automotive electronics. It is also used medical electronics and industrial plant controllers.

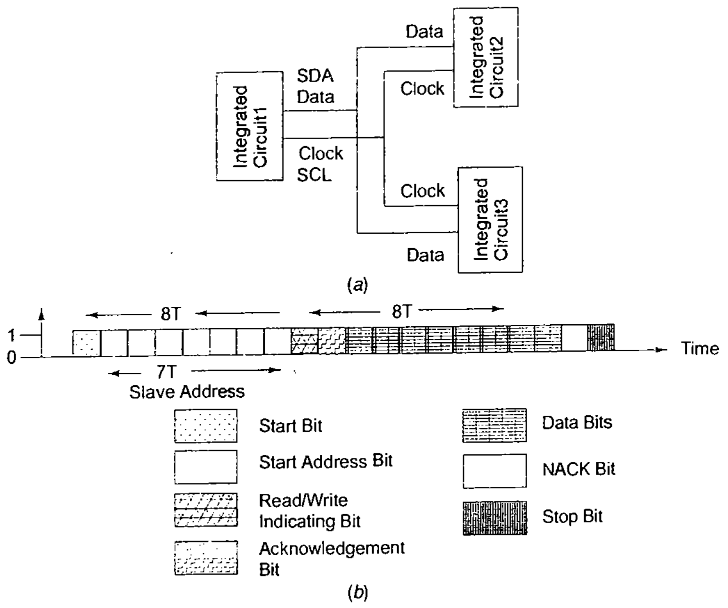


Fig. 3.10 (*a*) Signals during a transfer of a byte when using the I^{2}C (Inter Integrated Circuit) bus (*b*) Format of SDA bits at the I^{2}C bus

The CAN Bus [Figure 3.11(a)] network has a serial line, which is bi-directional. CAN bus has multimaster and multicast features. A CAN device using CAN controller receives or sends a bit at any instance by operating at the maximum rate of 1 Mbps (bit-period = 1 μs). It employs a twisted pair connection of 120 ohm line impedence at each controller node. The pair can run up to a maximum length of 40 m.

1. CAN serial line is pulled to logic level 1 by a resistor (active or passive) between the line and +4.5 V to +12.3 V. Line is at logic 1 in its idle state, also called the recessive state.
2. Each node has a buffer-gate between an input pin and a CAN serial line. A node gets the input at any instance from the line after sensing that instant when the line is pulled down to 0. The latter is called dominant state.
3. Each node has a current driver circuit between output pin and serial line. The node sends a bit to line by pulling the line 0 by its driver for a bit period. An NPN transistor is used current-driving transistor, the emitter of which also connects to the line ground and collector connects to the line. Using a driver (consisting of a buffer inverter gate connected to base of the NPN transistor), the node can pull the line to 0, which is otherwise at logic 1 in its idle state. This lets other nodes sense the input.
4. A node sends the data bits as a data frame. Data frames always start with 1 and always end with seven 0s. Between two data frames, there are minimum three fields. Table 3.10 gives the details of each field in a CAN frame. Figure 3.11(b) shows the format of the bits in a CAN frame.
5. The CAN-bus line usually interconnects to a CAN controller between the line and host node. [A host node is one that has controller for use as bus master.] The line gives input and gets output during reception and transmission using physical and data link layers at host node. The CAN controller has a BIU (bus interface unit consisting of buffer and driver), protocol controller, status-cum-control registers, receiver-buffer and message objects. These units connect the host node through host interface circuit.

6. There is an arbitration method called CSMA/AMP (Carrier Sense Multiple Access with Arbitration on Message Priority). A node stops transmitting on sensing a dominant bit, which indicates that another node is transmitting.

Table 3.10 Each field in a CAN frame

Field and its length	*Function*
First field of 12 bits	This is arbitration field, which contains the packet's 11-bit destination address and RTR bit (Packet means a set of bits sent on the bus). RTR stands for 'Remote Transmission Request'. The receiving addressed device is at destination address specified in 11-bit subfield and RTR is defined on the basis of whether the data byte being sent is a data for the device or a request to the device. 11-bit address identifies the device to which data is being sent or the request being made. When an RTR bit is at 1, it means this packet is for the device at destination address. If this bit is at 0 (dominant state) it means this packet is a request for the data from the device.
Second field of 6 bits	It is control field. The first bit is identifier extension. The second bit is always 1. The last 4 bits are code for data length.
Third field of 0 to 64 bits	Its length depends on the data length code in control field.
Fourth field (third if data field has no bit present) is of 16 bits	It is CRC (Cyclic Redundancy Check) field with 15-bit CRC plus 1-bit delimiter bit. The receiver node uses it to detect errors, if any, during the transmission.
Fifth field of 2 bits	First bit is 'ACK slot'. The sender sends it as 1 and the receiver, which would send back 0 in this slot when it detects error in reception. The sender, after sensing 0 in the ACK slot, retransmits the data frame. The second bit is the 'ACK delimiter' bit. It signals the end of ACK field. If the transmitting node does not receive any acknowledgement of data frame within a specified time slot, it should retransmit.
Sixth field of 7 bits	This is the end-of-the-frame specification and has seven 0s.

CAN is a serial bus for interconnecting a central control network. It is widely used in automobiles. It has fields for bus arbitration bits, control bits for address and data length, data bits, CRC check bits, acknowledgement bits and ending bits.

3.10.3 USB Bus

Universal Serial Bus (USB) is a bus between host system and number of interconnected peripheral devices. A maximum 127 devices can connect to a host. It provides a fast (up to 12 Mbps) and as well as a low speed (up to 1.5 Mbps) serial transmission and reception between host and serial devices. A USB host, which includes controller for function as bus master can connect flash memory cards, pen-like memory devices, digital camera, printer, mice, PocketPC and video games. Thre are three standards: USB 1.1(a low speed 1.5 Mbps 3 m channel along with a high speed 12 Mbps, 25 m channel); USB 2.0 (high speed 480 Mbps 25 meter channel), and wireless USB (high speed 480 Mbps 3 m).

USB protocol has this feature—a USB device can be hot plugged (attached), configured and used, reset, reconfigured and used; it can share bandwidth with other devices, detached (while others are in operation)

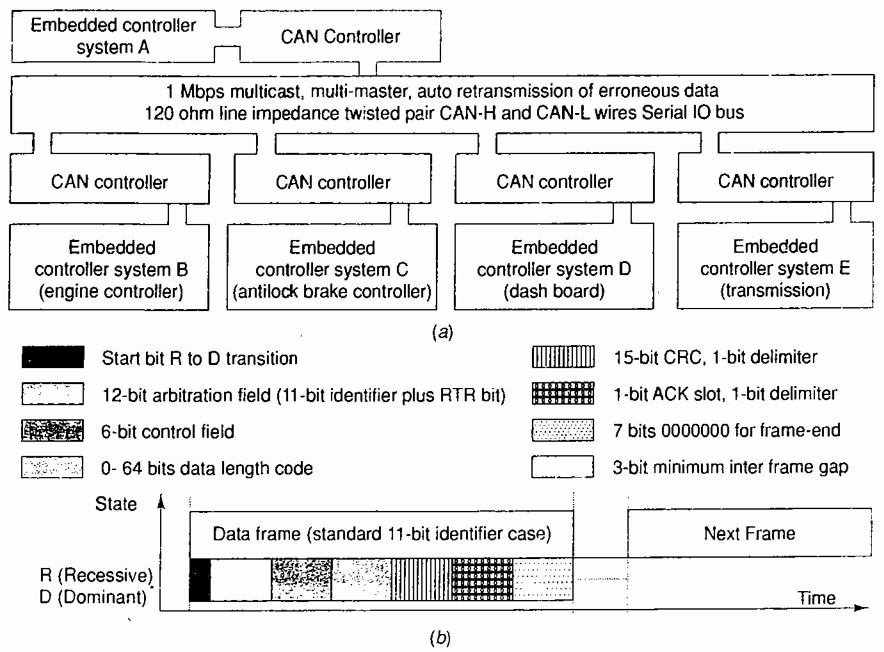

Embedded controller system A
CAN Controller
1 Mbps multicast, multi-master, auto retransmission of erroneous data
120 ohm line impedance twisted pair CAN-H and CAN-L wires Serial IO bus
CAN controller
CAN controller
CAN controller
CAN controller
Embedded controller system B (engine controller)
Embedded controller system C (antilock brake controller)
Embedded controller system D (dash board)
Embedded controller system E (transmission)
(a)
Start bit R to D transition
12-bit arbitration field (11-bit identifier plus RTR bit)
6-bit control field
0- 64 bits data length code
15-bit CRC, 1-bit delimiter
1-bit ACK slot, 1-bit delimiter
7 bits 0000000 for frame-end
3-bit minimum inter frame gap
State
Data frame (standard 11-bit identifier case)
Next Frame
R (Recessive)
D (Dominant)
Time
(b)

Fig. 3.11 (*a*) Network during a transfer of data when using the CAN (Controller Area Network) bus (*b*) Format of the bits at CAN bus

and reattached. Attaching and detaching can be done without rebooting. The host schedules sharing of bandwidth among the attached devices. A USB device can either be bus-powered or self-powered. In addition, there is a power management by software at host for USB ports.

USB host connects to devices or nodes using USB port-drving software and the host controller connected to a root hub. A hub is one that connects to other nodes or hubs. A tree-like topology forms as follows. The root hub connects to the hub and node at level 1. A hub at level 1 connects to the hub and node at level 2 and so on. Only the nodes are present at the last level. The root hub and each hub at a level connect in a star topology with the next level. The USB device descriptor data structure has a hierarchy, which is as follows: It has device descriptor at the root that has number of configuration descriptors and each configuration descriptor has number of interface descriptors and which has number of end point descriptors.

USB bus cable has four wires, one for +5 V, two for twisted pairs and one for ground. There are termination impedances at each end that are as per the device speed. Electromagnetic Interference (EMI)-shielded cable is used for 15 Mbps USB devices.

Serial signals are Non Return to Zero ((NRZI) and the clock is encoded by inserting a synchronous code (SYNC) field before each packet. [Refer to Table 3.2]. The receiver synchronizes its bit recovery clock continuously. The data transfer is of four types: (a) Controlled data transfer (b) Bulk data transfer (c) Interrupt driven data transfer (d) Isosynchronous transfer.

USB is a polled bus. The host controller circuit regularly polls the presence of a device as scheduled by the software. It sends a token packet. The token consists of fields for type, direction, USB device address and

device end-point number. The device does the handshaking through a handshake packet, indicating successful or unsuccessful transmission. A CRC field in a data packet enables transmission error detection at the receiver.

USB supports three types of pipes—(a) 'Stream' with no USB-defined protocol. It is used when the connection is already established and the data flow starts. (b) 'Default Control' for providing access. (c) 'Message' for the control functions of the device. The host configures each pipe for the followings: (a) data bandwidth to be used, (b) transfer service type and (c) buffer sizes.

Wireless USB is wireless extension of USB 2.0 and it operates at UWB (ultra wide band) 3.1 to 10.6 GHz frequencies. It is used for short-range personal area network (high speed 480 Mbps, 3 m or 110 Mbps, 10 m channel). FCC has recommended a host wire adapter (HWA) and a device wire adapter (DWA), which provide wireless USB solutions. Wireless USB also supports dual-role devices (DRDs). A device can be a USB device as well as a limited capability host. For example, a wireless USB digital camera uses a USB host when connected to a printer and a USB device when connected to a personal computer. A wireless USB device is used to provide Internet connectivity between laptop or computer and mobile service provider network.

USB is a serial bus that interconnects a system. It attaches and detaches a device from the network. It uses a root hub. Nodes containing the devices can be organized like a tree structure. It is mostly used in networking the IO devices like scanner in a computer system. Wireless USB is used for remote connections without wires.

3.10.4 FireWire — IEEE 1394 Bus Standard

Digital video cameras, digital camcorders, digital video disk (DVD), set-top boxes, and music systems multimedia peripherals, latest hard disk drives, and printers need a high-speed bus standard interface for communicating directly to a personal computer. FireWire (IEEE 1394b) is a standard for 800 Mbps serial isosynchronous data transfers.

A FireWire IEEE 1394 port can operate at up to 400 Mbps and the latest machines include FireWire ports that support IEEE 1394b which operate at up to 800 Mbps. Since FireWire can transfer data at a guaranteed rate, it is also used in real time devices, such as video device data transfers.

A single 1394 port can interface up to 63 external FireWire devices. It supports both plug and play and hot plugging. It also provides self-powered and bus-powered support on the bus.

FireWire is a high speed 800 Mbps serial bus for interconnecting a system with multimedia streaming devices and systems.

3.10.5 Advanced Serial High Speed Buses

Section 3.2.6 described SDIO, which is an advanced high-speed serial bus for handheld devices. An embedded system may need to connect multi gigabits per second (Gbps) transceiver (transmit and receive) serial interfaces. Exemplary products are wireless LAN, Gigabit Ethernet, SONET (OC-48, OC-192, OC-768). The following are examples of the advanced bus protocols.

1. IEEE 802.3-2000 [1 Gbps bandwidth Gigabit Ethernet MAC (Media Access Control)] for 125 MHz performance
2. IEE P802.3oe draft 4.1 [10 Gbps Ethernet MAC] for 156.25 MHz dual direction performance]
3. IEE P802.3oe draft 4.1 [12.5 Gbps Ethernet MAC] for four channel 3.125 Gbps per channel transceiver performance]
4. XAUI (10 Gigabit Attachment Unit)
5. XSBI (10 Gigabit Serial Bus Interchange)
6. SONET OC-48
7. SONET OC-192
8. SONET OC-768
9. ATM OC-12/46/192

3.11 PARALLEL BUS DEVICE PROTOCOLS—PARALLEL COMMUNICATION NETWORK USING ISA, PCI, PCI-X AND ADVANCED BUSES

A computer system connects at high speed to other subsystems having a range of IO devices at very short distances (<25 cm) using a parallel bus without having to implement a specific interface for each IO device. When the IO devices in the distributed embedded subsystems are networked, all can communicate through a common parallel bus. A parallel bus has a large number of lines as per the protocol. Figures 3.12(a) and (b) show the processor of an embedded system A connected to system memory bus and networked to other subsystems through a parallel bus PCI using PCI bridge and AMBA-APB bridge, respectively.

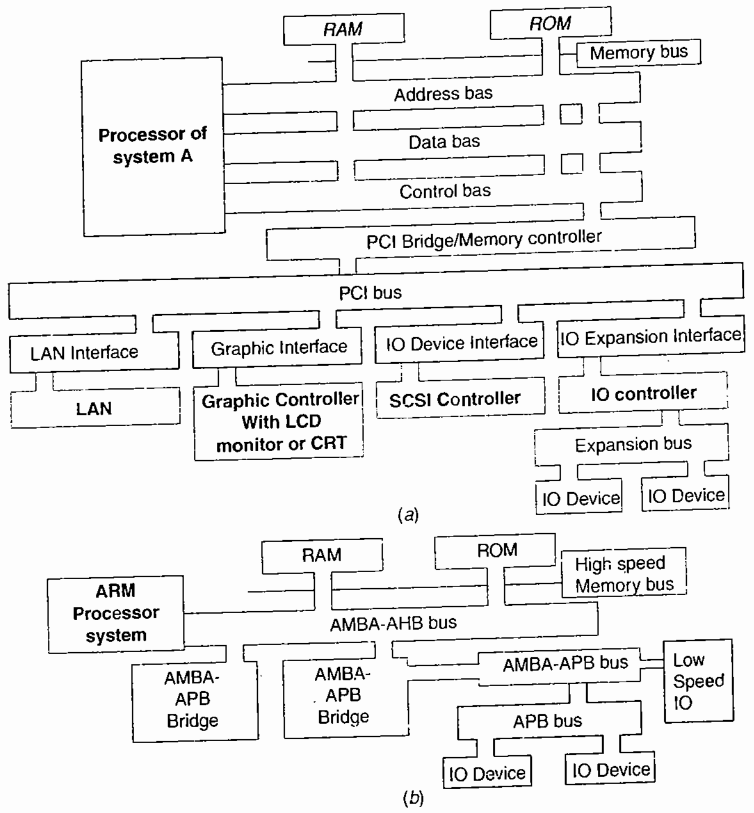


Fig. 3.12 (*a*) and (*b*) A processor of embedded system connected to system memory bus and networked to other subsystems through a parallel bus using PCI and AMBA-APB bridges

We need an interconnection bus within PC or embedded system to a number of PC-based IO cards, systems and devices. This bus needs to be separated from system-bus that connects the processor to memories. The

system bus and interconnection bus operate at different levels of speeds. Exemplary devices are display monitor, printer, character devices, network subsystems, video card, modem card, hard disk controller, thin client, digital video capture card, streaming displays, 10/100 Base T card and card using DEC 21040 PCI Ethernet LAN controller. Each of these devices, which performs a specific function, may contain a processor and drives by software. Each device has specific memory address-range, specific interrupt-vectors (pre-assigned or auto configured) and device IO port addresses. A bus of appropriate specifications and protocol interfaces these to host system or computer.

A switch, popularly called PCI bus interface, switches a processor communication with the memory bus to PCI bus. In most systems, the processor has a single data bus that connects to a switch module such as the PCI bridge found in many PC systems, although some processors integrate the switch module onto the same integrated circuit as the processor to reduce the number of chips required to build a system thereby reducing the system cost. The switch communicates with the memory through *memory bus* and dedicated set of wires that transfer data between these systems. A separate *IO bus* connects the switch to IO devices. Separate memory and IO buses are used because the IO system is generally designed for maximum flexibility, to allow as many different IO devices as possible to interface to the computer, while the memory bus is designed to provide the maximum possible bandwidth between the processor and memory system.

Two old interconnection buses for communication between the host and a device are ISA and EISA (Extended ISA). A new interconnection for the bus is either PCI or PCI/X. [A variant of it is Compact PCI (cPCI).] Sections 3.12.1 to 3.12.4 describe three parallel bus communication protocols.

Parallel bus interconnects IO devices and peripherals over very short distances and at high speed. ISA, PCI and ARM buses are examples of parallel buses. A parallel bus interfaces the system memory bus through a bridge or switching circuit.

3.11.1 ISA Bus

ISA bus (used in IBM Standard Architecture) connects only to an embedded device that has an 8086 or 80186 or 80286 processor, and in which the processor addressing and IBM PC architecture addressing limitations and interrupt vector address assignments are taken into account. There is no geographical addressing.

The limitation for memory access by a system using the ISA bus of the original IBM PC were as follows: ISA bus memory accesses can be in two ranges, 640 to 1 MB and 15 to 16 MB. The former range also overlaps with the range used by video boards and BIOS. [*Note*: Linux OS does not support the second range for accessing directly a device.]

The IO port address limitations for devices are as follows: The 8086 to 80286 processor has IO mapped IOs, not memory mapped IOs. Though the instruction set provides for IO instructions for 64 kB IO addresses, the IBM PC configuration ignores the address lines A_{10} to A_{15} and these are not decoded. Therefore, only 1024 IO port addresses are available. A hexadecimal addressing scheme with three nibble addressing between 000 to 3FF only can be used for a device. The A_{10} to A_{15} bits are thus immaterial. The following are the addresses allocated in IBM Standard Architecture (ISA).

1. Addresses allocated are 0x000–0x00F for DMA chip 8237. The addresses for other devices are as follows.
2. 0x020–0x021 addresses allocated are for programmable interrupt controller 8255. Hex 0x040–0x043 for timer 8253.
3. 0x060–0x063 for parallel port programmable parallel interface.
4. The hexa-decimal addresses 080-083, 0A0-0AF, 0C0-0CF, 0E0-0EF allocated are for components on the motherboard.

5. Reserved addresses from peripherals are hex 220-24F, 278-27F, 2F0-2F7, 3C0-3CF and 3E0 to 3F0.
6. The addresses allocated are hex 2F8-2FF and 3F8-3FF for IBM COM ports.
7. Addresses are hex. 320-32F and 3F0-3F7 for hard disk and floppy diskette, respectively.
8. Only 32 addresses between 0x300 to 0x31F are available for prototype card; for example, ADC card.
9. Addresses allocated are between hex 380-389 and 3A0-3A9 for synchronous communication.
10. Synchronous Data Link Control (SDLC) addresses allocated are between hex. 380-38C.
11. Display monitor ports are within 380-38F (monochrome) and 3D0-3DF for (colour and graphics).

There is a limited availability of interrupt vectors in the IBM PC 80x86 family. Only 256 vectors are available. Interrupt service functions are now shared at software level: for example, SWT interrupts. Original ISA specifications did not allow that.

EISA bus is a 32-bit data and address-lines version of ISA, and devices (system using this bus for IOs) are also supported. An EISA device driver first checks the EISA bus availability on the hosting computer or system. It supports the sharing of interrupt functions, SCI (Serial Communication Interface) controller and Ethernet devices. Unix and Linux support the EISA bus-driven cards and devices.

ISA and EISA buses are compatible with IBM architecture. They are used for connecting devices following IO addresses and interrupt vectors as per IBM PC architecture. EISA is 32-bit extension of ISA. It also supports software interrupt functions and Ethernet devices.

3.11.2 PCI and PCI/X Buses

Recently, the most used synchronous parallel bus in the computer system for interfacing PC-based devices is PCI (Peripheral Component Interconnect). PCI provides a superior throughput than EISA. It is almost platform-independent, unlike the ISA, which depended on the IBM PC platform, interrupt vectors, IO addresses and memory allocations. Its clock rate is nearest to the submultiple of the system clock. PCI provides three types of synchronous parallel interfaces. Its versions are 32/33 MHz, 64/66 MHz, PCI-X 64/100 MHz, PCI Super V2.3 264/528 MBps 3.3 V (on a 64-bit bus), 132/264 (on a 32-bit bus) and PCI-X Super V1.01a for 800 MBps 64-bit bus 3.3 V.

PCI bus has 32-bit data bus extendible to 64 bits. In addition, it has 32-bit addresses extendible to 64 bits. Its protocol specifies the interaction between the different components of a computer. A specification is version 2.1. Its synchronous/asynchronous throughput is up to 132/ 528 MB/s [33 M x 4/ 66 M x 8 Byte/s]. It operates on 3.3 V signals. A typical application is an exemplary PCI Card has a 16 MB Flash ROM with a router gateway for a LAN.

A PCI driver can access hardware automatically as well as by addresses assigned by the programmer. The PCI feature of automatically detecting the interfacing systems and assigning new addresses is important for coding a device driver. The PCI bus therefore simplifies the addition and deletion (attachment and detachment) of the system peripherals. A manufacturer registers a global number for PCI device or card, just as, 68HC11 or 80386 are globally registered numbers. A 16-bit register in PCI device identifies this number to let that device auto-detected. Another 16-bit register is for a device ID number. These two numbers allow the device to carry out auto-detection by its host computer. Each device may use FIFO controller with FIFO buffer for maximum throughput.

A device or host identifies its address space by three identification numbers (i) IO port, (ii) memory locations and (iii) configuration registers of total 256 B with a 4-byte unique ID. Each PCI device has address space allocation of 256 bytes to access it by the host computer. The unique feature of PCI bus is its configuration address space. A uniquely assigned interrupt type (a number) handles an interrupt. For example, interrupt type 3 has the interrupt vector address 0x0000C and four bytes at the address specify the interrupt service

routine address. Interrupt type can be between 0x00 and 0xFF. A configuration register number 60 stores the one byte for the interrupt type n(pci). The PCI device or host when interrupted handles the interrupt of type n(pci). Figure 3.13 shows 64-byte standard configuration registers in a PCI device. Following are the abbreviations used in the figure.

VID: Vendor ID. *DID*: Device ID. *RID*: Revision ID. *CR*: Common Register. *CC*: Class Code. *SR*: Status Register. *CL*: Cache Line. *LT*: Latency Timer. *BIST*: Base Input Tick. *HT*: Header Type. *BA*: Base Address. *CBCISB*: Card Base CIS Pointer. *SS*: Sub System. *ExpROM*: Expansion ROM. *MIN_GNT*: Minimum Guaranteed time. *MAX_GNT*: Maximum Guaranteed Time.

VID, *DID*, *RID*, *CR*, *SR*, and *HT* are compulsorily configured. The rest are optional.

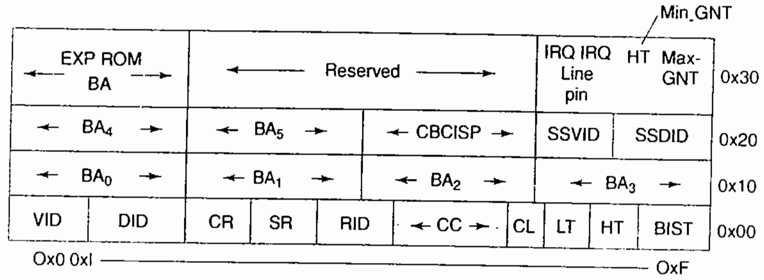


Fig. 3.13 64 bytes at standard device independent configuration registers in a PCI device or host

A PCI controller must access one device at a time. Thus, all the devices within host computer can share IO port addresses and memory locations but cannot share the configuration registers. That means that a device cannot modify other configuration registers but can access other device resources or share the work or assist the other device. If there are reasons for it doing so, a PCI driver can change the default bootup assignments on configuration transactions.

A device can initialize at booting time. This helps in avoiding any address collision. A PCI device on bootup disables its interrupt. Its address space is inaccessible and only the configuration registers space remains accessible. PCI BIOS with the device performs the configuration transactions and then memory and address spaces automatically map to the address space in host computer.

PCI parallel bus is popular in distributed embedded devices. PCI and PCI/X buses are used for parallel bus communication and these are independent from the IBM architecture. PCI/X is an extension of PCI and supports 64/100 MHz transfers. PCI bus new version support 132/528 MB/s data transfer with synchronous/ asynchronous throughputs.

3.11.3 ARM Bus

ARM processor interfaces the memory, external DRAM (dynamic RAM controller and on-chip IO devices, which connect to 32-bit data and 32-bit address line at high speed using AMBA (ARM Main Memory Bus Architecture)-AHB (ARM High Performance Bus). Figure 3.12(b) shows AMBA-AHB and AMBA-APB bridges. The bridges interface the memory and external-chip IO devices, which operate at low speed using AMBA-APB. The maximum AHB bps bandwidth is sixteen times the ARM processor clock.

A switch, popularly called the AMBA-APB bridge, switches ARM CPU communication with the AMBA bus to APB bus. The ARM processor-based microcontroller has a single data bus in AMBA-AHB that connects to the bridge, which integrates the bridge onto the same integrated circuit as the processor to reduce the number of chips required to build a system. This reduces the system cost. The bridge communicates with the memory through an AMBA-AHB, a dedicated set of wires that transfer data between these two systems. A separate *APB IO bus* connects the bridge to the IO devices. Separate AMBA-AHB and APB IO buses are used because the IO system is generally designed for maximum flexibility, to allow as many different IO devices as possible to interface to the computer, while the memory bus is designed to provide the maximum possible bandwidth between the processor and the memory system.

The APB can connect the I^{2}C, touchscreen, SDIO, MMC (multimedia card), USB, CAN and other required interfaces to an ARM microcontroller.

ARM bus is of two types: AMBA-AHB and AMBA-APB. AHB connects to high speed memory. APB connects the external peripherals to the system memory bus through a bridge.

3.11.4 Advanced Parallel High Speed Buses

Many telecommunication, computer and embedded processor-based products need parallel buses for system IOs. Three versions of PCI parallel synchronous/asynchronous buses provide system-synchronous parallel interfaces. These three versions may not have sufficiently high speed, ultra high speed and large bandwidth that are required for system IOs, routers, LANs, switches and gateways, SANs (Storage Area Networks), WANs (Wide Area Networks) and other products. These do not meet the source-synchronous parallel interfacing requirements. Bandwidth needs increase exponentially in the order of audio, graphics, video, interactive video and broadband IPv6 Internet. An embedded system may need to connect IO system using gigabit parallel synchronous interfaces. The following are advanced bus standard and proprietary protocols developed recently.

1. GMII (Gigabit Ethernet MAC Interchange Interface).
2. XGMI (10 Gigabit Ethernet MAC Interchange Interface)
3. CSIX-1. 6.6 Gbps 32-bit HSTL with 200 MHz performance.
4. RapidIO™ Interconnect Specification v1.1 at 8 Gbps with 500 MBps performance or 250 MHz dual direction registering performance using 8-bit LVDS (Low Voltage Data Bus).

3.12 INTERNET ENABLED SYSTEMS—NETWORK PROTOCOLS

Figure 3.14 shows an Internet-enabled embedded system communicating to other systems on the Internet. Internet-enabled embedded systems use html or MIME type files (Section 3.12.1), TCP (Section 3.12.2) or UDP (Section 3.12.3) transport layer protocol, and are addressed by an IP address (Section 3.12.4) and use IP protocol at network layer. An IP address is of 32 bits (four decimal numbers seperated by dots in between) or 48 bits in IPv4 or IPv6 respectively. IPv4 means IP protocol version 4 and IPv6 means version 6. A system at one IP address 1 communicates with another system at another IP address 2 or 3 or... using the physical connections on the Internet and the routers. Since the Internet is a global network, the system connects to remotely located as well as short range located system. Network connectivity is through the layers. Each layer has a protocol, which specifies the way in which the data or message from the previous layer transfers to the next layer.

There are five layers in a TCP/IP network. They are the application, transport, network, data-link and physical layers. The TCP/IP application layer protocol also specifies presentation ways. Transport layer protocol specifies session establishment and termination ways also.

Sections 3.12.1 to 3.12.5 describe the TCP/IP suite's five most used protocols.

Embedded systems are Internet enabled by using TCP/IP protocol suite protocols for networking to Internet and assigning the IP addresses to each system.

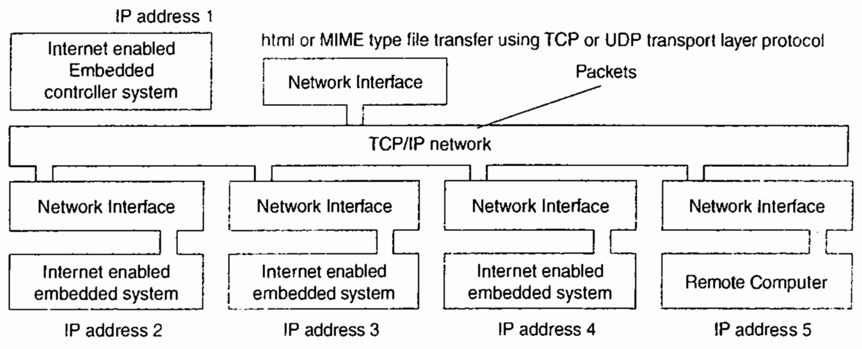

IP address 1
Internet enabled Embedded controller system
html or MIME type file transfer using TCP or UDP transport layer protocol
Network Interface
Packets
TCP/IP network
Network Interface
Network Interface
Network Interface
Network Interface
Internet enabled embedded system
Internet enabled embedded system
Internet enabled embedded system
Remote Computer
IP address 2
IP address 3
IP address 4
IP address 5

Fig. 3.14 An Internet enabled embedded system communication connected on the Internet

3.12.1 Hyper-Text Transfer Protocol (HTTP)

An application layer protocol is as per the application. This layer accepts the data, for example, in HTML or text format and puts the header words as per the protocol and sends the application layer header plus data to the transport layer. A port number specifies the application in the header. Following are the important application layer protocols that support TCP/IP networking.

1. NTP (Network Time Protocol) synchronizes system clocks on a network.
2. MIME enables attachment of multiple types of files. The examples are:
   - txt (text file),
   - doc (MSOFFICE Word document file),
   - gif (graphic image format file),
   - jpg (jpg format image file), and
   - wav voice or music file.
3. HTTP (Port 80) enables Internet connectivity by Hyper-Text Transfer Protocol (HTTP).
4. FTP (Port 21 for control, 20 for data) enables file transfer connectivity by File Transfer Protocol. TFTP (Port 69) for Trivial FTP. NFS (Network File System) is used for sharing files on a network.
5. TELNET (Port 23) enables remote login to remote terminals by Terminal Access Protocol.
6. SMTP (Port 25) enables e-mail transfer, store and forward by Simple Mail Transfer Protocol.
7. PoP3 (Port 110) enables e-mail retrieval.
8. NNTP (Port 119) (Network News Transfer Protocol) is used for news transfer port.
9. DNS (Port 53) for Domain Name Service.
10. SNMP (Port 161) Simple Network Management Protocol.
11. Bootps and Bootpc (Ports 67 and 68) for Bootstrap Protocol (DHCP) Server and Client, respectively.
12. DHCP (Dynamic Host Configuration Protocol) used for remote booting as well as for configuring a system.

A port-assigned number supports multiple logical connections using a socket. Each socket has IP address and port number. A registered port number can be between 0 and 1023. Registration is done by IANA (Internet

Assigned Number Authority. Port number 0 means host itself. A user unregistered server can have Port number above 5000.

HTTP port 80 is an application layer protocol. The HTTP features are as follows:

1. HTTP is standard protocol for requesting for a URL (universal resource locator) address, for example, http://www.mcgraw-hill.com.) An URL defines a web page resource, and is used for retrieving or sending web page file. The response from web may be with or without applying a process. An HTTP client requests an HTTP server on the Internet and the server responds by sending a response.
2. HTTP is a stateless protocol. For an HTTP request, the protocol assumes a fresh request. It means there is no session or sequence number field or no field that is retained in the next exchange. This makes a current exchange by an HTTP request independent of the previous exchanges. The later exchanges do not depend on the current one. An e-commerce-like application needs a state management mechanism. A Cookie is a text file created during a particular pair of exchanges of HTTP request and response. The creation is either at CGI or processing program or script or at client (Browser). A prior exchange may then depend on this cookie. By this mechanism, the stateless feature of HTTP is compensated. The cookie provides a HTTP state management mechanism.
3. HTTP is a file transfer-like protocol for HTML (hyper text markup language) files. This makes it easy to explore a web site URL. A request (from a client) is sent and reply (response from a server) is received.
4. The HTTP protocol is very light (a small format) and thus speedy as compared to other existing protocols. HTTP is able to transfer any type of data to a browser (a client) provided it is capable of handling that data.
5. Besides simplicity, another important feature of HTTP is its flexibility. Assume we are surfing the web and our connection breaks (or user does so); then too we can start surfing on the Net from just that point again. Being a stateless protocol, HTTP does not keep track of the state as FTP does. Each time a connection establishes between the web server and the client (browser), both these interpret this connection as a new connection. Simplicity is a must because a web page has its URL resources distributed over a number of servers.
6. HTTP protocol is based on the Object Oriented Programming System (OOPS). Methods are applied to objects identified by URL. It means that as in the normal case of Object Oriented Program, the various methods apply on an object.
7. From HTTP 1.0 and 1.1 version onwards, the following features have been included:
   (a) Multimedia file access is feasible due to provision or the MIME (Multipurpose Internet Mail Extension) type file definition.
   (b) From HTTP 1.1 version onwards, there are eight specified methods and extension methods. An extension method is method added for a specific HTTP. There can be none or one or several extension methods. The HTTP specific methods are as follows. 1. GET 2. POST. 3. HEAD 4. CONNECT. 5. PUT 6. DELETE 7. TRACE. 8. OPTIONS. (Last four from 1.1). In earlier versions, GET follows a space and then document name. Server returns the documents and closes the connection. From 1.1, the POST method has permitted form processing, as using it the client transmits the form data or other information to the server. From 1.1, the server does not close the connection after response and thus response can be processed before it is sent.
   (c) A provision of user authentication exists besides the basic authentication introduced from HTTP 1.1 version onwards, Digest Access Authentication prevents the transmission of username and password as HTML or text.
   (d) A host header field adds to support those ports and virtual hosts that do not accept or send IP packets. From HTTP 1.1 version onwards, An error report to client when a HTTP request is without a host header field.

   (e) From HTTP 1.1 version onwards, an absolute URL is acceptable to the server. Earlier only proxy server accepted that.
   (f) Status codes in the response.
   (g) Caching of a resource is provided at server (and proxy).
   (h) Byte range specification helps in large response in parts.
   (i) Selection among various characteristics on retrieval by the client is feasible when a server sends *response* to client *request*. For example, two characteristics, *language* and *encoding* can be specified in server environment variables while the client sends request header for retrieving a resource. The resource then retrieves in that *language* and with that *encoding*. The contents sent to client do not change, only the way in which these are presented to the client change.
   (j) Length specification helps in presentation in chunks.
8. An HTTP message header during a request from a client or during a response from server consists of two parts (a) A start-line, none or several message- headers (fields) and empty line, and (b) Body of message. HTTP specifies that request message to consist of request message headers. HTTP also specifies that response message to consist of response message headers.
9. HTTP provides for entity headers. These contain information about entity body contained in the message, or in case body is not present then information about the entity, not its body. For example, information of content length in bytes.
10. HTTP interaction scheme is that a client requests server directly or through proxy or a gateway. An HTTP message is therefore either *request* or *response*. The format of the messages called RFC 822, specifies ways of sending text messages on the Internet. The message during request from client or during response from server consists of two parts, (a) Start-line, none or several message-headers (fields) plus empty line, and (b) Body of the message. The start line is either a 'request-line' or 'status-line' for request-or response-message, respectively.

3.12.2 Transport Control Protocol (TCP)

TCP (Transport Control Protocol) is a protocol used in transport layer. This layer accepts messages from the upper layer on transmission by application or session layer. This layer also accepts a data stream from the network layer at receiving end. Before communicating a message to the next network layer, it may add a *header*. The message may communicate in parts or segments or fragments. The header generally has the additional bits for source and destination addresses. Also, there are bits in it for the sequence and acknowledge management, flow and error controls, etc.

There are bits for the offset, window, flags, checksum, urgent pointer, option and for padding also. TCP supports the point to point networking mode.

TCP specifies a format of byte streams at the transport layer of the TCP/IP suite. TCP is used for a full duplex acknowledged flow. Its format has a TCP header of five plus (n–5) words for options and padding and data of maximum l words. Then, $l \geq 2^{14} - n$. Here $n \geq 5$. n equals the number of words in the header and is called data offset, which means the number of words after which data bits start in the stream. If $n > 5$, it means there exists words for options and padding. Padding refers to bits used for filling the remaining part of the available field. For example, the option field may indicate the application to be run by the destined node. An acknowledged flow means that the messages communicate in a point-to-point network mode and that there is an acknowledgment for first establishing a connection. Full duplex means that at a given instance, messages go to and fro from sender to receiver, and that the receiver acknowledges receipt. A request and its response do not form a separate transmission. TCP is virtual-connection oriented. It does not permit multicasting but point-to-point virtual connection.

3.12.3 User Datagram Protocol (UDP)

TCP/IP also supports at the transport layer a simpler protocol than TCP. When a message is connectionless and stateless, then the transport layer protocol in the TCP/IP suite is User Datagram Protocol (UDP). UDP supports the broadcast networking mode. An example is application for communicating header before a data stream. The header specifies the bits for source and destination ports, total length of message including header and check sum (optional). During reception, this message to upper layer flows after deleting the header bits from the received transport layer header. Header bits add at the transmitting time in the application or session layer bytes.

3.12.4 Internet Protocol (IP)

All Internet enabled devices communicate using Internet protocol (IP). The transport layer data transmits on the network, divides into the packets at the network layer. Each packet transmits through a chain of routers on the Internet. A acket is minimum unit of data that transmits on the Internet through routers. Several pacets forming a source can reach a destination using different routes and can have different delays. The packet consists of IP header plus data or IP header plus routing protocol along with the routing messages. The packet has a maximum of 2^{16} bytes (2^{14} words, 1 word = 32 bits = 4 bytes).

Network routing is as per standard IPv4 (version 4) or IPv6 (version 6). IPv6 is a broadband protocol. Table 3.11 lists the fields in IPv4 protocol header.

Table 3.11 Various fields at IPv4 header for routing the packets through routers to destination node

Field at the IP header	*Explanation*
Version	IP version bits are 0100 for IPv4 (presently in wide use) and 0110 for IPv6 (IPng IP next genration for broadband Internet).
Precedance	Precedence type is between 8th to 10th bit. Bits 111 specifies highest precedence. For example, for streaming audio or video, 000 specifies common data.
Service	Service type is between 11th to 15th bit.
QoS (Quality of Service)	Bits are for QoS (Quality of Service) specification in terms of security, speed, delays and cost desired or must be achieved.
Fragment ID	Each message may have many packets fragmented through the routers. Each fragment must thus provide a unique ID for identification for re-assembly at the receiver end.
Flags	Flags indicate whether present fragment is last one, whether fragments are permitted and whether more fragments will follow. Let q = Number of header words (1 word = 32-bit). Flag bit 1 indicates whether more fragments of the packet will succeed this fragment. Flag bit 2 identifies it as a test fragment or not. Flag bit 3 checks whether the fragmentation is permitted or not.
Checksum	Header checksum checks errors in header transmission.
Time-to-live	Time to live indicates number of retransmission hops permitted in case of failed delivery.
Protocol type	Type indicates whether the packet is transmitting a UDP byte stream or a TCP stream from transport layer. The important routing protocols that encapsulate after the IP header in an IP packet are:
	IGMP (Internet Group Management Protocol). IGMP is a protocol to manage data transmission between select host groups. Several hosts join a group. Group multicasts use routers that uses the IGMP.
	ICMP (Internet Control Management Protocol). ICMP is a protocol to control routing between networked hosts.
	ICMP data byte stream is inside an IP packet (datagram). Its format is as follows. First 20 bytes minimum are the IP header. Next follows the fields of 1-byte each for Type and Code, successively. Next two bytes are for Checksum. Then follows the ICMP messages the format and length of which is variable.
	Interior routing protocols, for example, the RIP (Routing Information Protocol) and OSPF (Open Shortest Path First) protocol.
	Inter-Domain (exterior routing) protocols, for example, EGP (Exterior Gateway Protocol), BGP (Border Gateway Protocol) and GGP (Gateway to Gateway Protocol).
Header length (data-offset)	The IP header length (data offset) p ≤ 2^{14} – q. Here q ≥ 5. q equals the number of words in the header and is called the data offset [Number of words after which data bits start in the stream.]
Source and Destination addresses options	IP Source and destination IP addresses If q >5, there exist words for options. and padding. Padding refers to bits that are used for filling the remaining part of the available field. For example, option 4 will mean put time stamp at all the stoppages of the packet during transit to destination through routers. Time stamping enables packet delay measurements to calculate Network Performance Quality.

(Contd)

3.12.5 Ethernet

The inventor of Ethernet LAN is Robert Metcalfe. At present, about one third of the LANs in the world are the Ethernet LANs, and in each frame, there is a header like in a packet. In Ethernet LAN standard is IEEE 802.2 (ISO 8802.2). It is a protocol for local network of computers, workstation and devices. LAN is used for sharing local computers, systems and local resources such as printers, hard disk space, software and data. Table 3.12 gives the features of the Ehernet LAN devices.

Data for transmission fragments into the frames. Each frame has a header. Firstly, the header has eight bytes, which defines a preamble. The preamble indicates the start of a frame and is used for synchronization. Then the header has six bytes of destination address. Six bytes of source address then follows the destination address. Then there are six bytes. These are for the type field. These are meaningful only for the higher network layers and the length definition. The minimum 72 bytes and maximum 1500 bytes of data follow the length definition. Lastly, there are 4 bytes for CRC check for the frame sequence check.

3.13 WIRELESS AND MOBILE SYSTEM PROTOCOLS

Figures 3.15(a), (b) and (c) show a handheld device or computer system connected to other handheld devices or computer through IrDA, Bluetooth and ZigBee wireless protocols, respectively. Sections 3.13.1 to 3.13.4 describe the IrDA, Bluetooth, WLAN (wireless LAN) 802.11 and ZigBee protocols.

Table 3.12 Ethernet LAN features

Feature	*Ethernet LAN*
Topology and transmission mode	Bus
Speed	10 Mbps, 100 Mbps (unshielded and shielded wires) and 4 Gbps (in twisted pair wiring mode)
Broadcast Medium	Passive. Wired connections-based. Frame format like the IEEE 802.3.
SNMP (Simple Network Management Protocol)	Yes
System	Open (therefore allows equipment of different specifications)
Operation	Each one connected to a common communication channel in the network. It listens and if the channel is idle then transmits. If not idle, waits and tries again. Multi access is like in a packet switched network
Control	Passive, connection-based
Address Resolution Protocol (ARP) for resolving 32-bits Internet protocol addresses with the 48-bit destination host media address.	Yes, There is a media access control (MAC) address for transmitting and forwarding frames on the same LAN. We can also use multicast addressing to send frames to all or few select types of Ethernet devices.
Connectivity to Internet and Intranet	Yes, Outside a LAN the Internet Protocol addresses are sent.

3.13.1 Infrared Data Association (IrDA)

Infrared (IR) is electromagnetic radiation of wavelength greater than visible red light. An infrared source consists of a gallium–arsenic–phosphorus junction-based diode. An infrared receiver consists of a gallium–arsenic–phosphorus junction-based phototransistor, which conducts electric current when the IR beam falls on it and does not conduct when IR does not fall on it. The collector or drain has voltage close to 0 V when it conducts and is close to supply voltage when it does not conduct.

IrDA (Infrared Data Association) recommends a protocol suite as standard. It supports data transfer rates of up to 4 Mbps. It supports bi-directional serial communication over viewing angle between ±15° and distance of nearly 1 m. At 5 m, the IR transfer data can be up to data transfer rates of 75 kbps. There should be no obstructions or wall in between the source and receiver.

Figure 3.15(a) shows a handheld device connected to other computer through using IrDA protocol. Protocol-processing hardware device and the protocol software embeds in the system, which support line of sight communication using infrared.

IrDA supports 5 levels of communication. Level 1 is minimum required communication. Level 2 is access-based communication. Level 3 is index-based communication. Level 4 is synchronized communication. Synchronization software, for example, ActiveSync or HotSync is used. Level 5 is SyncML (synchronization markup language)-based communication. A SyncML protocol is used for device management and synchronization with server and client devices, which are connected by IrDA.

IrDA is used in mobile phones, digital cameras, keyboard, mouse, printers to communicate to laptop computer and for data and pictures download and synchronization. IrDA is also used for control TV, air-conditioning, LCD projector, VCD devices from a distance.

IrDA supports several protocols at three layers. Lower layer is physical layer 1.0 or 1.1. It supports data transfer rates of 9.6 kbps to 115.2 kbps and 115.2 kbps to 4 Mbps in IrDA physical layer 1.0 and 1.1, respectively.

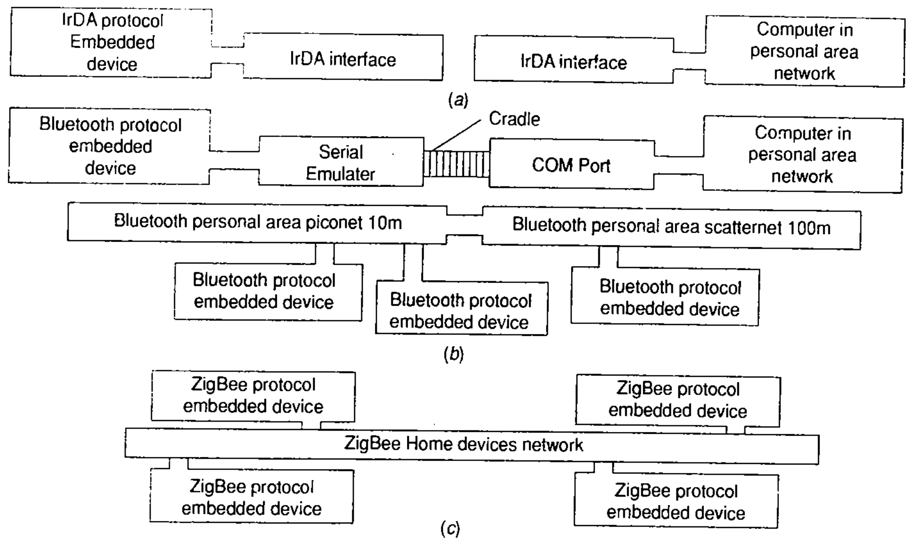

IrDA protocol Embedded device
IrDA interface
IrDA interface
Computer in personal area network
(a)
Bluetooth protocol embedded device
Serial Emulater
Cradle
COM Port
Computer in personal area network
Bluetooth personal area piconet 10m
Bluetooth personal area scatternet 100m
Bluetooth protocol embedded device
Bluetooth protocol embedded device
Bluetooth protocol embedded device
(b)
ZigBee protocol embedded device
ZigBee protocol embedded device
ZigBee Home devices network
ZigBee protocol embedded device
ZigBee protocol embedded device
(c)

Fig. 3.15 (*a*), (*b*) and (*c*) A handheld device connected to other handheld devices or computer through IrDA, Bluetooth and ZigBee wireless protocols, respectively

Intermediate layer is data-link layer. At data link layer, it specifies IrLMP (IR link management protocol) upper sublayer and IrLAP (IR link access protocol) lower sublayer. An IrLAP is HDLC synchronous communication (Section 3.2.4).

An IrDA upper layer protocol is Tiny TP (transport protocol). Another upper layer protocol is IrLMIAS (IR Link Management Information Access Service Protocol). A transport layer protocol during transmission specifies ways of flow control, segmentation of data and packetization. During reception, it assembles the segments and packets. The upper layer protocol for the session layer is IrLAN, IrBus, IrMC, IrTran, IrOBEX (Object Exchange) and standard serial port emulator protocol IrCOMM (IR communication). IrBus provides serial bus access to game ports, joysticks, mice and keyboard. Application layer protocol is for security and application and as specified by the IrDA. For example, IrDA Alliance Sync protocol is used to synchronize mobile devices personal information manager (PIM) data. It supports Object Push (PIM) or Binary File Transfer.

Windows and other operating systems support IrDA protocol-based communication devices. An infrared monitor in Windows monitors the IR port of the IR device. It detects a nearby IR source. It controls, detects and selects the IR communication activity. An IR device on command sets up connection using IrDA, and starts the IR communication. When IR communication is inactive, the monitor enables plug and play (unless disabled).

IrDA protocol overhead is between 2% to 50% of Bluetooth device overhead. The communication setup latency is just few milliseconds. The requirement of line of sight and unobstructed communication is the limitation.

3.13.2 Bluetooth

Bluetooth hardware is connected to embedded system buses and Bluetooth software embeds in the system to support WPAN using Bluetooth wireless protocol. Figure 3.15(b) shows a handheld device connected to other computers through wireless protocol using Bluetooth. A large number of CD players and mobile devices are Bluetooth-enabled. Bluetooth is also used for handsfree listening of Bluetooth-enabled iPod or CD music player or mobile phone by using Bluetooth-enabled era buds.

Bluetooth is an IEEE standard 802.15.1 protocol. The physical layer radio communicates at carrier frequencies in 2.4 GHz band with FHSS (frequency hopping spread spectrum). Hopping interval is 625 μs and number of hopped frequencies are 79. Data transfer is between two devices or between a device and multiple devices.

It supports range up to 10 m low power and up to 100 m high power. Range depends on radio interface at physical layer. Bluetooth 1.x data transfer rate supported is 1 Mbps. Bluetooth 2.0 has enhanced maximum data rate of 3.0 Mbps over 100 m. Bluetooth protocol supports automatic self-discovery and self-organization of network in number of devices. Bluetooth device self discovers nearby devices (<10 m) and they synchronize and form a WPAN (wireless personal area network). Bluetooth protocol supports power control so that the devices communicate at minimum required power level. This prevents drowning of signals by superimpositions of high power signals with lower level signals.

The physical layer has three sublayers: radio, baseband and link manager or host controller interface. There are two types of links: best effort traffic links and real-time voice traffic links. The real-time traffic uses reserved bandwidth. A packet is of about 350 bytes. The link manager sublayer manages the master and slave link. It specifies data encryption and device authentication handling, and formation of device pairs for Bluetooth communication. It gives specifications for state transmission-mode, supervision, power level monitoring, synchronization, and exchange of capability, packet flow latency, peak data rate, average data rate and maximum burst size parameters from lower and higher layers.

The Host Controller Interface (HCI) interface is a hardware abstraction sublayer. It is used in place of the link manager sublayer. It provides for emulation of serial port, for example, 3-wire UART emulation. A Bluetooth device can thus interface to the COM port of a computer.

Its communication latency is 3s. It has large protocol stack overhead of 250 kB. Provision of encrypted secure communication, self-discovery and self-organization and radio-based communication between tiny antennae are three main features of Bluetooth.

3.13.3 802.11

Wireless LAN uses IEEE standards 802.11a to 802.11g. Data transfer rates are 1 and 2 Mbps. The 802.11b is called wireless fidelity (WiFi). 802.11b support data rates of 5.5 Mbps by mapping 4 bits and 11 Mbps mapping 8 bits simultaneously during modulation.

A given set of the LAN-station access-points network together and the set is called extended service set (ESS). It is a backbone distribution system. A backbone set may network through the Internet. ESS supports fixed infrastructure network.

There are two types of wireless service sets.

1. One service set has one wireless station, which communicates to an access point, also called a hotspot. The service set is called basic service set (BSS). WLAN supports ad-hoc network, which, as and when a nodes come nearby in range, it forms the network. BSS supports ad-hoc network which, when nodes come nearby with in range of the access point, forms the network through ESS. A node can move from one BSS to another.
2. The other service set has several stations. It is called independent basic service set (IBSS). It has no access point. It does not connect to the distribution system. It may have multiple stations, which also cannot communicate among themselves. IBSS supports ad-hoc network.

802.11 provides specifications for physical layer and data link layers.

The data link layer specifies a MAC layer. The MAC layer uses carrier sense multiple access and collision avoidance (CSMA/CA) protocol. A station listening to the presence of the carrier during a time interval is called distributed inter-frame spacing (DIFS) interval. If the carrier is not sensed (detected) during DIFS, then the station backs off for a random time interval to avoid collision and retries after that interval. A receiver always acknowledges within a short interframe spacing (SIFS). Acknowledgment is made after successful CRC (cyclic redundancy check). If there is no acknowledgement within SIFS, then the transmitter retransmits and upto 7 retransmission attempts are made.

There is a packet called request to send (RTS), which is first sent. If the other end responses by the packet call clear to send (CTS), then the data is transmitted. MAC layer specifies power management, handover and registration of roaming mobile node within the backbone network at a new BSS within the ESS.

There are three communication methods at the physical layer. WLAN can use FHSS or DSSS or Infrared 250 ns pulses. The physical layer has two sublayers. 802.11b has three sublayers: one is Physical Medium Dependent (PMD) protocol which specifies the modulation and coding methods; the second is the Physical Layer Convergence Protocol (PLCP), which specifies the header and payload for transmission. It specifies the sensing of the carrier at receiver and how packet formation takes place at the transmitter and packets assemble at the receiver. It specifies ways to converge MAC (Medium Access Control) to PMD at transmitter and separate MAC (Medium Access Control) from PMD at the receiver. An additional sublayer in 802.11b specifies Complementary Code Keying (CCK).

3.13.4 ZigBee

ZigBee is an IEEE standard 802.15.4 protocol. The physical layer radio operates at 2.4 GHz band carrier frequencies with DSSS (direct sequence spread spectrum). It supports a range up to 70 m. Data transfer rate supported is 250 kbps. It supports sixteen channels. Figure 3.15(c) shows a handheld device connected to other devices through wireless protocol using ZigBee.

The ZigBee network is self-organizing and supports peer-to-peer and mesh networks. Self-organizing means that it detects nearby ZigBee devices and establishes communication and network. Peer-to-peer network means the each node at network functions as a requesting device as well as a responding device. Mesh network means that each network functions as a mesh. A node can connect to another directly or through mutually interconnected intermediate nodes. Data transfer is between two devices in peer-to-peer or between a device and multiple devices in the mesh network.

ZigBee protocol supports a large number of sensors, lighting devices, air conditioning, industrial controller and other devices for home and office automation and their remote control and formation of WPAN (wireless personal area network). ZigBee network has a ZigBee router, end devices and coordinator. ZigBee router transfers packets from a neighboring source to a nearby node in the path to destination. The coordinator connects one ZigBee network with another, or connects to WLAN or cellular network. ZigBee end devices are transceivers of data.

Its communication latency is 30 ms. Protocol stack overhead is 28 kB.

Summary

Important points dealt with in this chapter are as follows.

- IO ports, IO devices and timing devices are essential in any system.
- An embedded system connects to the devices like keypad, touchscreen, multiline display unit, printer or modem or motors through ports. During a read or write operation, the processor accesses that address in a memory-mapped IO, as if it accesses a memory address. A decoder takes the system memory or IO address bus signals as the input and generates a port or device select signal, CS and selects the port or device.

- There are two types of IO ports and devices, serial and parallel. Serial communication is in synchronous (master-slave) mode or asynchronous mode.
- A device connects and accesses from and to the system-processor through either a parallel or serial IO port. A device port may be full duplex or half-duplex.
- A device or port has an assigned port address using which the processor accesses the device port control register or status register or data. A device can use the handshaking signals before storing the bits at the port buffer or before accepting the bits from the port buffer.
- Serial communication bits are received at the receiver according to the clock phases of the transmitter. Synchronous serial communication bits from the master carry the clock information also to slave. Asynchronous serial communication bits from a device do not carry the clock information to receiver. Receiver clock phase is independent of the transmitter clock. However, the receiver clock adjusts its phase according to the received bits, for example, the start bit.
- HDLC is protocol for a synchronous communication data link network between the devices.
- A popular asynchronous serial communication mode is UART. Bits are received at the receiver independent of the clock phases at the UART (asynchronous serial input and output port) transmitter. UART in microcontrollers usually sends and receives a byte in a 10-bit format or 11-bit format.
- Another popular asynchronous serial communication mode is RS232C, which is based on UART and is used to connect the data communication equipment such as modem with a data terminal equipment such as computer.
- UART and RS232C can also use handshaking signal DCD and a pair of handshaking signals, (DSR, DTR) and (RTS, CTS).
- Other popular serial ports in the devices are SPI, SCI, SI and SDIO.
- Parallel communication is without or with handshaking signals. The number of embedded systems parallel port or device interfaces to switches, keypad, encoders, motors, LCD controllers and touchscreen. Special purpose ports exist at microcontroller for their interfacing. On-chip peripheral devices internally interface with the processor in microcontroller.
- A timer is essentially a counter getting the count-inputs (ticks) at regular time intervals. Timing and counting devices have a large number of uses in a system. There has to be at least one hardware timer in a system. Software timer is a virtual timing device. A program can use number of software timers in a system.
- Internal programmable timing devices with a processor (microcontroller) unit can be used for many applications and to generate interrupts for the ticks for software timers.
- Watchdog timer is special timer which timeouts and generates interrupts in case certain specified event does not occur during the preset interval. The watchdog timer is used to take care of a system stuck in a certain section of a task for an unnecessarily long time due to some error or hardware failure.
- Real-time clock generates ticks and interrupts the system at regular intervals.
- The use of buses simplifies the interfacing to multiple devices. Several devices can be placed on a common serial bus. Popular serial buses are I^{2}C, CAN, USB and FireWire. Each device has an assigned device address or set of addresses. Using the device addresses of the receiver or slave, a master-processor accesses the remote devices.
- I^{2}C bus is used between multiple ICs for inter Integrated Circuit communication. A device, which initiates the communication and sends the clock pulses, is the master at an instance. A master can communicate to maximum 127 slaves.
- The CAN bus is popularly used in centrally controlled network in automobile electronics.
- USB (Universal Serial Bus) is standard for serial bus communication between the system and devices like scanner, keyboard, printer and mouse. There is a root-hub and all nodes have a tree-like structure.
- Several devices can be placed on a common parallel bus. Popular parallel buses are ISA, PCI and ARM buses.
- Very short distance devices interconnect to a PC or embedded system main bus through the ISA or PCI or ARM bus can be used. These buses connect to main memory bus through a bridge (switch).
- Internet-enabled embedded systems network through protocols in a TCP/IP protocol suite. Popularly used protocols are HTTP, TCP, UDP, IP and Ethernet.
- Wireless communication is used for networking handheld devices over wireless personal area network.
- Embedded systems can interconnect and network without wires using IrDA, Bluetooth, 802.11 or ZigBee protocol compatible hardware and software support.

Keywords and their Definitions

Asynchronous Communication	: A communication in which a constant phase difference between the transmitter clock and bit recovery clock at the receiver need not be maintained and the clocks that guide the transmitter and receiver are not synchronized. Time interval between which a set or frame of bytes transmits is not pre-fixed and is indeterminate. Asynchronous communication also provides for exchange of handshaking signals before and during the communication.
Bluetooth	: A self-discovery and self-organizing network protocol for the wireless personal area network and popularly used in mobile handheld devices.
CAN bus	: A standard bus used at the control area network generally in automotive and industrial electronics.
COM port	: A port at the computer where a mice, modem or serial printer or mobile smart phone cradle connects for serial IOs in UART mode and there are handshaking signals for exchange of signals before UART mode communication.
Control register	: A register for bits, which controls or programs the actions of a device. It is used for a *write* operation only.
Control cum Status register	: A register at a port address that saves control and status bits and functions as a control register during write of commands and status register address during read of the status.
Counter	: Unit for getting the count-inputs on the occurrence of events that may be at irregular intervals. It functions as timer when given count-input at regular intervals.
Debouncing	: When a key is pressed, due to spring action, the key vibrates and thus makes and breaks the contacts. This causes multiple 0s and 1s before the switch pressed state is accounted for. Debouncing by hardware or software removes the signals due to bounces.
Delay	: An action or communication or execution of codes or occurrence of an event after blocking for a certain pre-defined period.
Demultiplexing	: A way to separate a multiplexed input and direct the messages to one of the multiple channels at an instance.
Device	: A unit that has a processing element and that connects to the processor of embedded system internally or through the port or bus. It has fixed pre-assigned port addresses (device addresses) according to its interfacing or bus controller circuit. A device may not provide for bus controller in it for enabling it to function in bus master mode and thus can function in slave mode only.
Device Decoder	: A circuit to take the system address bus signals as the input and generate a device select signal, CS, for the port address selection during the device read or write instructions of the system processor.
Event	: A change of present condition, which gives an electric signal at input or output pin or which changes a status bit or which interrupts the processor to enable some action by switching the context and running can ISR.
Event flag	: A Boolean variable to indicate the event occurrence when it is true; it can be a status register bit. An event register may store the event flag. A flag auto-resets on response to the event in most systems.

Free Running Counter	: A counter that starts on power-up, which is driven by an internal clock (system clock) directly or through a prescaler or rate control bits and which can neither be stopped nor be reset.
FSK modulation	: Frequency Shifted Keying. The 0 and 1 logic states are at different frequency levels. For example, 0 at 1050 Hz and 1 at 1250 Hz on a telephone line. It permits use of a channel or a line such as telephone line for serial bit transmission and reception.
Full duplex	: A serial port having two distinct IO lines or communication channels. For example, a modem connection to the computer COM port. There are two lines TxD and RxD at 9 pins or 25 pins connector. Message flows both ways at an instance.
Half duplex	: A serial port having one common IO line or channel. For example, a telephone line. Message flows one way at an instance.
Handshaking signals	: The signals before storing the bits at the port buffer or before accepting the bits from the port buffer or the signals to setup or end the communication between two source and destination.
Hardware timer	: A timer present in the system as hardware which gets the inputs from the internal clock with the processor or which enables the system clock ticks (interrupts). A device driver program programs it like any other physical device.
HDLC	: High Level Data Link Control Protocol. It is for synchronous communication between primary (master) and secondary (slave) as per standard defined. It is a bit-oriented protocol.
Host	: A controller node using a protocol and a circuit for enabling the system to connect the number of devices or peripherals and for providing bus master mode functioning as well as receiving signals and bits from other hosts or devices.
IO Port	: A port for input or output operation at an instant. Handshake input and handshake output ports are also known as IO ports. For example, a keypad is said to connect to an IO port.
I^2C bus	: A standard bus that follows a communication protocol and is used between multiple ICs. It permits a system to get data and send data to multiple compatible ICs connected on this bus.
Input Buffer	: A buffer where an input device puts a byte(s) and the processor read that later.
IrDA	: Infrared Data Association recommended protocol for IR remote control and communication over short distance to device or system in line of sight.
ISA bus	: A standard bus based on 'IBM Standard Architecture' Bus.
Isosynchronous Communication	: Communication in which a constant phase difference is not maintained between the frames but maintained within a frame. Clocks that guide the transmitter and receiver are not separate. Only the maximum time interval is not prefixed between which a frame of bytes transmits that is, it can be variable. Between the frames, there is handshaking between the two ends or there may be a pause. Uses are for transmission on a LAN or between two processors.
Keypad and Keyboard Controllers	: The controllers for interfacing with keypads and keyboard such that they do debouncing of keys, buffer the input characters and interrupt the processor on each input or at end of the line character and send ASCII code(s) as input(s) to the processor for further processing and interpretation as data or command.

LCD controller	: A controller for LCD displays with internal CGRAM (character graphic RAM) and ROM for fonts of the characters and which gets the commands and data for display from the system port.
Master slave communication	: A communication between two processors or devices when one processor guides the other using a clock the transmission of the bits to a slave after or before receiving acknowledgement or reply from the addressed slave. A slave can also function as master or vice versa by an appropriate program bit or hardware control.
Multiplexing	: A way to direct the messages to output channel from the multiple source channels.
On-chip ports and devices	: The ports and devices along with the processor unit, for example, in microcontrollers, which communicate using system internal buses.
Open drain output	: A gate with an internally missing connection between its drain and supply. The advantage is that it pulls up the required circuit voltage and current levels when interfacing. An external pull up circuit is needed when using the output.
Output buffer	: A register buffer from where an output device receives the byte(s) after a processor-write operation.
Parallel port	: A port for read and write operations on multiple bits simultaneously at an instance.
PCI bus	: A standard bus used as a 'Peripheral Component Interconnect' bus.
PCI/X bus	: A standard bus used as a 'PCI Extended 'bus'.
PISO	: A shift register for a Parallel Input and Serial Output. It is used for serial bit reception in synchronous mode.
Plug and play	: A device on a bus can be automatically detected when it is attached to the bus and the device can be used directly without resetting or restarting the system.
Protocol	: A way of transmitting messages on a network by using software that adds additional bits such as the starting bits, headers, addresses of source and destination, error control and ending bits. A protocol suite may have multiple layers and each layer or sublayer uses its protocol before a message transmits on a network.
PSK modulation	: Phase Shifted Keying modulation. The 0 and 1 logic have different phases in a high frequency signal. PSK modulation permits use of a channel or line such as telephone line for serial bit transmission and reception.
QPSK	: Quadrature Phase Shifted Keying. An example is the pair of bits 00, 01, 10 and 11, which are sent at different quadrant phase differences of a voice frequency signal. It permits use of the telephone line for serial bit transmission and reception at double the rate. It permits the 56 kbps modem to show a performance equivalent to 112 kbps. QPSK and its enhancements are also used in wireless communication extensively.
Quasi bi-directional port	: A port with the dual advantage of using a pull-up circuit as per the voltage and current levels required when interfacing it and using no pull-up circuit for a short period sufficient to drive an LSTTL circuit.
Real Time	: A time that always increments at constant intervals without stopping or resetting and that is used as a reference by the system at all times.
Real Time Clock (RTC)	: A clock that continuously generates interrupts at regular intervals endlessly. An RTC interrupt ticks the other timers of the system, for example, software timers (SWTs).

RS232C port	: A standard for UART transmission and reception in which TxD and RxD are at different voltage levels (+12 V for '0' and –12 V for '1') and handshaking signals, CTS, RTS, DTR, DCD and RI are at the TTL levels. The RS232C standard is used at the COM ports.
RxD	: A line used for reception of UART serial bits. The 0 and 1 signals are at TTL or RS232C levels and are similar to that for a TxD line.
Serial Port	: A port for read and write operations with one bit at an instance and where each bit of the message is separated by constant time intervals.
SIPO	: A shift register for Serial Parallel Input and Parallel Output. It is used for serial bits transmission in synchronous mode.
Software timer	: Software (a service routine) that executes and increases or decreases a count-variable on an interrupt from a timer output or from a real-time clock interrupt. A software timer also generates interrupt on overflow of count-value or on finishing value of the count-variable (reaching the predefined value) or generating a message for a task. The interrupts can generate by using software interrupt instruction such as SWI.
Status register	: A register for bits, which reflects the status at the port buffer. It is for a *read* operation only. The status register bit or bits may or may not auto-reset on device servicing after the read.
Synchronous Communication	: Communication in which a constant phase difference is maintained between the clocks that guide the transmitter and receiver. A maximum time interval is pre-fixed between which a frame of bytes transmits.
System Clock	: A clock scaled to the processor clock and which always increments without stopping or resetting and generates interrupts at preset time intervals.
Time division multiplexing	: A way by which messages from different channels can be sent in different time slots.
Timer Finish	: A state after the timer acquired the preset count-value and stopped. An interrupt generates on finishing.
Timer Overflow or Time-Out	: A state in which the number of count-inputs exceeded the last acquirable value and on reaching overflow state, an interrupt can be generated.
Timer Reset	: A state in which the timer shows all bits as 0s or 1s. A reset can also be after overflow in case a timer is programmed for continuous running.
Timer Reload	: State in which timer shows all bits as 0s or 1s. A reload can also take place after finishing in case a timer is programmed for auto reload and start again.
Touch Screen	: A GUI device for displaying icons, pictograms, menus and virtual keyboard on an LCD screen and giving input commands or selecting menus or keying in the data using finger or stylus for touching at appropriate screen-position.
TxD	: A line used for transmission of UART serial bits. The 0 and 1 signals are at RS232C voltage levels when RS232C COM port is used, or at the TTL levels in microcontrollers.
UART	: A standard Asynchronous Serial Input and output port for serial bits. UART (in microcontrollers) usually sends a byte in 10-bit format or 11-bit format. The 10-bit format is used when a start bit precedes the 8-bit message (character) and a stop bit succeeds the message. An 11-bit format is used when a special bit also precedes the stop bit.
USB bus	: A standard plug and play bus for fast serial transmission and reception.
Watchdog timer	: A timing device in a system that resets or executes a watchdog timer service routine (WDT routine) after fetching the interrupt vector address at the system after a predefined timeout in case a watched event does not happen. When the watched event occurs, it is restarted so that it does not timeout and does not execute WDT routine.
WLAN	: A wireless LAN for networking mobile and wireless devices with a fixed infrastructure and which enables access of devices through access points. The device functions according to IEEE 802.11 standards specified protocols.
ZigBee	: A new wireless network protocol for short-range communication among number of sensors and devices and has self-discovery and self-organizing features.

Review Questions

1. (a) What is the advantage of a processor that maps the addresses of IO ports and devices like a memory-device? (b) Give a diagram to interface the port devices with the system buses.
2. Compare the advantages and disadvantages of data transfers using serial and parallel ports/devices.
3. (a) Explain three modes of serial communication, 'asynchronous' 'isosynchronous' and 'asynchronous' using serial devices with one example each. (b) Describe and compare UART, RS232C and SDIO devices.
4. How do the following indicate the start and end of a byte or data frames? (a) UART (b) HDLC (c) CAN
5. What are the internal serial-communication devices in (a) 8051 and (b) 68HC11? Compare the modes of working of each of these.
6. A device port may have multibyte data input buffer(s) and data output buffer(s). What are the advantages of these?
7. Explain the advantages of Internet-enabled systems. How is the Internet-enabled device incorporated in the embedded system?
8. Explain the advantages of wireless devices. How do wireless devices network using different protocols?
9. What do you mean by buses for networking of serial devices? What do you mean by buses for networking of parallel devices?
10. Explain use of each control bit of I^2C bus protocol.
11. What do you mean by plug and play devices? What are bus protocols of buses UART, RS232C, USB, Bluetooth, CAN and PCI that support plug and play devices?
12. What do you mean by hot attachment and detachment? What are bus protocols of buses Bluetooth, UART, CAN, PCI, and USB that support hot attachment and detachment?
13. What is a timer? How does a counter perform (a) timer functions (b) prefixed time inititated events generation and (c) time capture functions?
14. Why do you need at least one timer device in an embedded system?

Practice Exercises

15. How do the following device features help in embedded systems? (a) Schmitt trigger input (b) low voltage 3.3 V IOs (c) Dynamically controlled impedance matching (c) PCS subunit (d) PMA subunit and (e) SerDes. Give one exemplary application of each.
16. PPP protocol for point to point networking has 8 starting flag bits, 8 address bits, 8 protocol specification bits, variable number of data bits, 16-bit CRC and 8 ending flag bits. The maximum number of bits per PPP frame can be 12064. How many maximum number of bytes can be transferred per PPP frame? What is the minimum percentage of overhead in the payload (frame)?
17. List the applications of the free running counter, periodically interrupting timer and pulse accumulator counter (PACT). How do you get PWM output from a PACT? How do you get DAC output from a PWM device?
18. A 16-bit counter is getting inputs from an internal clock of 12 MHz. There is a prescaling circuit, which prescales by a factor of 16. What are the time intervals at which overflow interrupts occur from this timer? What will be period before which these interrupts must be serviced?
19. What do you mean by a software timer (SWT)? How do the SWTs help in scheduling multiple tasks in real time? Suppose three SWTs are programmed to timeout after 1024, 2048 and 4096 times from the overflow interrupts from the timer. What will be rate of timeout interrupts from each SWT?
20. What are the advantages and disadvantages of negative acknowledgement bit?
21. A new generation automobile has about 100 embedded systems. How do the bus arbitration bits, control bits for address and data length, data bits, CRC check bits, acknowledgement bits and ending bits in CAN bus help the networking of devices distributed in an automobile system.
22. How does the USB protocol provide for the device attachment, configuration, reset, reconfiguration, bandwidth sharing with other devices, device detachment (while others are in operation) and reattachment?
23. Design a table that compares the maximum operational speeds and bus lengths and give two example of the uses of each of the following serial devices: (a) UART (b) 1-wire CAN (c) Industrial I^{2}C (d) SM I^{2}C Bus (e) SPI of 68 Series Motorola Microcontrollers (f) Fault tolerant CAN (g) Standard Serial Port (h) FireWire (i) I^{2}C (j) High Speed CAN (k) IEEE 1284 (l) High Speed I^{2}C (m) USB 1.1 Low Speed Channel and High Speed Channel (n) SCSI parallel (o) Fast SCSI (p) Ultra SCSI-3 (q) FireWire/IEEE 1394 (r) High Speed USB 2.0.
24. Use web search. Design a table that compares the maximum operational speeds and bus lengths and give two example of the uses of each of the following parallel devices: (a) ISA (b) EISA (c) PCI (d) PCI-X (e) COMPACT PCI (f)_GMII (Gigabit Ethernet MAC Interchange Interface) (g) XGMI (10 Gigabit Ethernet MAC Interchange Interface) (h) CSIX-1. 6.6 Gbps 32-bit HSTL with 200 MHz performance (i) RapidIO™ Interconnect Specification v1.1 at 8 Gbps with 500 MBps performance or 250 MHz dual direction registering performance using 8-bit LVDS (Low Voltage Data Bus).
25. Use web search and design a table that gives the features of the following latest generation serial buses. (a) IEEE 802.3-2000 [1 Gbps bandwidth Gigabit Ethernet MAC (Media Access Control)] for 125 MHz performance (b) IEE P802.3oe draft 4.1 [10 Gbps Ethernet MAC] for 156.25 MHz dual direction performance] (c) IEE P802.3oe draft 4.1 [12.5 Gbps Ethernet MAC] for four channel 3.125 Gbps per channel transreceiver performance] (d) XAUI (10 Gigabit Attachment Unit) (e) XSBI (10 Gigabit Serial Bus Interchange) (f) SONET OC-48, OC-192 and OC-768 (g) ATM OC-12/46/192.
26. Take a mobile smart phone with a T9 keypad. Write a table for the states of each key. Write another table for the new states generated by a combination of two keys.
27. Compare the parallel ports interfaces for the keypad, printer, LCD-controller and touchscreen.
28. Show the use of USB devices in the digital camera, printer and computer for downloading a picture from camera to computer, printing the pictures in camera and saving in flash memory. What is the difference between USB host and USB device in a system?
29. Compare different serial buses.
30. Compare different wireless protocols.

Device Drivers and Interrupts Service Mechanism

4

R e c a l l

We have learnt the following in previous chapter:

- **Embedded system hardware has devices which communicate through serial and parallel ports and buses. There may also be ports for real-time voice and video I/Os.**
- **A microcontroller has serial communication and timing devices. It may have keypad, stepper motor, LCD and touch screen controllers.**
- **Serial or parallel buses interconnect the distributed ports and devices.**
- **I^{2}C bus is used for inter-IC communication. It interconnects multiple distributed ICs.**
- **CAN bus is used at control network of the distributed devices. It is used in automobiles and industrial systems.**
- **USB is used for the fast serial transmission and reception between the embedded-system and serial devices such as the keyboard, printer and scanner.**
- **FireWire (IEEE 1394) is bus used for the high-speed interfacing of 800 Mbps multimedia devices.**

R e c a l l

- Parallel buses, ISA and PCI/PCI-X are used for bus communication of devices between the host computer or system and PC-based devices or systems or cards, for example, NIC (Network Interface Card).
- Wireless protocols are used for the communication and synchronization of distributed devices in wireless personal area network.
- Internet-enabled embedded systems can network to the Internet using the TCP/IP suite of protocols.

We also learnt

- A device-access is required for opening, connecting, binding, reading, writing, disconnecting or closing it. Processor accesses a device using the addresses of device registers and buffers. A processor accesses the internal devices, devices at the I/O ports, peripheral devices and other off-chip devices using the addresses.
- A simple device such as SPI port (Section 3.2.4) has addresses for three sets of its registers: data register(s) (or buffers), control register(s) and status register(s).
- A device can also have number of registers (Table 3.4). For example, PCI bus-driven device (Section 3.12.2) has 64 bytes standard device-independent configuration registers.

L E A R N I N G O B J E C T I V E S

In this chapter, we will learn how the concept of interrupt service routines is used to address and service the device IOs, requests and interrupts. We will learn the following:

1. *Programmed I/O busy and wait method and problems with this I/O method.*
2. *Interrupts and working of interrupts service mechanism in the system and simple examples of hardware and software interrupts.*
3. *Interrupt service routine (ISRs) are called by the system when device-hardware interrupts take place.*
4. *Software functions for the signals and exceptions also call ISRs. An ISR is also called on a trap or execution of a software instruction for interrupt.*
5. *Use of interrupt vectors, vector table and masking.*
6. *Interrupt latency and deadline for an interrupt service.*
7. *Context and context switching on an interrupt.*
8. *New methods for the fast context switching adopted in the processors.*
9. *Classification of processors for an interrupt service that 'Save' or 'Don't Save' the context other than the program counter.*
10. *Use of the DMA channel for facilitating the small interrupt latency period for the multiple data transfers in quick succession.*
11. *Assignment of software and hardware priorities among the multiple sources of interrupts.*
12. *Methods of service in case of simultaneous service demand from multiple interrupting sources.*
13. *Device drivers for a device or port initialization and accesses.*
14. *Use of device drivers, for example, Linux Internals.*
15. *Examples of device initialization and device driver coding for the parallel ports and serial-line UART.*

4.1 PROGRAMMED-I/O BUSY-WAIT APPROACH WITHOUT INTERRUPT SERVICE MECHANISM

Example 4.1 shows an example of programming a device service with programmed I/O *busy-wait* approach without using a device interrupt and the corresponding ISR. This example will make clear the problems in this approach and advantages of using an interrupt-based service mechanism.

Example 4.1

Assume a 64 kbps network. Using a UART that transmits in the format of 11-bit per character, the network transmits at most 64 kbps ÷ 11 = 5818 characters per second, which means that for every 171.9 µs a character is expected. Before 171.9 µs, the receiver port must be checked to find and read another character assuming that all the received characters are in succession without any time gap.

Let port A be at an Ethernet interface card in a PC, and port B be its modem input which puts the characters on the telephone line. Let *In_A_Out_B* be a routine that receives an input character from port A and re-transmits an output character to port B. Assume that there is no interrupt generation and interrupt service (handling) mechanism. Let *In_A_Out_B* routine has to call the following steps *a* to *e* and executes the cycles of functions *i* to *v*, thus ensuring that the modem port A does not miss reading the character.

In_A_Out_B routine:

1. Call function *i*
2. Call function *ii*
3. Call function *iii*
4. Call function *iv*
5. Call function *v*
6. Loop back to step 1

In_A_Out_B routine calls the following steps.

Step *a*: Function *i*: Check for a character at port A. If not available, then wait.

Step *b*: Function *ii*: Read port A bytes (characters for message) and return to step *a* instruction, which will call function *iii*.

Step *c*: Function *iii*: Decrypt the message and return to step *a* instruction, which will call function *iv*.

Step *d*: Function *iv*: Encode the message and return to step *a* instruction, which will call function *v*.

Step *e*: Function *v*: Transmit the encoded message to port B and return to step *a* last instruction, which will start step *a* from the beginning.

Step *a* is also called polling. Polling a port means to find the status of the port, ready with a character (byte) at input. Polling must start before 171.9 μs because characters are expected at 64 kbps. If the program instructions in the steps *b, c, d* and *e* and functions *ii* to *v* take a total running time of less than 171.9 μs then this approach works.

Problems with the busy-wait programming approach is as follows.

1. The program must switch to execute the *In_A_Out_B* cycle of steps *a* to *e* within a period less than 171.9 μs. Programmer must ensure that steps of *In_A_Out_B* and any other device program steps never exceed this time.
2. When the characters are not received at Port A in regular succession, the waiting period during step *a* for polling the port can be very significantly. Wastage of processor time for the waiting periods is the most significant disadvantage of the busy-wait approach.
3. When other ports and devices are also present in the system, the programming problem is to poll each port and device and ensure that the program switches to execute the *In_A_Out_B* step *a* as well as switches to poll each port or device on time and then execute each service routines related to the functions of other ports and devices within a specific time interval and ensure that each one is polled on time.
4. The program and functions are processor- and device-specific in the previous busy-wait approach and all system functions must execute in synchronization and the timings are completely dependent on periods taken for software execution.

Instead of continuously checking for characters at the port A by executing function (*i*), when a modem receives an input character and sets a status bit in its status register, an interrupt from port A should be generated. In response to the interrupt an interrupt service routine ISR_ PortA _Character should then be executed (Example 4.2 in Section 4.2). This will be the efficient solution instead of wait at step *a*.

Device service without using an ISR is by the routine (function) call similar to *In_A_Out_B*.

Each routine (function) call has the following features.

1. A function call after executing any instruction in any program is a planned (user-programmed) diversion from the current sequence of instructions to another sequence of instructions and this sequence of instructions executes till the return from that.
2. On a function call, the instructions are executed as a function in the 'C' or a method in Java.
3. Function calls are nested. Nesting can be explained as follows: when a function 1 calls another function 2 which in turn calls another function 3, then on return from 3, the return is to function 2 and then to function 1.

Figure 4.1 shows the *In_A_Out_B* routine steps *a* to *e* for the five functions *i* to *v* called by *In_A_Out_B* and how each called function processes on a *call* and on a *return* from that. Numberings on the arrows show the sequences during the program run (flow).

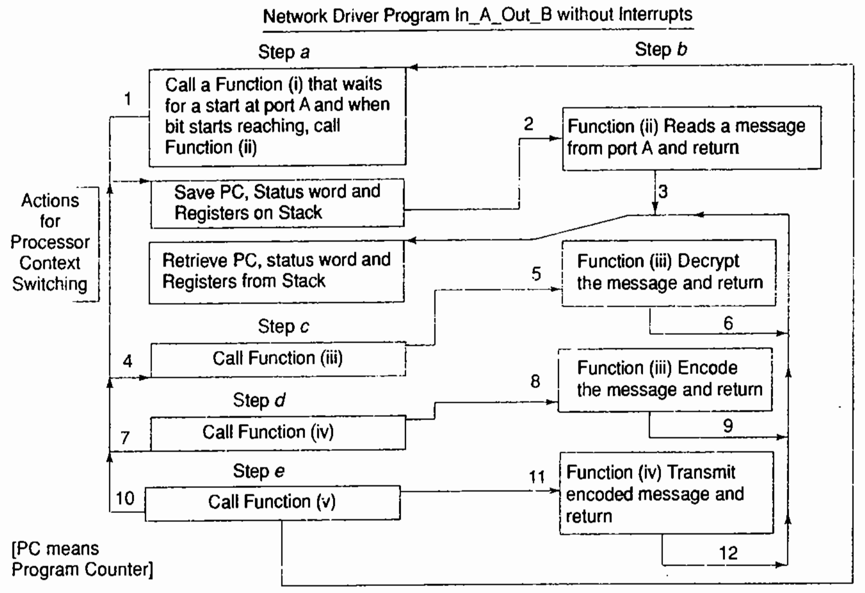

Network Driver Program In_A_Out_B without Interrupts
Step *a*
Step *b*
Call a Function (i) that waits for a start at port A and when bit starts reaching, call Function (ii)
1
2
Function (ii) Reads a message from port A and return
Save PC, Status word and Registers on Stack
3
Actions for Processor Context Switching
Retrieve PC, status word and Registers from Stack
5
Function (iii) Decrypt the message and return
6
Step *c*
4
Call Function (iii)
8
Function (iii) Encode the message and return
Step *d*
9
7
Call Function (iv)
Step *e*
11
Function (iv) Transmit encoded message and return
10
Call Function (v)
12
[PC means Program Counter]

Fig. 4.1 Steps *a* to *e* for five function calls in an exemplary network drive program. *IN_A_OUT_B*, also shown is how each called function processes on a call and on a return. Numberings on the arrows show the program running sequences

One approach is 'programmed IO' transfer, also called 'busy and wait' transfer for service (accessing the device addresses for input or output or any other action). System functions in synchronization and the timings are completely software-dependent. When waiting periods are a significant fraction of the total program's execution period, wastage of the processor's time in waiting is the most significant disadvantage of this approach. Programmed IO approach can be used in single-purpose processors (controllers).

4.2 ISR CONCEPT

Interrupt means event, which invites the attention of processor for some action on the hardware or software event.

1. 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. This interrupt is called hardware interrupt.
2. When software run-time exception condition is detected, either processor hardware or a software instruction generates an interrupt. This interrupt is called software interrupt or *trap* or *exception*.
3. Software can execute the software instruction for interrupt to *signal* the execution of ISR. The interrupt due to signal is also a software interrupt [The *signal* differs from the function in the sense that the execution of the signal handler function (ISR) can be masked and till the mask is reset, the handler will not execute. Function on the other hand always executes on the call after a call-instruction.]

In response to the interrupt, the routine or program, which is running at present gets interrupted and an ISR is executed. ISR is also called device driver ISR in the case of devices and is called *exception* or *signal* or *trap handler* in the case of software interrupts. Device driver ISRs execute on software interrupts from device open ( ), close ( ), read ( ), write ( ) or other device functions.

Examples in Sections 4.2.1 and 4.2.2 show the importance of interrupts and accessing of the devices using the ISRs and the importance of using the ISRs which generate on *traps* or *exceptions* or *signals*.

4.2.1 Examples of Port or Device Interrupts and ISRs

Following are the examples of interrupt events and accessing of devices using the ISRs.

Example 4.2

Recapitulate Example 4.1. Assume that a character input to the modem generates a port A interrupt and sets a status bit in the status register. On interrupt, a service routine ISR_PortA_Character runs so that it ensures that the modem port A does not miss reading the character. ISR_ PortA_Character executes step *f* in place of the step *a* function *i* and step *b* function *ii* of *In_A_Out_B* routine in Example 4.1. It places the read character in a memory buffer. Steps *c*, *d* and *e* are independent and are now parts of a function-call Out_B.

ISR_ PortA _Character executes as follows:

1. Step *f* function *vi*: *Read* Port A character. *Reset* the status bit so that the modem is ready for the next character input (resetting of the status bit is generally automatic without the need for specific instruction). *Put* it in a memory buffer. Memory buffer is a set of memory addresses where the bytes (characters) are queued for processing later.
2. Return from the ISR.

Out_B routine is as follows:

1. Step *g*: Call function *vi* to decrypt the message characters at the memory buffer and return for the next instruction step *h*.
2. Step *h*: Call function *vii* to encode the message character and return for the next instruction step *k*.
3. Step *k*: Call function *viii* to transmit the encoded character to port B.
4. Return from the function.

Figure 4.2 shows the step *f* executing on ISR_ PortA _Character on port A interrupt and steps *g* to *k* in Out_B routine. Numberings on the arrows show the program running sequences.

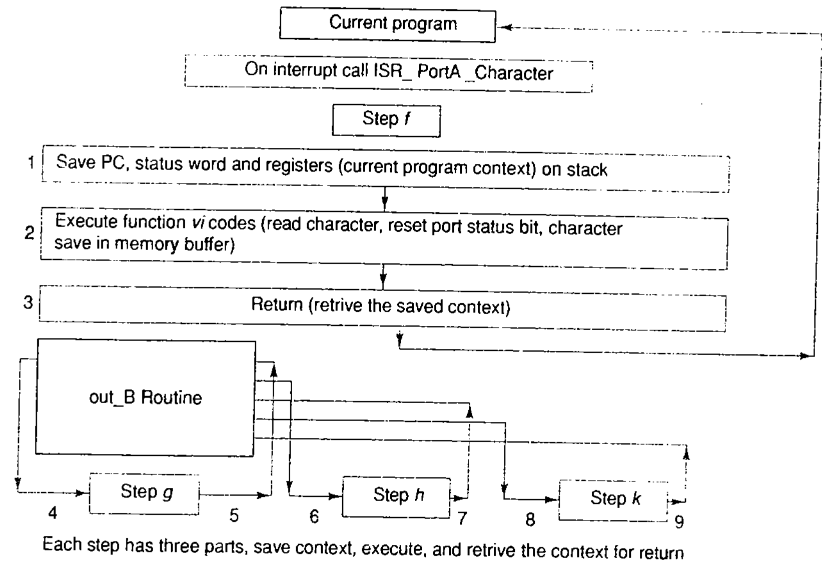
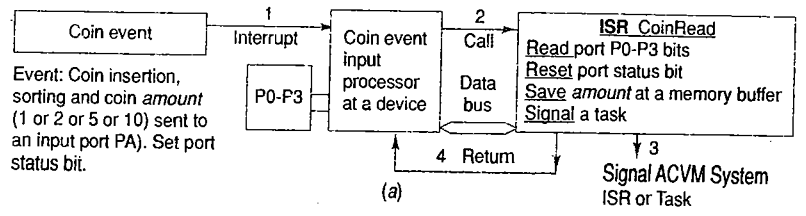

Current program
On interrupt call ISR_ PortA _Character
Step *f*
1 Save PC, status word and registers (current program context) on stack
2 Execute function *vi* codes (read character, reset port status bit, character save in memory buffer)
3 Return (retrive the saved context)
out_B Routine
4 Step *g* 5
6 Step *h* 7
8 Step *k* 9
Each step has three parts, save context, execute, and retrive the context for return

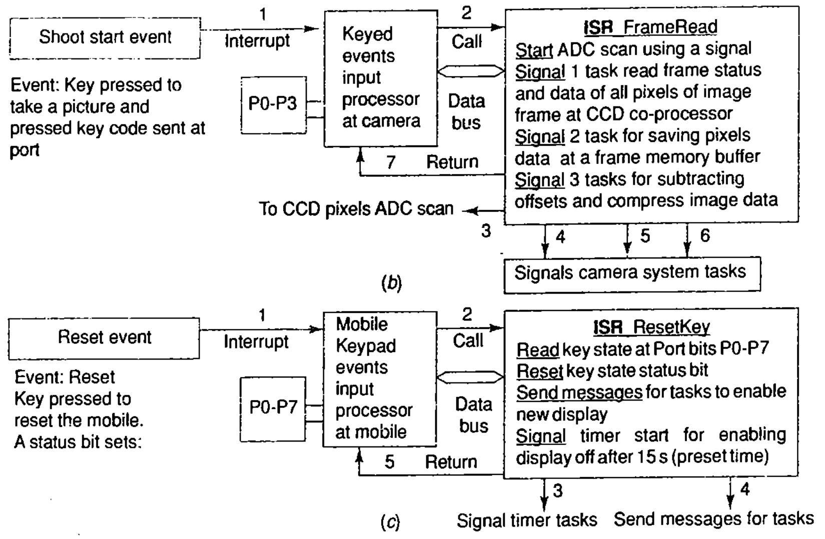
Fig. 4.2 Step *f* executing on ISR_ PortA _Character on port A interrupt and steps *g* to *k* in Out_B routine. Numberings on the arrows show the sequences of running the program-steps *f, g, h* and *k*

Example 4.3

Assume a device for coin amount input in an automatic chocolate-vending machine (Section 1.10.2). Without interrupt mechanism, one way is the busy wait transfer by which the device waits for the coin continuously, activates on sensing the coin and runs the service routine.

In the event-driven method the device should awaken and activate on each *interrupt* after sensing each coin-inserting event. The device is at an input port. It collects a coin inserted by a child. The system awakens and activates on interrupt through a hardware interrupt. The system on port hardware *interrupt* collects the coin by running a service routine. This routine is called interrupt handler routine or ISR or *device driver function* for the coin-port read. Figure 4.3(a) shows the ISR in the ACVM example.


Coin event
Event: Coin insertion, sorting and coin *amount* (1 or 2 or 5 or 10) sent to an input port PA). Set port status bit.
1 Interrupt
P0-P3
Coin event input processor at a device
2 Call
Data bus
ISR CoinRead
Read port P0-P3 bits
Reset port status bit
Save *amount* at a memory buffer
Signal a task
4 Return
3
Signal ACVM System ISR or Task
(a)



Shoot start event
1
Interrupt
Event: Key pressed to take a picture and pressed key code sent at port
P0-P3
Keyed events input processor at camera
2
Call
Data bus
7 Return
ISR FrameRead
Start ADC scan using a signal
Signal 1 task read frame status and data of all pixels of image frame at CCD co-processor
Signal 2 task for saving pixels data at a frame memory buffer
Signal 3 tasks for subtracting offsets and compress image data
To CCD pixels ADC scan
3
4 5 6
Signals camera system tasks
(b)
Reset event
1
Interrupt
Event: Reset Key pressed to reset the mobile. A status bit sets:
P0-P7
Mobile Keypad events input processor at mobile
2
Call
Data bus
5 Return
ISR ResetKey
Read key state at Port bits P0-P7
Reset key state status bit
Send messages for tasks to enable new display
Signal timer start for enabling display off after 15 s (preset time)
3 4
(c)
Signal timer tasks
Send messages for tasks

Fig. 4.3 (*a*) Use of ISR in the automatic chocolate vending machine (*b*) Use of ISR and three signals 1, 2 and 3 for three tasks in the digital camera example (*c*) Use of ISR in the mobile phone reset-key interrupt example

Example 4.4

Assume a digital camera system (Section 1.10.4). It has an image input device. When the system activates the device should grab image-frame data. The system awakens and activates on a switch *interrupt*. The interrupt is through a hardware signal from the device switch. On the *interrupt*, an ISR (can also be considered as camera's imaging device-driver function) for the *read* starts execution, it passes a message (signal) 1 to a function or program thread or task, which senses the image and then the function reads the CCD device frame buffer; then the routine passes signal 2 to another function or program thread or task to process and then signal 3. Subtracts offsets using a task and compresses image-data using a task. This task also saves the image frame data-compressed file in a flash memory. The camera system again awakens and activates on *interrupt* through a hardware signal from a device switch and prints the file picture image after file decompression. The system on *interrupt* then runs another ISR. The ISR routine is the device-driver write-function for the outputs to printer through the USB bus connected to the printer. Figure 4.3(b) shows the use of the ISR for frame read in the digital camera example.

ISR accesses a device for service (configuring, initializing, activating, opening, attaching, reading, writing, resetting, deactivating or closing). ISRs thus function as the device drivers.

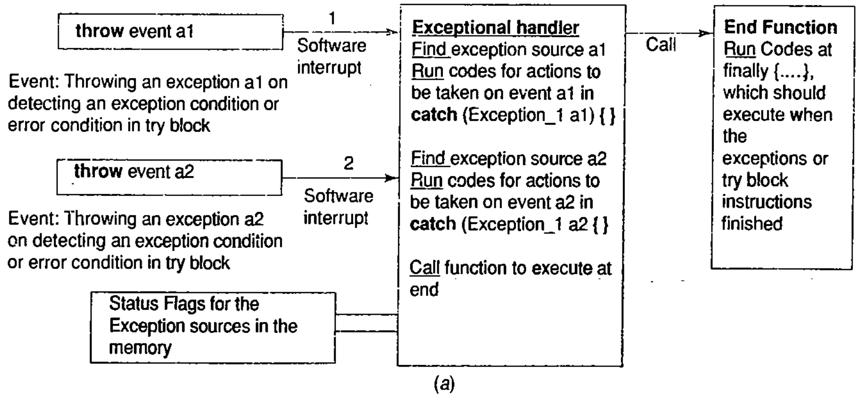
Example 4.5

Assume a mobile phone system (Section 1.10.5). It has a system reset key, which when pressed resets the system to an initial state. When reset key is pressed the system awakens and activates a *reset interrupt* through a hardware signal from reset key circuit. On the *interrupt*, an ISR (can also be considered as reset-key device-driver function) suspends all activity of the system, sends messages to the display functions for the program threads or the tasks for displaying the initial reset state menu and graphics on LCD screen, and also signals to activate LCD display-off timer device for 15s timeout (for example). After the timeout the system again awakens and activates on *interrupt* through the internal hardware signal from timer device and runs another ISR to send a control bit to the LCD device. The ISR routine is the device-driver LCD off-function for the LCD device. The devices switch off by reset of control bit. Figure 4.3(c) shows the use of the ISR in the mobile-phone reset-key interrupt.

4.2.2 Examples of Software Interrupts and ISRs

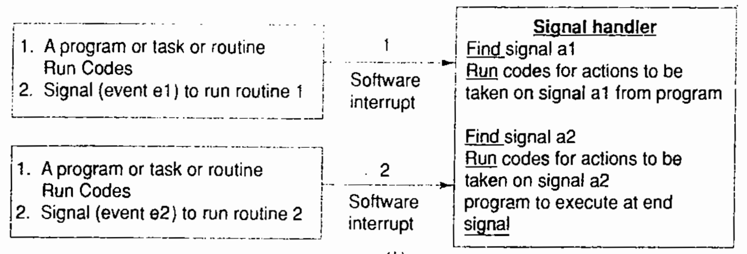
Examples 4.2 to 4.5 clearly show that interrupts and ISRs (device-drivers) play the major role in using the system hardware and devices. Think of any system hardware and it will have devices and thus needs device drivers. The embedded software or the operating system for application software must consist of the codes for the device (i) configuring (initializing), (ii) activating (also called opening or attaching), (iii) driving function for read, (iv) driving function for write and (iv) resetting (also called deactivating or closing or detaching). Each device task is completed by first using an ISR—a device-driver function calls the ISR by using a software interrupt instruction (SWI).

A program must detect error condition or run-time exceptional condition encountered during the running. In a program either the hardware detects this condition (called trap) or an instruction SWI is used that executes on detecting the exceptional run-time condition during computations or communication. For example, detecting that the square root of a negative number is being calculated or detecting the illegal argument in the function or detecting that the *connection* to network is not found. Detection of exceptional run-time condition is called *throwing* an exception by the program. An interrupt service routine (exceptional handler routine) executes, which is called *catch* function as it executes on catching the exception thrown by executing an SWI. Figure 4.4(a) shows use of SWI instruction for calling an ISR in the function for throwing and catching the exceptional run-time conditions encountered during computations. Figure 4.4(b) shows the use of signal generated by SWI, and signal handling after that.



throw event a1
1
Software interrupt
Event: Throwing an exception a1 on detecting an exception condition or error condition in try block
throw event a2
2
Software interrupt
Event: Throwing an exception a2 on detecting an exception condition or error condition in try block
Status Flags for the Exception sources in the memory
Exceptional handler
Find exception source a1
Run codes for actions to be taken on event a1 in catch (Exception_1 a1) { }
Find exception source a2
Run codes for actions to be taken on event a2 in catch (Exception_1 a2 { }
Call function to execute at end
Call
End Function
Run Codes at finally {....}, which should execute when the exceptions or try block instructions finished

(*a*)


1. A program or task or routine
Run Codes
2. Signal (event e1) to run routine 1
1
Software interrupt
1. A program or task or routine
Run Codes
2. Signal (event e2) to run routine 2
2
Software interrupt
Signal handler
Find signal a1
Run codes for actions to be taken on signal a1 from program
Find signal a2
Run codes for actions to be taken on signal a2
program to execute at end signal

(*b*)

Fig. 4.4 (*a*) Use of software interrupt (SWI) instruction for calling an interrupt service routines (ISR) in the software on throwing and catching the exceptional run-time conditions encountered during computations (*b*) Use of SWIs to signal another routines or program tasks or program threads to start

Example 4.6 is given here to clearly show that SWI and execution of the ISRs (also called *exception handlers* or just *exceptions*) on SWIs. The SWIs also play a major role in embedded system software by the use of ISRs for device driver functions—create ( ), open ( ), read ( ) or other.

Example 4.6

Consider the following codes.

```
try {
/* Codes for execution in which a run-time exception or number of run-
    time exception conditions may encounter, for example square root
    of a negative number or a percentage value exceeding 100% or decreasing
    below 0%. The condition is trapped and on trapping throws an
    exception*/
.
If ((A - B) < 0.1) throw a1; x = y + sqrt (A - B); /* Find if A - B is
    -ve number. If yes, throw an exception a1 and call a catch
    function). */
.
y = ....../ * Calculate y */
If ((y > 100 | | y < 0) throw a2; /* Find if y > 100. If yes, throw
    exception a2 and call a catch function*/
.
}
```

```
catch (Exception_1 a1) {
 /* Code for action on throwing (trapping) A - B < 0.0 exception */
```

```
 .
 .
}
```

```
catch {Exception_1 a2
 /* Code for action on throwing (trapping) y < 0 or y > 100
    exception*/
 .
 .
}
```

```
finally {
/* Final codes, which should execute on exception or after
    try block instructions over */
 .
 .
}
```

High-level Java or C++ codes when compiled, during compilation the SWI instructions will be inserted for trapping (A – B) as a negative number and for trapping y > 100 or less than 0 as follows:

1. Software instruction SWI a1 will cause processor interrupt. In response, the software ISR function 'catch (Exception_1 a1) { }' executes on throwing of the exception a1 at try block execution. SWI a1 is used after catching the exception a1 whenever it is thrown.
2. Software instruction SWI a2 will cause processor interrupt. In response, the software ISR function 'catch (Exception_2 a2) { }' executes on throwing of the exception a2 during try block execution. SWI a2 is used after catching the exception a2 whenever it is thrown.
3. Software instruction SWI a3 will cause processor interrupt and in response will *signal* software ISR function 'finally { }' to execute either at the end of the try or at the end of the catch function codes. SWI a3 is used after the try and catch functions finish, then *finally* function will perform final task, for example, exit from the program or call another function.

A user program under execution currently by the processor does not know when its try function will throw the exceptions a1 or a2 or when the signal handler throws a3.

An ISR call has the following features.

1. An ISR call due to interrupt executes an event. Event can occur at any moment and event occurrences are asynchronous.
2. ISR call is event-based diversion from the current sequence of instructions (routine or program) to another sequence of instructions (routine or program). This sequence of instructions executes till the return instruction.
3. Event can be a device or port event or software computational exceptional condition detected by hardware or detected by a program, which throws the exception. An event can be signalled by software interrupt instruction SWI used in device driving functions create ( ), open ( ), etc.
4. An interrupt service mechanism exists in a system to call the ISRs from multiple sources (Section 4.4).

5. Diversion to ISR may or may not take place on finishing the execution of any instruction in the presently running routine. The execution of the ISRs can be masked by an instruction to set a mask bit and can be unmasked by another instruction to reset the mask bit. [Except a few interrupt sources called non-maskable source (Section 4.4.3).] An instruction in a function or program thread or task can disable or enable an ISR call or all ISR calls (Section 4.4.3).
6. On an interrupt call, the instructions do not execute continuously exactly like a C function or a Java method. These execute as per the interrupt mechanism of the system. For example, *'return'* from an ISR differs in certain important aspects. An interrupt mechanism may be such that an ISR on beginning the execution may disable automatically other device(s) interrupt services. These are automatically re-enabled if they were enabled before a service call. Another interrupt mechanism may be such that an ISR on beginning the execution does not disable automatically other device(s) interrupt services and there can be in-between diversion in the case of the unmasked higher priority interrupts (Section 4.5.1).
7. There can be multiple interrupt calls during running of an ISR for diversion to other ISRs. The ISR calls need not be the nesting of the ISRs unlike the case of the function calls and there is diversion to pending higher priority interrupt either at the end or in-between the interrupted ISR.

Section 7.6 will explain the distinction between functions, ISRs and tasks by their characteristics.

Interrupt is an event from a device or hardware action or software instruction. In response to the interrupt, a presently running program is interrupted and a service routine executes. The routine is called ISR. It is also called *device driver* in case of interrupts from the devices. It is also called *exception* handler in case of interrupts from the software. ISR-based approach facilitates an efficient synchronization of the function-calls and ISR-calls. The timings when an ISR executes are hardware or software interrupt event dependent. There is therefore no waiting period due to no need of device polling.

4.2.3 Interrupt Service Threads as Second-Level Interrupt Handlers

An ISR can be executed in two parts.
1. One part is the short execution time taking service routine and can be called as first-level ISR (FLISR). It runs the critical part of the ISR and execute a *signal* function to enable the OS to schedule for running the remaining part later. It can also send a message using a function to enable the OS to iniate a task later on after return from the ISR. The task waits during execution of interrupts routines and signal functions. The FLISR does the device-dependent handling only. For example, it does not perform decryption of data received from the network. It simply does the transfer of data to the memory buffer for the device data.
2. The second part is the long service routine called interrupt service thread (IST) or second-level ISR (SLISR), which executes on the *signal* of the first part. The OS schedules the IST as per its priority. IST does the device-independnet handling. IST is also the software interrupt thread as it is triggered by an SWI (software interrupt instruction) for the *signal* in FLISR.

Figure 4.3(b) showed used of *signal* in ISR-FrameRead in digital camera system. Figure 4.5 shows how ADC scan is initiated by an SLISR call from FLISR. Figure 4.5 shows the FLISR and second-level IST approach to handle the device hardware interrupts followed by software interrupts in upper part and the use of this approach in a camera in lower part.

Interrupt service can be done in two parts: a hardware device-dependent code in the FLISR, which has a short execution time and a software interrupt initiated SLISR, which is also called IST. A task can also be sent message by FLISR. The task runs after the IST.

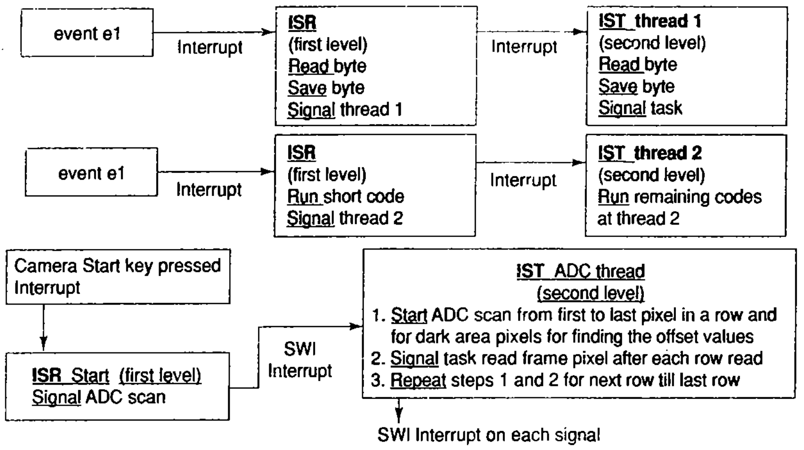


Fig. 4.5 First-level interrupt service routine and second-level interrupt service thread approach to handle the device hardware interrupt followed by the software interrupt to call SLISR–an IST and the use of this approach in a camera

4.2.4 Device Driver

Each device in a system needs device driver routines. An ISR relates to a device driver function. A device driver is a function used by a high-level language programmer and does the interaction with the device hardware and communicates data to the device, sends control commands to the device and runs the codes for reading the device data. A programmer uses generic commands for the device driver for using a device. The OS provides these generic commands.

The examples of generic functions used for the commands to the device are device *create* ( ), *open* ( ), *connect* ( ), *bind* ( ), *read* ( ), *write* ( ), *ioctl* ( ) [for IO control], *delete* ( ) and *close* ( ). Device driver code is different in different OS. Same device may have different codes for the driver when the system is using different OS.

A device driver function uses SWI, which initiates the interrupt service. The device uses the system and IO buses required for the device service. Device driver can be considered as a function in software layer of an application program and the device.

For example, the application program sends the commands to write on display screen of a mobile the *contact names* from the contact database. LCD display device driver calls an SWI, and an ISR does that without the application programmer knowing how does LCD device interface in the system, what are the addresses which are used, what and where and how are the control (command) and status registers used.

For a programmer, using the device driver's generic functions for reading or writing from and to the device is analogous to reading or writing any other device or data file except that the device and file have different device identity numbers.

The driver routine controls a device without requiring understanding of the device configuration, control, status, data and other registers, when using the generic functions. Device driver runs the ISRs of the device. Each ISR is the low-level part of the device driver generic function, which executes on software interrupt instruction.

The driver translates a program generic function for using the device and sends the necessary commands to the device configuration and control registers. The driver uses the device control, status and data registers.

The driver does the opening, configuring, initializing, attaching, reading, writing, closing and detaching the device by initiating the corresponding ISRs.

Drivers of many devices, such as printers, touch screen, LCD display, keypad, keyboard, are part of the OS. Section 4.9 will describe device drivers in detail.

Each device high-level language program in a system uses device driver functions. A programmer uses generic commands, create ( ), open ( ), connect ( ), bind ( ), read ( ), write ( ), ioctl ( ), delete ( ) and close ( ) and uses for each device a device identity number. Device driver executes the SWIs, which call the ISRs for using the device hardware and memory allotted to that. SWIs are dedicated for the device service and perform all the necessary actions.

4.3 INTERRUPT SOURCES

Hardware sources can be from internal devices or external peripherals, which interrupt the ongoing routine and thereby cause diversion to corresponding ISR. Software sources for interrupt are related to (*i*) processor detecting (trapping) computational error for an illegal op-code during execution or (*ii*) execution of an SWI instruction to cause processor-interrupt of ongoing routine.

Each of the interrupt sources (when not masked) (or groups of interrupt sources) demands a temporary transfer of control from the presently executed routine to the ISR corresponding to the source.

The internal sources and devices differ in different processors or microcontrollers or devices and their versions and families. Table 4.1 gives a classification as hardware and software interrupts from several sources. Not all the given types of sources in the table may be present or enabled in a given system. Further, there may be some other special types of sources provided in the system.

Hardware Interrupts Related to Internal Devices There are number of hardware interrupt sources which can interrupt an ongoing program. These are processor or microcontroller or internal device hardware specific. An example of a hardware-related interrupt is timer overflow interrupt generated by the microcontroller hardware. Row 1 of Table 4.1 lists common internal devices interrupt sources.

Hardware Interrupts Related to External Devices – 1 There can be external hardware interrupt source for interrupting an ongoing program that also provides the ISR address or vector address (Section 4.4.1) or interrupt-type information through the data bus. Row 2 of Table 4.1 lists these interrupt sources. External hardware interrupts with ISR addresses information sent by the devices themselves (Section 4.4.1) and are device hardware-specific.

Example 4.7

An example of external hardware-related interrupt with device sending the interrupt on INTR pin is the 80x86 processor. When INTR pin activates on an interrupt from the external device, the processor issues two cycles of acknowledgements in two clock cycles through the INTA (interrupt acknowledgement) pin. During the second cycle of acknowledgement, the external device sends the type of interrupt information on data bus. Information is for one byte n. 80x86 internally signals instruction INT n, which means that it executes interrupt of type n, where n can be between 0 and 255. INT n causes the processor vectoring to address 0x00004 × n. [SWI in 80x86 is denoted by INT.]

Hardware Interrupts Related to External Devices – 2 External hardware interrupts with their ISR vector addresses (Section 4.4.1) are processor or microcontroller-specific interrupts of an ongoing program. External interrupting source does not send interrupt-type or ISR address-related information. An example of external hardware-related interrupt in which the interrupt-type information internally generates is an interrupt on NMI (non-maskable interrupt) pin in the 80x86 processor. Row 3 of Table 4.1 lists these interrupt sources.

Table 4.1 Classification and Sources of Interrupts[1]

Sources	*Examples*
Internal hardware device sources	1. Parallel port; 2. UART serial receiver port – [Noise, Overrun, Frame-Error, IDLE, RDRF in 68HC11]; 3. Synchronous receiver byte completion; 4. UART serial transmit port-transmission complete [e.g. TDRE (transmitter data register Empty); 5. Synchronous transmission of byte completed; 6. ADC start of conversion; 7. ADC end of conversion; 8. Pulse-accumulator overflow; 9. Real-time clock time-outs (Section 3.8); 10. Watchdog timer resets (Section 3.7); 11. Timers overflows on time-out (Section 3.6); 12. Timer comparison with output compare register; 13. Timer capture on input (Section 3.6)
External hardware devices providing the ISR address or vector address or type externally[2]	INTR in 8086 and 80x86
External hardware devices with internal vector address generation	1. Non-maskable pin [NMI in 8086 and 80x86]; 2. Within first few clock cycles unmaskable declarable pin (interrupt request pin) but otherwise maskable XIRQ [in 68HC11]; 3. Maskable pin (interrupt request pin) [INT0 and INT 1 in 8051, IRQ in 68HC11]
Software error-related sources (exceptions[3] *or SW-traps)*	1. Division by zero detection (or *trap*) by hardware; 2. Over-flow by hardware; 3. Under-flow by hardware; 4. Illegal opcode by hardware
Software instruction-related sources (exceptions[4]*or SW-traps SW-signal)*	Programmer-defined exceptions[3] or traps for handling exceptional run-time conditions or programmer-defined *signal* for executing ISR to handle further actions or *signals* from device driver functions

[1] Processor-specific examples are in bracket.

[2] Example 4.7 explains this.

[3] The processor internally generates a trap or exception. An example is *division by zero* in 80x86. Example 4.8 explains this.

[4] The second type of exception is the user program-defined exception. Example 4.6 explained this. *Signal* is a term sometimes used in high level program for software interrupt instruction in assembly language. For example, in VxWorks RTOS. [Refer Section 9.3.] *Signal* or *exception* is an interrupt on the setting of certain conditions or on obtaining certain results or output during a program run or a signal for some action. The condition examples are square root of a negative number or percentage computation resulting in values greater than 100% or an IO connection not found.

Software Error-Related Hardware interrupts There can be the software-error related interrupts generated by processor hardware. Each processor has a specific instruction set. It is designed for that set only. An illegal code (instruction in the software) is an instruction, which does not correspond to any instruction in this set. Whenever the processor fetches illegal code, an interrupt occurs in certain processors. The error-related interrupts are also called hardware-generated *software traps* (or *software exceptions*). A software error called *trap* or *exception* may generate in the processor hardware for an illegal or not-implemented opcode found during execution. The examples are as follows: (i) There is an *illegal opcode trap* in 68HC11. This error causes an interrupt to a vector address (Section 4.4.1). (ii) Non-implementable opcode error causes an interrupt to a vector address in 80196.

Software error *exception* or *trap*-related sources cause the interrupt of an ongoing program computations in certain processors. Examples are the division by zero (also known as type 0 interrupt as it is also generated by a software interrupt instruction *INT 0* in 80x86) and overflow (also known as type 2 interrupt as it is also generated by Int 2 instruction) in 80x86. These two interrupts, types 0 and 2 are generated by the hardware within the ALU of the processor. Row 4 of Table 4.1 lists these interrupt sources. Example 4.8 explains a software-related *trap* or *exception*, which is an interrupt generated by the processor hardware on division by 0.

Example 4.8

Assume that a division by zero occurs during execution of a certain instruction of a program. An ISR is needed which must execute whenever the division by zero occurs. This ISR could be to display 'A division by zero error at ....' on the screen and then terminate or pause the ongoing program.

A user program under execution currently by the processor does not know when its ALU will issue this internal error flag (a hardware signal). The service routine executes by using an interrupt mechanism which is meant for service on a zero-division error-signal. On setting of the signal, an interrupt of the ongoing program happens just after completing the current instruction that is being executed, and then the ISR executes for postzero division tasks after resetting the flag.

Executing software error-related processor interrupts are needed to respond to errors such as division by zero or illegal opcode, which is detected by the processor hardware. These are called traps and some time also called exceptions. These are essential for handling run-time errors detected by the system hardware.

Software Instruction-Related Interrupts Sources A program can also *handle* specific computational errors or run-time conditions or signalling some condition. For instance, Example 4.6 showed the handling of negative number square root SWI, which is handled by SWI instruction in the instruction set of a processor. Processors provide for software instruction(s) related to the traps, signals or exceptions.

1. There are certain software instructions for interrupting and then diverting to the ISR also called the signal handler. These are used for signalling (or switching) to another routine from an ongoing routine or task or thread (Section 7.10). Figure 4.4(b) showed the signal generated by SWI and signal handling.
2. Software instructions are also used for trapping some run-time error conditions (called throwing exceptions) and executing exceptional handlers on catching the exceptions (Example 4.6).

An example of a software interrupt is the interrupt generated by a software instruction INT n in the 80x86 processor or SWI in ARM7. Row 5 of Table 4.1 lists these interrupt sources. SWI instruction differs from a function *call* instruction as follows.

1. Software interrupt in 68HC11 is caused by instruction, SWI.
2. There is a single-byte instruction INTO in 80x86. It generates type 0 interrupt, which means that the interrupt should be generated with the corresponding vector address 0x00000. Instead of the type 0 interrupt that 8086 and 80x86 hardware may also generate on a division by zero, the instruction INTO does exactly that.

3. There is another single byte 8086 and 80x86 instruction TYPE3 (corresponding vector address 0 × 00C0H). This generates an interrupt of type 3, called break point interrupt. This instruction is like a PAUSE instruction. PAUSE is a temporary stoppage of a running program. It enables a program to do some housekeeping, and return to the instruction after the break point by pressing any key.
4. There are another 80x86 two-byte instruction INT n, where n represents the type and is the second byte. This means 'generate type n interrupt' and processor hardware get the ISR address by computing by the vector address 0x00004 × n. When n = 1, it represents single-step trap in 8086 and 80x86.
5. There is another 80x86 instruction, which uses a flag called trap and is denoted by TF. This flag is at the FLAG register and EFLAG register of 8086 and 80x86, respectively. This means when TF sets (written '1'), automatically after every instruction, the processor action causes an interrupt of type 1 repeatedly. The processor fetches each time the ISR address from the vector address 0x00004 (same as type 1 interrupt address). INT 1 software instruction will also cause type 1 interrupt once but the TF flag set instruction action is identical to the action caused at the end of each instruction after type 1 interrupt.
6. There is instruction in 80196 called *Trap*. It enables debugging of instructions. Till the next instruction after the Trap is executed, no interrupt source can interrupt the process and cause diversion to ISR.

SWI-related details in the instruction set help in programming the program diversion to ISR on exception. The exceptional condition if occurs (sets) during execution, causes a diversion to the ISR called *exception handler* or *signal handler* using the software instruction for interrupt in the set.

A programmer can program for the exception on a *queue* (a memory buffer similar to a print buffer) getting full. This is an exceptional run-time condition. It should cause the diversion to routine called *exception* handler function that initiates the appropriate action. *Exceptions* are important routines for handling the run-time errors.

Software instruction-related or software-defined condition-related software interrupts are used in the embedded system. They are essential to design ISRs like error-handling ISRs, software timer-driving ISRs and signalling another routines to run. These interrupts are also called *traps* or *exceptions* or *signals*.

4.4 INTERRUPT SERVICING (HANDLING) MECHANISM

Each system has an interrupt servicing (handling) mechanism. The OS also provides for mechanism for interrupt-handling (Section 8.7).

4.4.1 Interrupt Vector

Interrupt vector is a memory address to which the processor vectors. The processor transfers the program counter to the interrupt vector new address on an interrupt. Using this address, the processor services that interrupt by executing corresponding ISR. The memory addresses for vectoring by the processor are processor- or microcontroller-specific. Vectoring is as per the provisions in interrupt-handling mechanism. The various mechanisms are as follows:

Processor Vectoring to the ISR_ VECTADDR On an interrupt, a processor vectors to a new address, ISR_VECTADDR. It means that the PC (program counter), which has the instruction address of next instruction, saves that address on stack or in some CPU register, called link register and the processor loads the ISR_VECTADDR into the PC. The stack pointer register of CPU provides the saved address to enable return from the ISR using the stack. When the PC saves at the link register it is part of the CPU register set. Section 4.6 will explain the mechanism for saving the CPU registers in detail. The ISR last instruction is RETI (return from interrupt) instruction.

A processor provides for one of the following ways of using the ISR_VECTADDR-based addressing mechanism.

Processor Vector Address

1. A system has internal devices like the on-chip timer and A/D converter. In a given microcontroller, each internal device interrupt source or source-group has a separate ISR_VECTADDR address. Each external interrupt pin has separate ISR_VECTADDR. An example is 8051. Figure 4.6(a) shows the ISR_VECTADDRs for the hardware interrupt sources. A very commonly used method is that the internal device (interrupt source or interrupt source group) in the microcontroller autogenerates the corresponding interrupt vector address, ISR_VECTADDR. Thus vector addresses are specific for specific microcontroller or processor with that internal device. An internal hardware signal from the device is sent for the interrupt source or source group.
2. In 80x86 processor architecture, a software instruction, for example, *INT* n explicitly also defines the *type* of interrupt and the *type* defines the ISR_VECTADDR. Figure 4.6(b) shows the ISR_VECTADDRs with different vector addresses for different interrupt types. This mechanism results in the handling of n number of exception handling routines or ISRs for n interrupt types.

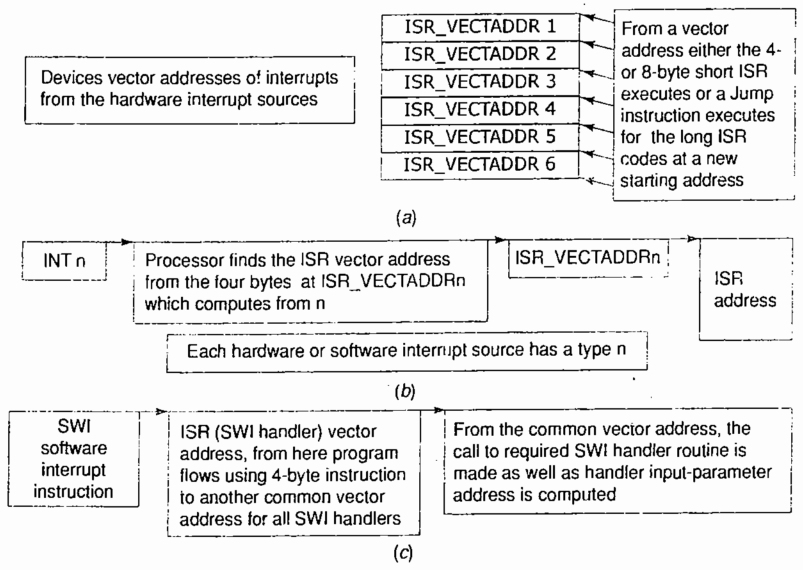

Devices vector addresses of interrupts from the hardware interrupt sources
ISR_VECTADDR 1
ISR_VECTADDR 2
ISR_VECTADDR 3
ISR_VECTADDR 4
ISR_VECTADDR 5
ISR_VECTADDR 6
From a vector address either the 4- or 8-byte short ISR executes or a Jump instruction executes for the long ISR codes at a new starting address
(a)
INT n
Processor finds the ISR vector address from the four bytes at ISR_VECTADDRn which computes from n
ISR_VECTADDRn
ISR address
Each hardware or software interrupt source has a type n
(b)
SWI software interrupt instruction
ISR (SWI handler) vector address, from here program flows using 4-byte instruction to another common vector address for all SWI handlers
From the common vector address, the call to required SWI handler routine is made as well as handler input-parameter address is computed
(c)

Fig. 4.6 (*a*) ISR_VECTADDRs for hardware interrupt sources (*b*) ISR_VECTADDRs with different vector address for different interrupt types using INT n instruction (*c*) The ISR_VECTADDR with common vector addresses for different exceptions, traps and signals using software interrupt instruction SWI

3. In ARM processor architecture, the software instruction SWI does not explicitly define the type of interrupt for generating different vector address and instead there is a common ISR_VECTADDR for each *exception* or *signal* or *trap* generated using SWI instruction. ISR that executes after vectoring has to find out which exception caused the processor to interrupt and divert the program. Such a mechanism in the processor architecture results in provisioning for the unlimited number of exception handling routines in the system all having the common interrupt vector address. Figure 4.6(c) shows the ISR_VECTADDR with common vector address for all *exceptions, traps* and *signals* resulting from SWI.

A group of Interrupt Sources having Common Vector Address A source group in the hardware may have the same ISR_VECTADDR.

Example 4.9

Consider 8051, TI (transmitter interrupt) and RI (receiver interrupt) are the sources in the same group having identical ISR_VECTADDR. TI is an interrupt that is generated when the serial buffer register for transmission completes serial transmission, and RI is when the buffer receives a byte from the serial receiver. ISR at the ISR_ADDR to which the program jumps or which is called from bytes at the ISR_ VECTADDR must first identify the interrupt source (whether TI or RI) in case of the identical vector address or ISR address for a group of sources. Identification is from a flag in the status register. Setting of a specific status flag in the device flag register enables identification of the interrupt source in the group by the ISR that runs after vectoring.

There are two types of handling mechanisms in processor hardware. The processor-handling mechanism provides for fetching into the PC either (i) the ISR instruction at the ISR_VECTADDR or (ii) the ISR address from the bytes at the ISR_VECTADDR.

1. There are some processors, which use ISR_VECTADDR directly as ISR address and the processor fetches the ISR instruction from there, for example, ARM or 8051. The ARM permits the use of 4-byte instruction for the jump to the ISR (routine for the interrupt servicing). Figure 4.7(a) shows the use of ISR_ VECTADDR in ARM for the jump to the routine for the interrupt servicing. The 8051 microcontroller permits the use of short ISR of maximum 8 bytes for the internal devices. The short ISR codes can also use a call instruction to call a detailed routine. Figure 4.7(b) and (c) shows the use of ISR_ VECTADDR in 8051 in case of short-code and long-code ISR, respectively.
2. There are some processors, which use ISR_VECTADDR indirectly as ISR address and the processor fetches the ISR address from the bytes saved at the ISR_VECTADDR, for example, 80x86. Figure 4.7(d) shows the use of ISR_ VECTADDR address in 8086. Processor of interrupt of type n vectors to address 0x00004 × n and fetches 16 bits for sending into IP (instruction pointer register) and another 16 bits for sending into CS (code segment register) The ISR for interrupt will execute from address 0x100000 × CS + IP.

Interrupt Vector Table System software designer must provide for specifying the bytes at each ISR_VECTADDR address. The bytes are for either ISR short code [Figure 4.7(b)] or jump instruction to the ISR first instruction [Figure 4.7(a)] or ISR short code with call to the full code of the ISR [Figure 4.7(c)] or for fetching the bytes for finding the ISR address [Figure 4.7(d)].

A table facilitates the service of the multiple interrupting sources for each internal device. Each row of table has an ISR_VECTADDR and the bytes are saved at each ISR_VECTADDR. Vector table location in the memory depends on the processor. It is located at the higher memory addresses, 0xFFC0 to 0xFFFB in 68HC11. It is at the lowest memory addresses 0x00000 to 0x003FF in 80x86 processor. It starts from the lowest memory addresses 0x00000000 in ARM7. Figure 4.8 shows a vector table in the memory in case of multiple interrupt sources or source groups.

An external device may also send to the processor the ISR_VECTADDR through the data bus (row 2, Table 4.1).

An interrupt vector is an important part of interrupts service mechanism, which associates a processor. The processor first saves the PC and/or other registers of CPU on interrupt and then loads a vector address into the PC. Vector address provides the ISR or ISR address to the processor for an interrupt source or a group of sources or for the given interrupt type. The interrupt vector table is an important part of interrupts service mechanism, which associates the system provisioning for the multiple interrupt sources and source groups.

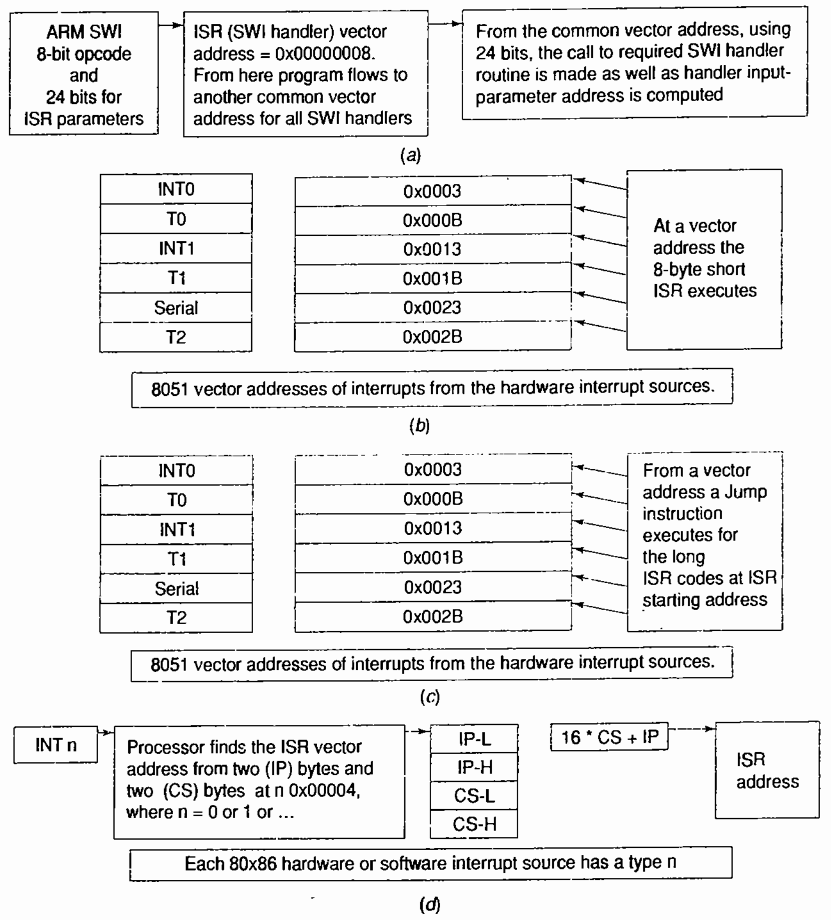

ARM SWI 8-bit opcode and 24 bits for ISR parameters
ISR (SWI handler) vector address = 0x00000008. From here program flows to another common vector address for all SWI handlers
From the common vector address, using 24 bits, the call to required SWI handler routine is made as well as handler input-parameter address is computed
(a)
INT0
T0
INT1
T1
Serial
T2
0x0003
0x000B
0x0013
0x001B
0x0023
0x002B
At a vector address the 8-byte short ISR executes
8051 vector addresses of interrupts from the hardware interrupt sources.
(b)
INT0
T0
INT1
T1
Serial
T2
0x0003
0x000B
0x0013
0x001B
0x0023
0x002B
From a vector address a Jump instruction executes for the long ISR codes at ISR starting address
8051 vector addresses of interrupts from the hardware interrupt sources.
(c)
INT n
Processor finds the ISR vector address from two (IP) bytes and two (CS) bytes at n 0x00004, where n = 0 or 1 or ...
IP-L
IP-H
CS-L
CS-H
16 * CS + IP
ISR address
Each 80x86 hardware or software interrupt source has a type n
(d)

Fig. 4.7 (*a*) Use of ISR_ VECTADDR in ARM for the jump to the routine for the interrupt servicing (*b*) Use of ISR_ VECTADDR in 8051 in case of short-code interrupt service routine (ISR) (*c*) Use of ISR_ VECTADDR in 8051 in case of long-code ISR (*d*) Use of ISR_ VECTADDR address in 80x86 processors

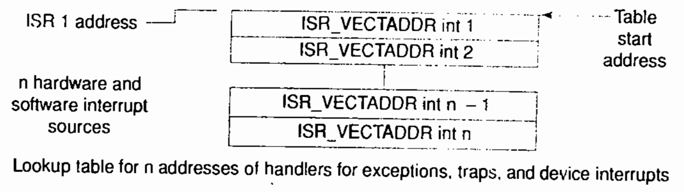

ISR 1 address
ISR_VECTADDR int 1
ISR_VECTADDR int 2
Table start address
n hardware and software interrupt sources
ISR_VECTADDR int n – 1
ISR_VECTADDR int n
Lookup table for n addresses of handlers for exceptions, traps, and device interrupts

Fig. 4.8 Vector table in memory in case of multiple interrupt sources or source groups

4.4.2 Classification of All Interrupts as Non-Maskable and Maskable Interrupts

Maskable sources of interrupts provide for masking (no diversion) and unmasking the interrupt services (diversion to the ISRs). Execution of ISR for each device interrupt source or source group can be masked or unmasked. An external interrupt request can also be masked. Execution of a software interrupt (trap or exception or signal) can also be masked. Most interrupt sources are maskable. A few specific interrupts cannot be masked. A few specific interrupts can be declared non-maskable within few clock cycles of the processor reset, else that is maskable. There are three types of interrupt sources in a system.

1. *Non-maskable*: Examples are RAM parity error in a PC and error interrupts like division by zero. These must be serviced.
2. *Maskable*: Maskable interrupts are those for which the service may be temporarily disabled to let higher priority ISRs be executed first uninterruptedly.
3. *Non-maskable only when defined so within few clock cycles after reset*: Certain processors like 68HC11 has this provision. For example, an external interrupt pin, XIRQ interrupt, in 68HC11. XIRQ interrupt is non-maskable only when defined so within few clock cycles after 68HC11 is reset.

4.4.3 Enabling (Unmasking) and Disabling (Masking) in Case of Maskable Interrupt Sources

There can be interrupt control bits in devices. There may be one bit EA (enable all), also called the primary-level bit for enabling or disabling the complete interrupt system. When a routine or ISR is executed by the codes in a critical section, an instruction DI (disable interrupts) is executed at the beginning of the critical section and another instruction EI (enable interrupts) is executed at the end of the critical section. DI instruction resets the EA (enable all) bit and EI instruction sets primary level bit denoted by EA (enable all). An example of a critical section code is as follows. Assume that an ISR transfers data to the printer buffer, which is common to the multiple ISRs and functions. No other ISR of the function should transfer the data to the print buffer, else the bytes at the buffer will be from multiple sources. Data shared by several ISRs and routines need to be generated or used by protecting its modification by another ISR or routine.

There may be multiple bits denoted by E_0, ... E_{n-1} for n source group of interrupts in case of multiple devices. These bits are called mask bits and are also called secondary-level bits for enabling or disabling specific sources or source-groups in the system. By appropriate instructions in the user software, a write to the primary enable bit and secondary level enable bits (or the opposite of it, mask bits) either all or a part of the total maskable interrupt sources are disabled.

Example 4.10

Consider a system in which there are two timers and each timer has an interrupt control bit. Timer interrupt control bits are ET0 and ET1. Consider a system in which there is an SI device and an interrupt control bit ES, common to serial transmission and serial reception. There is an EA bit to interrupt control for disabling all interrupts.

When EA = 0, no interrupt is recognized and timers as well as SI interrupts service is disabled. When EA = 1, ET0 = 0, ET1 = 1 and ES = 1, interrupts from timer 1 and SI are enabled and timer 0 interrupt is disabled (masked).

4.4.4 Status Register or Interrupt Pending Register

An identification of a previously occurred interrupt from a source is performed by one of the following:

1. A local-level flag (bit) in a status register, which can hold one or more status flags for the one or several of the interrupt sources or groups of sources.
2. A processor-interrupt service pending flag (boolean variable) in an interrupt-pending register (IPR), that sets by the source (setting by hardware) and auto-resets immediately by the internal hardware when at a later instant, the corresponding source service starts diversion to the corresponding ISR.

Example 4.11

Consider a system in which there are two timers and each timer has a status bit TF0 and TF1. Consider a system in which there is SI device there are the status bits TxEMPTY and RxReady for serial transmission completed and receiver data ready.

1. The ISR_T1 corresponding to timer 1 device reads the status bit TF1 = 1 in the status register to find that timer 1 has overflowed; as soon as the bit is read the TF1 resets to 0.
2. The ISR_T0 corresponding to timer 0 device reads the status bit TF1 = 0 in the status register to find that timer 0 has overflowed; as soon as the bit is read the TF0 resets to 0.
3. The ISR corresponding to the SI device is common for the transmitter and the receiver. The ISR reads the status bits TxEMPTY and RxReady in the status register to find whether a new byte is to be sent to the transmit buffer or whether the byte is to be read from the receiver buffer. As soon as the byte is read the RxReady resets and as soon as the byte is written into the SI for transmission, TxEMPTY resets.

Some processor hardware provide for use of status register bits and some IPR bits. The IPR and status registers differ as follows. The status register is read only. (i) A status register bit (an identification flag) is *read only*, and is cleared (auto-reset) during the *read*. An IPR bit either clears (auto-resets) on the service of the corresponding ISR or clears only by a *write* instruction for resetting the corresponding bit. (ii) An IPR bit can be set by a write instruction as well as by an interrupt occurrence that waits for the service. A status register bit is set by the interrupting source hardware only. (iii) An IPR bit can correspond to a pending interrupt from a group of interrupt sources, but identification flags (bits) are separate for each source among the multiple interrupts.

Properties of the interrupt flags are as follows. A separate flag for every identification of an occurrence from each of the interrupt sources must exist. The flag sets on occurrence of interrupt: (i) It is present either in the internal hardware circuit of processor or in the IPR or in the status register. (ii) It is used for a *read* by processor or instruction after a *write* by the interrupting source hardware. (iii) It *resets* (becomes inactive) as soon as it is *read*. This is auto-reset characteristic provided in most hardware designs in order to enable this flag to indicate the next occurrence from same interrupt source. (iv) If set at once, it does not necessarily mean that it will be recognized and serviced later using an ISR. When a mask bit corresponding to that interrupt is set, even if the flag sets, the processor may ignore it unless the mask (or enable) bit modifies later. This makes it possible to prevent an unwanted interrupt from being serviced.

Example 4.12

Consider a touch screen.

It generates an interrupt when a screen position is touched. A status bit b_t is also set. It activates a interrupt request (IRQ). From the status bit, which is set, the interrupting source is recognized among the sources group (multiple sources of interrupts from same device or devices). The ISR_VECTOR_{IRQ} and ISR_{IRQ} are common for all the interrupts at IRQ.

IRQ results in processor vectoring to an ISR_VECTOR_{IRQ}. Using ISR_VECTOR_{IRQ} when the ISR_{IRQ} starts, ISR_{IRQ} instruction reads the status register and discovers that bit b_t as set. It calls for service function (get_touch_position), which reads register R_{pos} for touched screen position information. This action of reading b_t also resets the b_t if the touch screen controller-processing element provides for auto-resetting of b_t. This enables next IRQ interrupt and thus reading next-position on next touch.

4.5 MULTIPLE INTERRUPTS

4.5.1 Multiple Interrupt Calls

When there are multiple interrupt sources, each occurrence of interrupt from a source (or source group) is identifiable from a bit in the status register and/or in the IPR (Section 4.4.4). There can be interrupt service calls in succession case higher priority interrupt sources activate in succession. Then return from high priorit ISR is to lower priority pending ISR.

Let us understand two processor interrupt service mechanisms for the case of multiple interrupts.

1. Certain processors do not provide for in-between routine diversion to higher priority interrupts and presume that all interrupts or interrupts of priority greater than the presently running routine are masked till the end of the routine. Figure 4.9(a) shows diversion to higher priority interrupts at the end of the present interrupt service routine only.
2. Certain processors permit in-between routine diversion to higher priority interrupts. Figure 4.9(b) shows the actions in such processors. These processors provide, in order to prevent diversion in-between, a mechanism as follows: There is provisioning for masking of all interrupts by a primary-level bit. These processors also provision selective diversion by provisioning for masking the interrupt service selectively by secondary-level bits (Section 4.4.3).

4.5.2 Hardware Assigned Priorities

There is assigned priority order by hardware. ARM7 provides for two types of external interrupt sources (requests), IRQs and FIQs (fast interrupt requests). 8051 provides for priority order in order of interrupt vector addresses. Lower address has highest and higher has the lower priority. Interrupts in 80x86 are assigned priority order according to interrupt-types. Interrupt of type 0 has highest priority and 255 has lowest assigned priority.

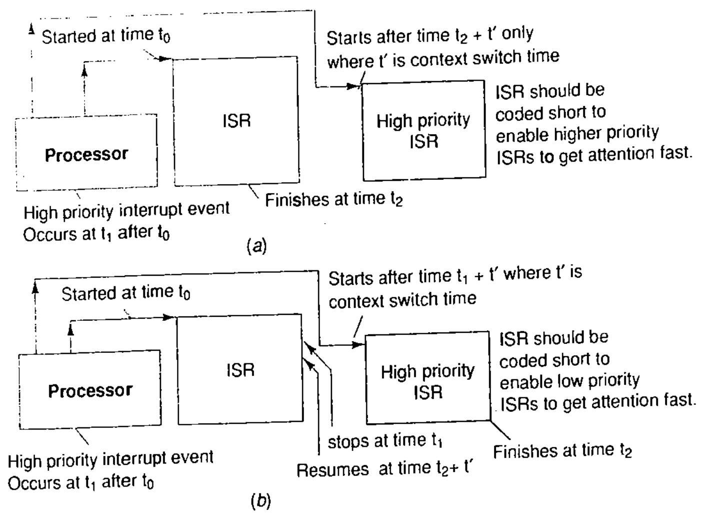

Started at time t_0
Starts after time t_2 + t′ only where t′ is context switch time
Processor
ISR
High priority ISR
ISR should be coded short to enable higher priority ISRs to get attention fast.
High priority interrupt event Occurs at t_1 after t_0
Finishes at time t_2
(a)
Started at time t_0
Starts after time t_1 + t′ where t′ is context switch time
Processor
ISR
High priority ISR
ISR should be coded short to enable low priority ISRs to get attention fast.
High priority interrupt event Occurs at t_1 after t_0
stops at time t_1
Resumes at time t_2+ t′
Finishes at time t_2
(b)

Fig. 4.9 (*a*) Diversion to higher priority interrupts at the end of the present interrupt service routine only (*b*) In-between routine diversion to higher priority interrupts unless all interrupts or interrupts of priority greater than the presently running routine are masked

When there are multiple sources of interrupts from the multiple devices, the processor hardware assigns to
each source (including traps or exceptions) or source group a preassumed priority (or level or type). Let us
assume a number, p_{hw} that represents the hardware-presumed priority for the source (or group). Let the number
be among 0, 1, 2, ..., k, ..., m − 1. Let p_{hw} = 0 mean the highest; p_{hw} = 1 next to highest........; p_{hw} = m − 1
assigned the lowest. Why does the hardware assign the presumed priority? Several interrupts occur at the
same time during the execution of a set of instructions, and either all or a few are enabled for service. The
service using the source corresponding to the ISRs can only be done in a certain order of priority. (There is
only one processor.) Assume that there are seven devices or interrupt source groups. The processor's hardware
can assign p_{hw} = 0, 1, 2, …, 6. The hardware service priorities will be in the order p_{hw} = 0, 1, 2, …, 6.
Software assigned priorities override these priorities, for example in 8051. Section 4.6.3 will explain this point.
Consider the example of the 80x86 family processor. Consider its six interrupt sources; division by zero,
single step, NMI (non-maskable interrupt from RAM parity error and so on), break point, overflow and print
screen. These interrupts are presumed to be of p_{hw} = 0, 1, 2, 3, 4 and 5, respectively. The hardware processor
assigns the highest priority for a *division by zero*. This is so because it is an exceptional condition found in
user software. The processor assigns the *single stepping* as the next priority as the user enables this source of
interrupt because of the need to have a break point at the end of each instruction whenever a debugging
software is run. NMI is the next priority because external memory *read* error needs urgent attention. Print
screen has the lowest priority.

Which is the Interrupt to be Serviced First among those Pending? Some Way of Polling Resolves this Question. The 8086 has a 'Vectored Priority Polling Method' A processor interrupt mechanism may internally provide for the number of vectors, ISR_VECTADDRs. The *vectored priority' method* means that the interrupt mechanism assigns the ISR_VECTADDR as well as p_{hw}. There is a

call at the end of each instruction cycle (or at the return from an ISR) for a highest priority source among those enabled and pending. Vectored priorities in 80x86 are as per the n_{type}. n_{type} = 0 highest priority and n_{type} = 0xFF (=255) lowest priority.

When there are multiple device drivers, traps, exceptions and signals as a result of hardware and software interrupts the assignment of priorities for each source or source group is required so that the ISRs of smaller deadline execute earlier by assigning them higher priorities. Hardware-defined priorities are used as default. Software assigned priorities override these priorities, for example, in 8051.

4.6 CONTEXT AND THE PERIODS FOR CONTEXT SWITCHING, INTERRUPT LATENCY AND DEADLINE

Getting an address (pointer) from where the new function begins, loading that address into the PC and then executing the called function's instructions will change a running function at the CPU to another. Before executing new instructions of the new function the processor or the OS also saves the current program's status word, registers and program contexts. If not done automatically by the processor or the OS, then the new functions, instruction should do that. This is because these (status word and registers) may be needed by the newly called function. *CPU registers including processor status word, registers, stack pointer and program current address in the PC define a function's context.* Figure 4.10(a) shows a current program context. What should exactly constitute the context? It depends on the processor or the operating system supervising the program.

The context must save if a function program or routine left earlier has to run again from the state which was left. When there is a *call* to a function (called *routine* in assembly language, *function* in C and C++, *method* in Java also called task or process or thread when it runs under supervison of the OS), the function or ISR or *exception*-handling function executes by three main steps.

1. Saving all the CPU registers including processor status word, registers and function's current address for next instruction in the PC. Saving the address of the PC onto a stack is required if there is no link register to point to the PC of left instruction of earlier function. Saving facilitates the return from the new function to the previous state.
2. Load new context to switch to a new function.
3. Readjust the contents of stack pointer and execute the new function.

These three actions are known as *context switching*. Figure 4.10(b) shows a current program's context switching to the new context.

The last instruction (action) of any routine or function is always a *return*. The following steps occur during return from the called function.

1. Before return, retrieve the previously saved status word, registers and other context parameters.
2. Retrieve into the PC the saved PC (address) from the stack or link register and load other part of saved context from stack and readjust the contents of stack pointer.
3. Execute the remaining part of the function, which called the new function.

These three actions are also known as *context switching*.

We can say that on interrupt or function call and return the context switches and a new program is executed whenever the new context loads into the processor CPU registers. Figure 4.10(c) and (d) shows context switching for new routine and another context switch on return or on in-between call to another routine. Nesting means one function calling the second which in turn calls the third and so on and the return to the calling functions will be in the reverse order. In case of function calls there is nesting and in the case of multiple ISRs because of the presence of multiple interrupts there may or may not no nesting.

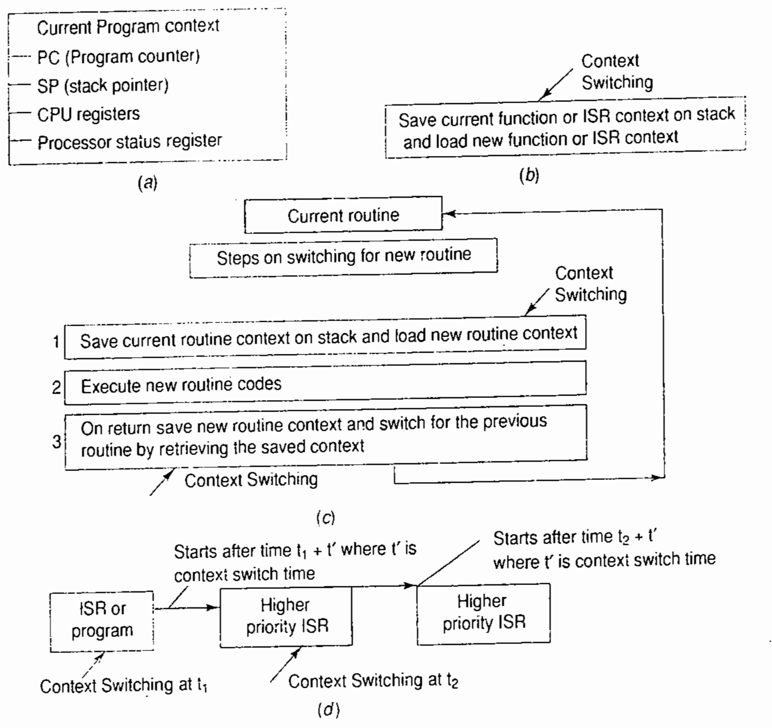

Current Program context
PC (Program counter)
SP (stack pointer)
CPU registers
Processor status register
(a)
Context Switching
Save current function or ISR context on stack and load new function or ISR context
(b)
Current routine
Steps on switching for new routine
Context Switching
1 Save current routine context on stack and load new routine context
2 Execute new routine codes
3 On return save new routine context and switch for the previous routine by retrieving the saved context
Context Switching
(c)
Starts after time t1 + t′ where t′ is context switch time
Starts after time t2 + t′ where t′ is context switch time
ISR or program
Higher priority ISR
Higher priority ISR
Context Switching at t1
Context Switching at t2
(d)

Fig. 4.10 (*a*) Current program context (*b*) New program executes with the new context of the called function or routine (*c*) Context switching for new routine and another switch on the return from routine (*d*) Context switching for new routine and another switch on return or in-between the call to another routine

Context switching means saving the context of the interrupted routine (or function) and then retrieving or loading the new context of the called routine. Example 4.13 shows how the context switching takes place in the ARM processor.

Example 4.13

Context switching is as follows in the ARM7 processor on *ISR call.* (i) The interrupt mask (disable) flags are set. (Disable low priority interrupts.) (ii) Next instruction PC is saved at link register. (iii) Current program status register (CPSR) copies into the saved program status register (SPSR) and CPSR stores the new status during new instructions. (iv) PC gets the new value as per the interrupt source from the vector table. An ISR return context switching back to the previous context is as follows. (i) PC is retrieved from link register. (ii) The corresponding SPSR copies back into the CPSR. (iii) The interrupt mask (disable) flags are reset. (Enable again the earlier disabled low priority interrupts.)

The time taken in context switching, T_{switch} has to be included in a period called *interrupt latency* period, T_{lat}. Example 4.14 shows how to calculate the context switching time period, which is to be accounted in calculating the interrupt latency (Section 4.6.1).

Example 4.14

1. ARM7 processor context switching's minimum period equals two clock cycles plus 0–20 clock cycles for finishing an ongoing instruction plus 0–3 cycles for aborting the data. The 0 cycle when an interrupt occurs just before the end and 3–20 when during an instruction. Longest time taken for an ARM instruction is 20 cycles.
2. During context switching for new routine call or for return. CPSR copies into SPSR on switching from a routine. CPSR means current program status register and SPSR means saved program status register. 3 cycles are taken in switching the CPSR.
3. Two clock cycles are needed for the start of the execution stage of switched routine's first instruction.

Aborting the processor data means CPSR not coping into an SPSR. Then step 2 three cycles are not taken up.

1. Minimum period is thus four (2 + 2) for data abort interrupt. [Steps 1 and 2 above]
2. Maximum is 27 clock cycles (2 + 20 + 3 + 2) for other than data abort interrupt. Maximum is when the interrupt occurs just at the start of execution of the longest time taking instruction in the processor. [Steps 1, 2 and 3 above]

Thus for any latency period calculation, 27 clock cycle periods as context switching time are taken into account when estimating latency in an ARM-based system.

Each running program has a context at an instant. Context reflects a CPU state (PC, stack pointer(s), registers and program state (variables that should not be modified by another routine). Context saving on the call of another ISR or task or routine is essential before switching to another context.

4.6.1 Interrupt Latency

When an interrupt occurs the service of the interrupt by executing the ISR may not start immediately by context switching. The interval between the occurrence of an interrupt and start of execution of the ISR is called interrupt latency.

1. When the interrupt service starts immediately on context switching the interrupt latency T_{switch} equals the context switching period. When instructions in a processor take variable clock cycles, maximum clock cycles for an instruction are taken into account for calculating the latency. Figure 4.11(a) shows latency in case the interrupt service starts immediately.
2. When the interrupt service does not start immediately but context switching starts after all the ISRs corresponding to the higher priority interrupts complete the execution. If the sum of time intervals for completing the higher priority ISRs equals ΣT_{exec}, then interrupt latency equals $T_{switch} + \Sigma T_{exec}$. Figure 4.11(b) shows latency in case the interrupt service starts after present ISR of higher priority interrupt completes the execution.
3. We disable the interrupt system when a routine enters a critical section and enable the interrupts when the routine exits the critical section codes. A routine of function or ISR may consist of codes for critical region instructions and before the critical section codes all the interrupts are disabled and enabled by the

end of the critical section. $T_{disable}$ may or may not be included depending on the programmer's approach. Let $T_{disable}$ be the period for which a routine is disabled in its critical section. The interrupt service latency from the routine with the interrupt-disabling instruction (because of the presence of the routine with critical section) for an interrupt source will be $T_{switch} + \Sigma T_{exec} + T_{disable}$. Figure 4.11(c) shows interrupt latency as sum of the periods for T_{switch}, ΣT_{exec} and $T_{disable}$ when the presently running routine to be interrupted is executing critical section codes.

Worst case latency is sum of the periods T_{switch}, ΣT_{exec} and $T_{disable}$ where the sum is for the interrupts of higher priorities only. Minimum latency is the sum of the periods T_{switch} and $T_{disable}$ when the interrupt is of the highest priority. For latency computations, worst case is taken into account.

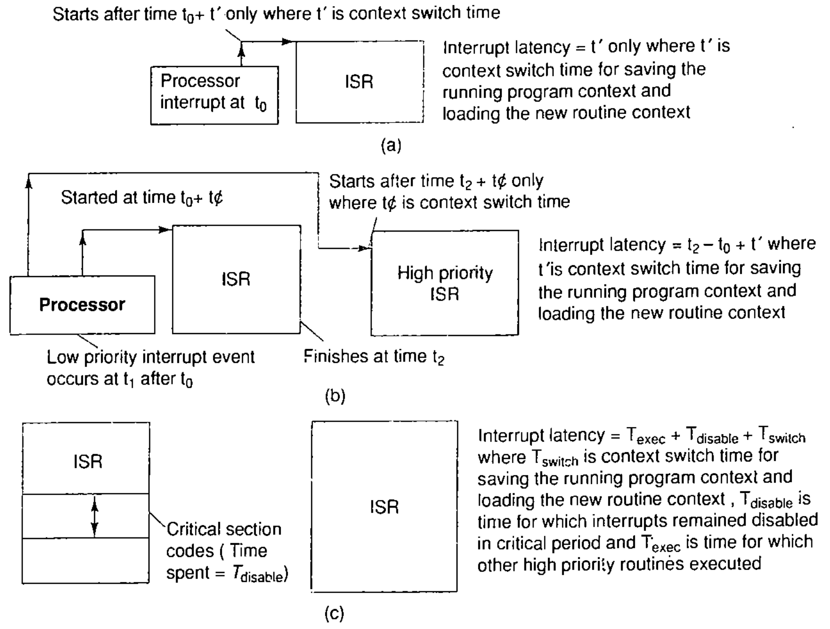

Starts after time t_0+ t′ only where t′ is context switch time
Processor interrupt at t_0
ISR
Interrupt latency = t′ only where t′ is context switch time for saving the running program context and loading the new routine context
(a)
Started at time t_0+ t¢
Starts after time t_2 + t¢ only where t¢ is context switch time
Processor
ISR
High priority ISR
Interrupt latency = $t_2 - t_0$ + t′ where t′is context switch time for saving the running program context and loading the new routine context
Low priority interrupt event occurs at t_1 after t_0
Finishes at time t_2
(b)
ISR
Critical section codes (Time spent = $T_{disable}$)
ISR
Interrupt latency = T_{exec} + $T_{disable}$ + T_{switch} where T_{switch} is context switch time for saving the running program context and loading the new routine context , $T_{disable}$ is time for which interrupts remained disabled in critical period and T_{exec} is time for which other high priority routines executed
(c)

Fig. 4.11 (*a*) Latency in case the interrupt service starts immediately (*b*) Latency in case the interrupt service starts only after the interrupt service routine presently running completes execution (*c*) Interrupt latency as sum of the periods for T_{switch}, ΣT_{exec} and $T_{disable}$ when the presently running low priority routine to be interrupted is having critical section codes

Example 4.15

80196 microcontroller has an SI device which has a FIFO (first in first out) buffer and the SI reads the bytes and puts it in the buffer. SI generates three interrupts: RI on one byte reception, FIFO_4thEntry interrupt when FIFO is half full and FIFO_Full interrupt when the FIFO is full. Assume that a microcontroller has two devices: SI similar to 80196 and timer T. SI has a serial input buffer of 8 bytes (a FIFO of 8 bytes

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.

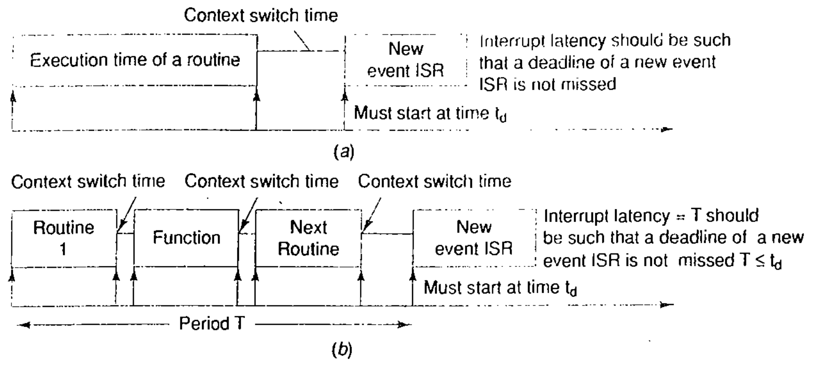

Context switch time
Execution time of a routine
New event ISR
Interrupt latency should be such that a deadline of a new event ISR is not missed
Must start at time t_d
(a)
Context switch time
Context switch time
Context switch time
Routine 1
Function
Next Routine
New event ISR
Interrupt latency = T should be such that a deadline of a new event ISR is not missed T ≤ t_d
Must start at time t_d
Period T
(b)

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.

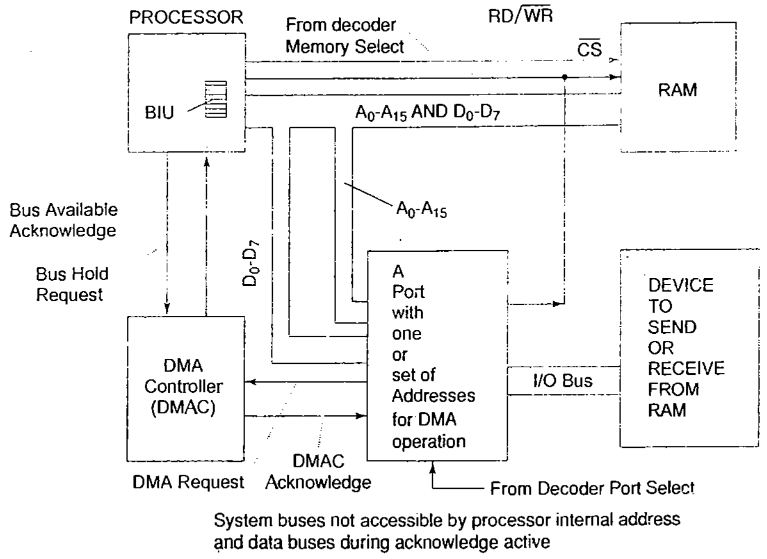

PROCESSOR
From decoder
Memory Select
RD/$\overline{WR}$
$\overline{CS}$
RAM
BIU
A_0-A_{15} AND D_0-D_7
A_0-A_{15}
D_0-D_7
Bus Available Acknowledge
Bus Hold Request
A Port with one or set of Addresses for DMA operation
DEVICE TO SEND OR RECEIVE FROM RAM
I/O Bus
DMA Controller (DMAC)
DMA Request
DMAC Acknowledge
From Decoder Port Select
System buses not accessible by processor internal address and data buses during acknowledge active

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).

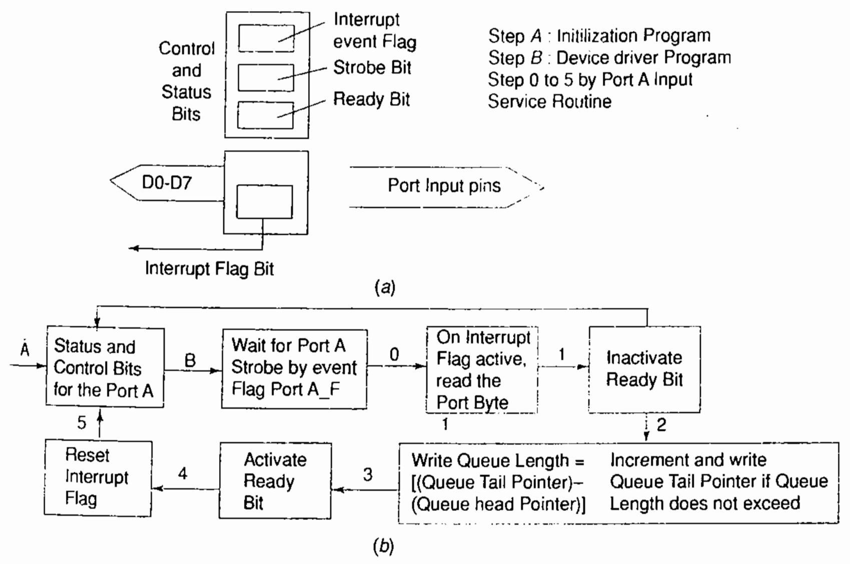

Control and Status Bits
Interrupt event Flag
Strobe Bit
Ready Bit
Step A : Initilization Program
Step B : Device driver Program
Step 0 to 5 by Port A Input Service Routine
D0-D7
Port Input pins
Interrupt Flag Bit
(a)
A
Status and Control Bits for the Port A
B
Wait for Port A Strobe by event Flag Port A_F
0
On Interrupt Flag active, read the Port Byte
1
Inactivate Ready Bit
2
Write Queue Length = [(Queue Tail Pointer)– (Queue head Pointer)]
Increment and write Queue Tail Pointer if Queue Length does not exceed
3
Activate Ready Bit
4
Reset Interrupt Flag
5
(b)

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.

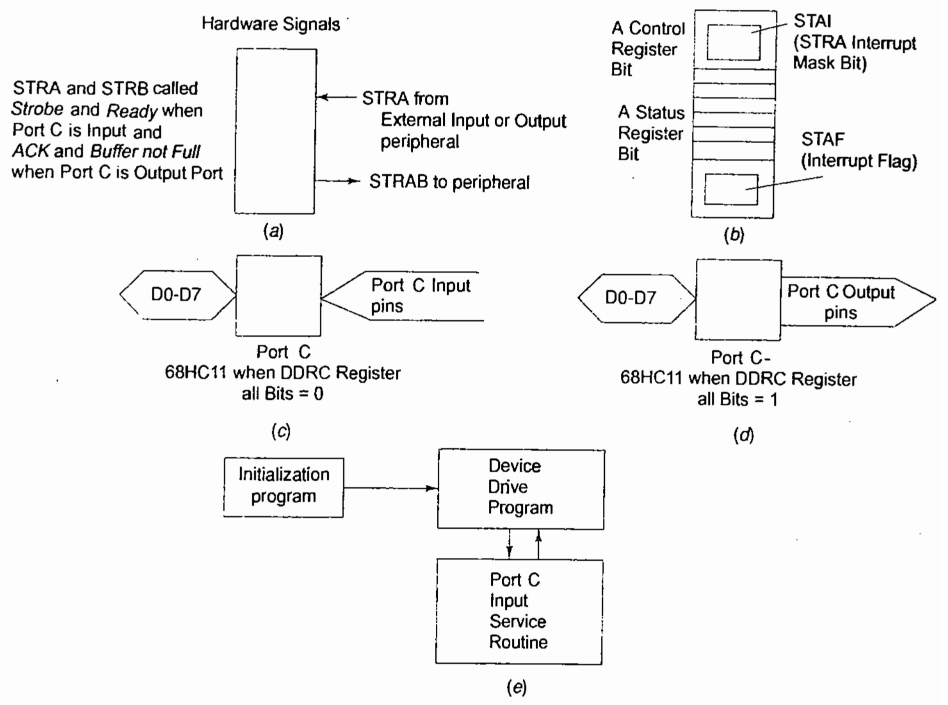

Hardware Signals
STRA and STRB called Strobe and Ready when Port C is Input and ACK and Buffer not Full when Port C is Output Port
STRA from External Input or Output peripheral
STRAB to peripheral
(a)
A Control Register Bit
STAI (STRA Interrupt Mask Bit)
A Status Register Bit
STAF (Interrupt Flag)
(b)
D0-D7
Port C Input pins
Port C
68HC11 when DDRC Register all Bits = 0
(c)
D0-D7
Port C Output pins
Port C-
68HC11 when DDRC Register all Bits = 1
(d)
Initialization program
Device Drive Program
Port C Input Service Routine
(e)

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 arge 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 unctions. 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, efer *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. (Hardware signal is different.)
Software-assigned priority	:	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?

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*.

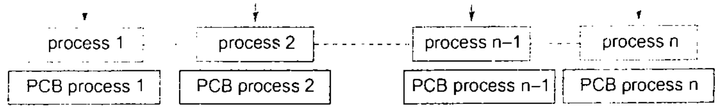
Example 7.1

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.

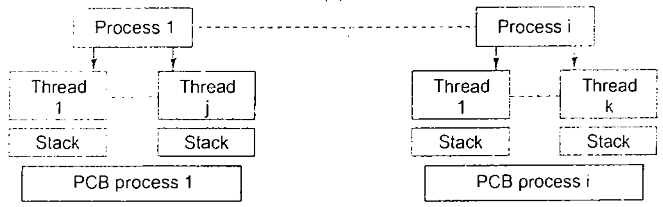
Process Control Block PCB is a data structure having the information using which the OS controls the process state. The PCB stores in the protected memory addresses at kernel. The PCB consists of following information about the process state.

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.

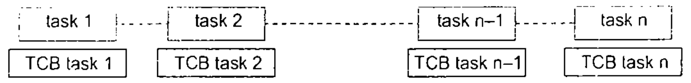


(a)



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.

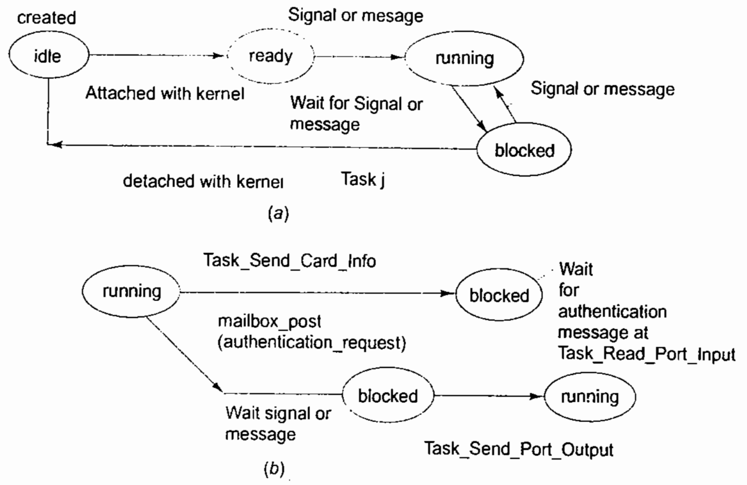


Fig. 7.2 (*a*) Task and its states (*b*) States of the task Task_Send_Card_Info in Example 7.4

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.

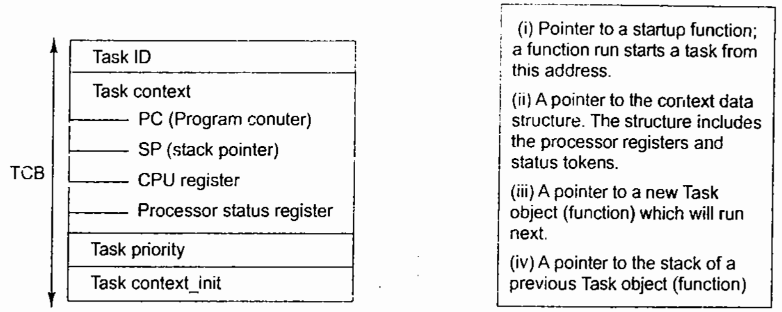
7.5 TASK AND DATA

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 another 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.



TCB
Task ID
Task context
PC (Program conuter)
SP (stack pointer)
CPU register
Processor status register
Task priority
Task context_init
(i) Pointer to a startup function; a function run starts a task from this address.
(ii) A pointer to the context data structure. The structure includes the processor registers and status tokens.
(iii) A pointer to a new Task object (function) which will run next.
(iv) A pointer to the stack of a previous Task object (function)

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 pre-emption 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)

Function	*ISR*	*Task*
5. ***Structure***: A function or a number of functions can be the subunits of a process or thread or task or ISR or subunit of another function or main function.	An ISR is independent and can be considered as a function. which runs on an event due to the interrupting source. A pending interrupt is scheduled to run using an interrupt-handling mechanism in the OS. The system, during running of an ISR, can let another higher priority ISR run.[2]	A task is independent and can be considered as a function, which is called to run by the OS scheduler using a context switching and task-scheduling mechanism of the OS. The system, during running of a task, can let another higher priority task run. The kernel manages the task scheduling.
6. ***Global variables use***: Function can change the global variables. The interrupts must be disabled and after completing the use of the global variable the interrupts are enabled (Section 7.8).	When using a global variable, the interrupts must be disabled and after completing the use of the global variable the interrupts are enabled.	When using a global variable, either the interrupts are disabled and after completing the use of global variable the interrupts are enabled or the semaphores are used as mutex critical sections (Sections 7.7.2, 7.7.5 and 7.8.3).
7. ***Posting and sending parameters***: Function can get the parameters and messages through the arguments passed to it or global variables, reference to which is made. Function returns the results of the operations through the references in the arguments and through return as per reference date type defined for it.	ISR using IPC functions for post can send (post) the signals (Section 7.7.1) and messages (Sections 7.10 to 7.16), for example, OSSemPost ( ). ISR cannot use the mutex protection of the critical sections. ISR does not for signal or message during running.	Task can send (post) the signals and messages and the task can wait for the signals and messages using the IPC functions, for example, OSSemPost ( ) and OSSemPend ( ) (Section 7.7). Task can use the mutex protection of the code sections. (Sections 7.7.2, 7.7.4 and 7.8.3).

-Depending on the scheduling of the real-time operating system (RTOS) by the kernel two common methods are: (i) cooperative scheduling, (ii) pre-emptive scheduling (Refer to Sections 8.10.1 to 3).
Either the interrupt handler or RTOS interrupt handler functions control the ISR scheduling. It depends on how the kernel or interrupt-handling hardware manages the ISRs (Section 8.7).

igure 7.4(a) shows a characteristic feature of the nested function calls in a program. Figure 7.4(b) shows the C assignments at different times on the nested calls.

7.7 CONCEPT OF SEMAPHORES

.7.1 Use of a Semaphore as an Event-Signalling or Notifying Variable

uppose that there are two trains. Assume that they use an identical track. When the first train A is to start on the ack, a signal or token for A is set (true/taken) and the signal or token for the other train, B is reset (false released).
OS provides for the use of a semaphore for signalling or notifying of a certain action and for notifying the cceptance of the notice or signal. Let a binary Boolean variable, *s*, represent the semaphore. The operations n the variable s signals or notifies the operations for communicating the occurrence of the event and for ommunicating taking note of the event. It is like a token. Release of a token is the occurrence of an event and ie acceptance of the token is taking note of that event.
Let us assume that the *s* increments from 0 to 1 for signalling or notifying occurrence of an event from a

section of codes in a task or thread. When the event is taken note by a section in another task waiting for that event, the s decrements from 1 to 0 and the waiting task codes start at another action.

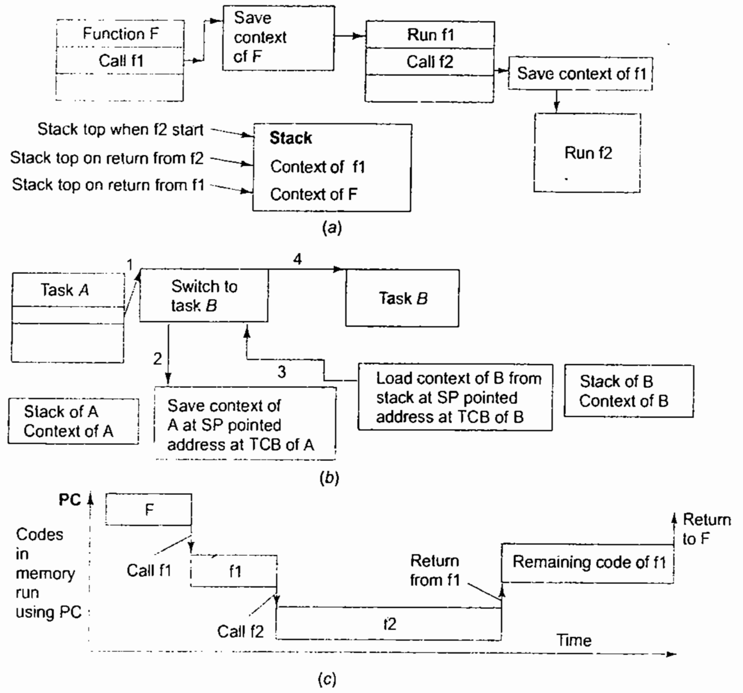


Fig. 7.4 (*a*) Actions on the function calls in a program (*b*) Action on pre-emption of task *A* by *B* or switch to task *B* during program run (*c*) Program counter assignments to the various functions in a process on the nested calls to the functions

A semaphore is called *binary* semaphore when its value is 0, it is assumed that it has been taken (or accepted), and when its value is 1, it is assumed that it has been released (or sent or posted) and no task has taken it yet.
An ISR can release the token. A task can release the token as well accept the token or wait for taking the token (row 7 Table 7.1).
Following is an example how to use a binary semaphore for signalling or notifying occurrences of an event from a task or thread and for signalling or notifying another task waiting for that event.

Example 7.7

Consider an ACVM (Section 1.10.2) *Chocolate delivery task* (Examples 7.5 and 7.6). After the task delivers the chocolate, it has to notify to the display task to run a waiting section of the code to display '*Collect the nice chocolate. Thank you, visit again*'. The waiting section for the display of the thank you message takes this notice and then it starts the display of thank you message.

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.]

The task *B* can access the same resource using OSSemPend ( ) if it is waiting for that semaphore. The task *B* posts the semaphore using OSSemPost ( ) after completing the access to that resource.

Figure 7.5(a) shows the use of a semaphore between *A* and *B*. It shows the five sequential actions at five different times, T0, T1, T2, T3 and T4. Figure 7.5(b) shows the timing diagram of the tasks in the running states as a function of time. It marks the five sequential actions at five different times, T0, T1, T2, T3 and T4.

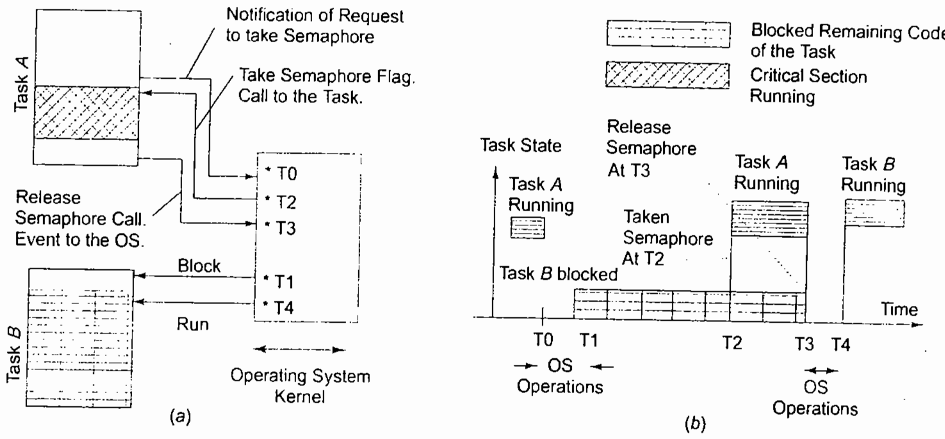


Fig. 7.5 (*a*) Use of a semaphore between tasks, *A* and *B*. It shows the five sequential actions at five different times, T0, T1, T2, T3 and T4 (*b*) Timing diagram of the tasks in the running states as a function of time. It marks the five sequential actions at five different times, T0, T1, T2, T3 and T4, and shows the use of a semaphore between tasks *A* and *B* by the operating system functions

An ISR can't be used to wait for the resource key since an ISR can just release the key. A task on the other hand can release the key as well accept the key or wait for taking the key. (row 7 Table 7.1)

Example 7.8

Consider an (Section 1.5.5) *Update_Time task*. When the task for updating time t on a system clock tick interrupt I_S starts, it has to notify that it is writing the t in a *time device* and that the t is changing. After the *Update_Time task* updates t information at the time device on I_S, it has to notify to the *Read_Time* task to run a waiting section of the code to read t from the *time device*. After the *Read_Time* reads t, it has to notify to *Update_Time task* to make note of it.

Assume OSSemPost ( ) is an IPC function at OS for posting a semaphore and assume OSSemPend ( ) is another IPC function for waiting the semaphore. Let *supdateT* be the binary semaphore pending and posted at *Update_Time* and pending and posted at *Read_Time* section for reading t. Let *supdateT* initial value = 1. The following will be the codes.

```
static void Task_ Update_Time (void *taskPointer) {
.
```

while (1) {

.

OSSemPend (*supdateT*) /* Wait the semaphore *supdateT*. This means that when wait over, an OS function decrements *supdateT* in corresponding event control block and *supdateT* becomes 0 at T2. */
/* Codes for writing date and time into the time device. */

.

OSSemPost (*supdateT*) /* Post the semaphore *supdateT*. This means that OS function increments *supdateT* in corresponding event control block and *supdateT* becomes 1 at T3. */

.

};
static void Task_ *Read_Time (void *taskPointer)* {

.

while (1) {

.

OSSemPend (*supdateT*) /* Wait for the semaphore *supdateT*. This means that task waits till *supdateT* is posted and becomes 1. When *supdateT* becomes 1 and the OS function then decrements *supdateT* in corresponding event control block and *supdateT* becomes 0 at T4. Task then runs further the following code*/
/* Code for reading the date and time at time device */

.

.

OSSemPost (*supdateT*) /* Post the semaphore *supdateT*. This means that OS function increments *supdateT* in corresponding event control block. *supdateT* becomes 1. */
};

Mutex When a binary semaphore is used to at beginning and end of critical sections in two or more tasks uch that at any instance only one section code can run, then the semaphore is called *mutex*. (mutex word is lerived from mutually exclusive). Example 7.8 showed that Task_ *Update_Time* and Task_ *Read_Time* uses the emaphore *supdateT* as mutex and at any instance either code section in Task_ *Update_Time* or code section in 'ask_ *Read_Time* runs and has exclusive access to the time device date and time variables. Mutex helps in etting access to a file or network or printer by multiple tasks at distinct instances because at an instance only odes for one of the tasks will be run to send the data to the file or network or printer. Section 7.8.3 will discuss s help in sharing data between the tasks.

In certain OS, a semaphore as a resource key is called mutex when the semaphore takes care of priority nversion problem also. Section 7.8.5 explains the inversion problem. In certain OS, a semaphore as a resource ey is called mutex even when the semaphore does not take care of the priority inversion problem. In certain)S, a semaphore as a resource key is called mutex and the option is provided to programmer for using priority nversion safe or without inversion safe mutex.

1. Semaphore provides a mechanism to let a section of the task code or a task wait till another task section finishes another set of codes such that these sections use a common resource, device or file or variable.
2. When a semaphore is waiting (for taking or accepting) by a task code, then that task has the access to the necessary resources when semaphore is 'given' (sent or posted), the resources unlock.
3. Semaphore can be used as a resource key. Resource key is one that permits the use of resources like CPU, memory or other functions or critical section codes.
4. Binary semaphore can be used as a mutex as well as event notifying flag.

7.7.3 Use of Multiple Semaphores for Synchronizing the Tasks

OS semaphore functions are provided for multitasking operations. Figure 7.6 shows an example of the use of two semaphores for synchronizing the tasks *I*, *J* and *M* and the tasks *J* and *L*. Example 7.9 gives another example in which *I*, *J*, *K* and *L* are synchronized to run sequentially.

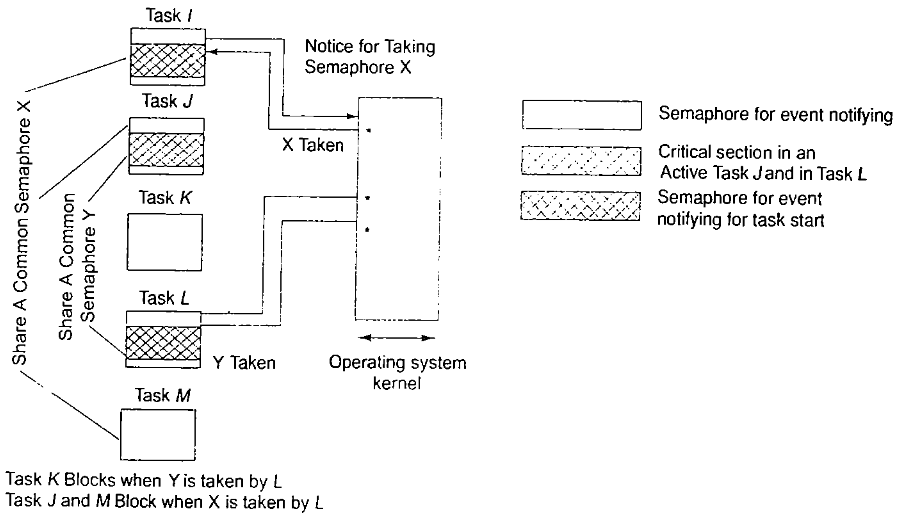

Task *I*
Notice for Taking Semaphore X
Task *J*
X Taken
Task *K*
Task *L*
Y Taken
Task *M*
Operating system kernel
Share A Common Semaphore X
Share A Common Semaphore Y
Semaphore for event notifying
Critical section in an Active Task *J* and in Task *L*
Semaphore for event notifying for task start
Task *K* Blocks when Y is taken by *L*
Task *J* and *M* Block when X is taken by *L*

Fig. 7.6 An example of the use of two semaphores —one for synchronizing the tasks *I*, *J* and *M* and other for tasks *J* and *L*

Example 7.9

A semaphore can let one task run among many in the initiated into a list with *state* = '*ready*' or '*running*' run at an instance while others are waiting. Let the following be the codes.

Assume OSSemPost ( ) is an OS IPC function for posting a semaphore and assume OSSemPend ( ) is another OS IPC function for waiting for the semaphore. Let *sTask* be the binary semaphore pending and posted at each task to let another run. Let *sTask1* initially be 1 and *sTask2 sTask3* and *sTask4* initially be 0.

The following will be the codes and the first task *I* will run, then *J*, then *K*, then *L*, then *I* when at that instance *sTask1* = 1 and *sTask2 sTask3, sTask4* = 0.

static void Task_ *I* (void *taskPointer) {

.

while (1) {
OSSemPend (sTask1) /* wait for semaphore sTask1 and when wait over then an OS function decrements sTask1 in corresponding event control block and sTask1 becomes 0 */
/* Codes for Task_*I* */

.

```
OSSemPost (sTask2) /* Post the semaphore sTask2. This means that OS function increments
     sTask2 in corresponding event control block. sTask2 becomes 1 */
};
static void Task_ J (void *taskPointer) {
.
while (1) {
OSSemPend (sTask2) /* Wait for the semaphore sTask2. This means that task waits till sTask2is posted
     and becomes 1. When sTask2 becomes 1 and the OS function is to decrements sTask2 in corresponding
     event control block. sTask2 becomes 0. Task then runs further the following code*/
/* Code for Task J */
.
.
OSSemPost (sTask3) /* Post the semaphore sTask3. This means that OS function increments sTask3 in
     corresponding event control block. sTask3 becomes 1. */
};
static void Task_ K (void *taskPointer) {
.
while (1) {
OSSemPend (sTask3) /* Wait for the semaphore sTask3. This means that task waits till sTask3 is posted
     and becomes 1. When sTask3 becomes 1 and the OS function is to decrements sTask3 in corresponding
     event control block. sTask3 becomes 0. Task then runs further the following code*/
/* Code for Task K */
.
.
OSSemPost (sTask4) /* Post the semaphore sTask4. This means that OS function increments sTask4 in
     corresponding event control block. sTask4 becomes 1. */
};
static void Task_ L (void *taskPointer) {
. .
while (1) {
OSSemPend (sTask4) /* Wait for the semaphore sTask4. This means that task waits till sTask4 is posted
     and becomes 1. When sTask4 becomes 1 and the OS function is to decrements sTask3 in
     corresponding event control block. sTask4 becomes 0. Task then runs further the following code*/
/* Code for Task J */
.
.
OSSemPost (sTask1) /* Post the semaphore sTask1. This means that OS function increments
     sTask1 in corresponding event control block. sTask1 becomes 1. */
};
```

For example, when a task *K* is to start running, it takes the semaphore sTask3. The OS blocks the tasks *I*, *J* and *L*. A task *L* waits for the release of the semaphore by *K*.

Number of tasks Waiting for the Same Semaphore An RTOS has the answer to the following: when a number of tasks has the same semaphore waiting then which of them takes the semaphore? In certain OS, a semaphore is given to the task of highest priority among the waiting tasks. In certain OS, a semaphore is given to the longest waiting task in the FIFO mode. In certain OS, a semaphore is given to select an option and the option is provided for priority or FIFO mode. The task having priority if started, takes a semaphore first in case the priority option is selected. The task pending since a longer period takes a semaphore first in case the FIFO option is selected.

1. Multiple semaphores are used and different set of semaphores can share among different set of tasks.
2. Semaphore provides a mechanism to synchronize the task codes. Multiple semaphores can be used in multitasking system.

7.7.4 Counting Semaphores

An OS may provide for the counting semaphores. A counting semaphore can be an unsigned 8- or 16- or 32-bit integer. A value of counting semaphore controls the blocking or running of the codes of a task. The counting semaphore decrements each time it is taken. It increments when released by a task.

The value of counting semaphore at an instance reflects the initialized value minus the number of times it is taken plus the number of times released. The counting semaphore can be considered as the number of tokens present and the waiting task will not wait and run further if at least one token is present. The use of a semaphore is such that one of the task thus waits to execute the codes or waits for a resource till at least one token is found.

Assume that a task can send the stacks on a network into eight sequentially transmitting buffers. Each time the task runs it takes the semaphore and sends the stack into a buffer, which is next to the earlier one. Assume that a counting semaphore *scnt* is initialized = 8. After sending the data of the stack, the task takes the *scnt* and *scnt* decrements. When a task tries to take the *scnt* when it is 0, then the task blocks and cannot send stack into any buffer.

Example 7.10

Consider an ACVM (Section 1.10.2). Consider *Chocolate delivery task*. It cannot deliver more than the total number of chocolates, *total* loaded into the machine. Assume that a semCnt is initialized equal to the *total*. Each time, the new chocolates are loaded in the machine, semCnt increments by the number of new chocolates added. The *Chocolate delivery task* can be coded as follows.

```
static void Task_Deliver (void *taskPointer) {
.
while (1) { /* Start an infinite while-loop. */
/* Wait for an event indicated by an IPC from Task Read-Amount */
.
OSSemPend (semCnt) /* If chocolate available is true then the task takes the semaphore if
semCnt is 1 or > 1 (which means is not 0) and decrement the semCnt and continue remaining
operations */
.
};
```

Counting semaphore can be consider as an unsigned integer semaphore that can be 'taken' till its value = 0 and is initialized to a high value. It can also be 'given' a number of times.

7.7.5 P and V SEMAPHORES

An OS may provide for an efficient synchronization mechanism, called P and V semaphores in a standard, called POSIX 1003.1.b, an IEEE standard (POSIX stands for portable OS interfaces in Unix). OS semaphore functions P and V represent the semaphore by integer variables. A semaphore variable, apart from initialization, is accessed only through two standard atomic-operations: P and V. [P (for *wait* operation) is derived from a Dutch word 'Proberen', which means 'to test'. V (for *signal* notifying operation) is derived from the word 'Verhogen' which means 'to increment'.] (Atomic-operation is one, which can not be in parts.)

1. P semaphore function signals that the task requires a resource and if not available waits for it.
2. V semaphore function signals from the task to the OS that the resource is now free for the other users.

Consider P semaphore. It is a function, P (&*sem_1*) which, when called in a process, does the following operations using semaphore, *sem_1*.

1. /* Decrease the semaphore variable*/
sem_1 = *sem_1* –1;
2. /* If sem_1 is less than 0, send a message to OS by calling a function waitCallToOS. Control of the process transfers to OS, because less than 0 means that some other process has already executed P function on sem_1. Whenever there is return to the OS, it will be to step 1. */
if (*sem_1* < 0){waitCallToOS (*sem_1*);}.

Consider V semaphore. It is a function, V(&*sem_2*) which, when called in a process, does the following operations using semaphore, *sem_2*.

3. /* Increase the semaphore variable*/
sem_2 = *sem_2* + 1;
4. /* If sem_2 is less or equal to 0, send a message to OS by calling a function signalCallToOS. Control of the process transfers to OS, because < or = 0 means that some other process has already executed P function on sem_2. Whenever there is return to the OS, it will be to step 3. */
if (*sem_2* < = 0){signalCallToOS (*sem_2*);}

Use of P and V Semaphore Functions with a Signal or Notification Property P and V functions can represent a signalling or notifying variable, sem_s when used as shown in Example 7.11.

Example 7.11

Let sem_s be a semaphore variable. Let it function as a signal or notifying an event using variable sem_s. P and V semaphore functions are used in two processes, task 1 and task 2 as follows.

Process 1 (Task 1)
```
while (true) {
/* Codes */
.
.
.
.
V (&sem_s);
.
.
};
```

Process 2 (Task 2)
```
while (true) {
/* Codes */
.
P (&sem_s);
/* The following codes will execute only when
sem_s is not less than 0. */
.
/* Continue Process 1 if sem_s is not equal to 0
or not less than 0. It means that no process is
executing at present. */
.
};
```

Use of P and V Semaphore Functions with a Mutex Property P and V functions can represent a mutex semaphore variable, sem_m when used as explained in Example 7.12.

Example 7.12

Let sem_1 and sem_2 be the same variable, *sem_m*. The latter functions as a mutex, as follows, when P and V semaphore functions are used in two processes, task 1 and task 2.

Process 1 (Task 1)
```
while (true) {
/* Codes before a critical region*/
.
.
.
/* Enter Process 1 Critical region codes*/
P (&sem_m);
/* The following codes will execute only when
sem_m is not less than 0. */
.
.
.
/* Exit Process 1 critical region codes */
V (&sem_m);
/* Continue Process 1 if sem_m is not equal to 0
or not less than 0. It means that no process is
waiting and has executed P function using
sem_m at present. */
.
.
.
};
```

Process 2 (Task 2)
```
while (true) {
/* Codes before a critical region*/
.
.
.
/* Enter Process 2 Critical region codes*/
P (&sem_m);
/* The following codes will execute only when
sem_m is not less than 0. */
.
.
.
/* Exit Process 2 critical region codes */
V (&sem_m);
/* Continue Process 2 if sem_m is not equal to 0
or not less than 0. It means that no process is waiting
and executed P function using sem_m at present.
*/
.
.
.
};
```

The same variable, sem_m, is shared between process 1 and process 2. Its use is in making both processes gain mutually exclusive access to the resource (CPU). Either process 1 runs after executing P or process 2 runs after executing P. Also, either process 1 runs after executing V or process 2 runs after executing V.

Figure 7.7(a) shows the use of P and V semaphores and the task *I*, task *J* and the scheduler. When a task takes a semaphore P, if sem_m = 'true' (=1) earlier then it becomes 'false' (=0) and task run continues as sem_m is not less than 0. When a task executes V, if sem_m was 'false' earlier then it sets to 'true' and the task continues running, else the task blocks and waits for the execution of another task. Figure 7.7(b) shows the PC assignments to a process or function when using P and V semaphores.

Use of P–V Semaphore Functions with a Counting Semaphore Property Let there be a process (task *c*). The P function decrements the count and the function increments the counts. The P function operates on a counting semaphore, sem_c1 as shown in Example 7.13.

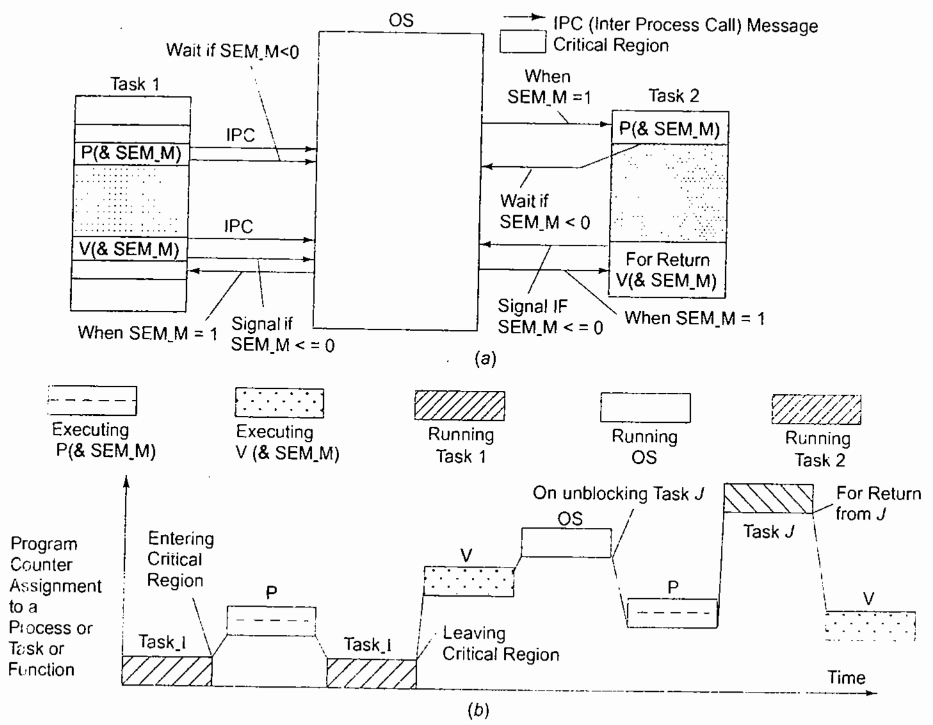


Fig. 7.7 (*a*) Use of P and V semaphores at a task 1, and at another task 2 and at a scheduler (*b*) The program counter assignments to a process or function when using P and V semaphores

Example 7.13

Assume a processes using P semaphore functions in task, task_c. Let *sem_c1* be a counting semaphore variable and represent the number of empty places created by the process c. P functions operate on these and reduces the number of empty places as follows:

```
Process c (Task_c)
   while (true) {
   /* Codes on entering a producer region*/
   .
   .
```

```
      .
      .
      .
      .
      /* After exiting the producer Process 3 region codes */
      P (&sem_c1);
      /* Continue Process c if sem_c1 is not less than 0. */
      .
      .
      .
      };
```

Use of P and V Semaphore Functions with a Counting Semaphore Property for Bounded Buffer Problem Solution Let a task generate the outputs for use by another task. A task can have a separate counting semaphore.

Consider three examples:

(i) a task transmits bytes to an I/O stream for filling the available places at the stream;
(ii) a process '*writes*' an I/O stream to a printer buffer and,
(iii) a task of producing chocolates is being performed.

In example (i) another task reads the I/O stream bytes from the filled places and creates empty places.

In example (ii), from the print buffer an I/O stream prints after a buffer-read and after the printing, more empty places are created.

In example (iii) again, as consumer consumes the chocolates produced more empty places (to stock the produced chocolates) are created.

A task blockage operational problem is commonly called producer–consumer problem. A task cannot transmit to the I/O stream if there are no empty places in the stream. The task cannot write from the memory at the print buffer if there are no empty places at the print buffer. The *producer cannot produce if there are no empty places at the consumer stock.*

A classic program for synchronization is called the *producer–consumer problem program*. It is also called *bounded buffer problem program*. Here, one or more producers (task or thread processes) create data outputs that are then processed by one or more consumers (tasks or processes). The data outputs from the producers are passed for processing by the consumers using some type of IPC (Section 7.8) that uses a shared memory and counting semaphores (or message queues or mailboxes).

Let there be two processes (tasks 3 and 4). The P and V functions operate on two shared counting semaphores, sem_c1 and sem_c2, as in Example 7.14.

Example 7.14

Assume two processes using P and V semaphore functions and two tasks, tasks 3 and 4. Let *sem_c1* and *sem_c2* be two counting semaphore variables and represent the number of filled places created by the process 3 and a number of empty places created by process 4, respectively. P and V functions operate on these as follows.

```
Process 3 (Task 3)                    Process 4 (Task 4)
while (true) {                        while (true) {
```

```
/* Codes before a producer region*/
.
.
.
/* Enter Process 3 Producing region codes*/
P (&sem_c2);
/* The following codes will execute only when
sem_c2 (number of empty places)is not less
than 0. */
.
.
.
/* Exit Process 3 region codes */
V (&sem_c1);
/* Continue Process 3 if sem_c1 is not equal to 0
or not less than 0. */
.
.
.
};
```

```
/* Codes before a consumer region*/
.
.
.
/* Enter Process 4 Consuming region codes*/
P (&sem_c1);
/* The following codes will execute only when
sem_c1 (number of filled places) is not less
than 0. */
.
.
.
/* Exit Process 4 region codes */
V (&sem_c2);
/* Continue Process 4 if sem_c2 is not equal to 0
or not less than 0. It means that filled places are
still available at present. */
.
.
.
};
```

Two semaphores, sem_c1 and sem_c2 are shared between processes 3 and 4. When process 3 executes, it first reduces the number of empty places at process 4. When process 3 completes production, it increases the number of filled places at process 3. When process 4 consumes, it first reduces the number of filled places at process 3. When process 4 completes consumption, it increases the number of empty places at process 4. Either process 3 produces output after executing P, or process 4 consumes (uses) inputs after executing P. Also either process 3 proceeds after executing V or process 4 proceeds after executing V.

P and V semaphore functions are in POSIX 100.3b, an IEEE-accepted standard for the IPCs. They can be used as an event signalling, as a mutex, as a counting semaphore and the semaphores for the bounded buffer problem solution.

7.8 SHARED DATA

7.8.1 Problem of Sharing Data by Multiple Tasks and Routines

The shared data problem can be explained as follows: Assume that several functions (or ISRs or tasks) share a variable. Let us assume that at an instant the value of that variable operates and during its operations, only a part of the operation is completed and a part remains incomplete. At that moment, let us assume that there is an interrupt. Now, assume that there is another function. It also shares the same variable. The value of the variable may differ from the one expected if the earlier operation had been completed. The incomplete operation can occur as follows.

Suppose a variable is of 128 bits and the processor is of 32 bits. The operations on the variable will be using four 32-bit ALU operations in order to use the 32-bit ALU of this processor. Atomic operation is one, which

cannot be subdivided into the suboperations or none of the suboperations can be left incomplete and no other operation can start before all suboperations are complete. Now, assume that the 128-bit operation on the variable is non-atomic. This means that the operation can be interrupted before all the four operations are completed. An interrupt can occur at the end of each 32-bit ALU operation, not necessarily at the end of the 128-bit operation. Therefore, the called ISR or another function can use the incompletely operated variable or can change that variable when it shares with another function. On return, new values of that variable will be found and the incomplete operations will be performed on that new 128-bit variable in place of the old one.

Example 7.15

(*a*) Consider x, a 128-bit variable, b_{127}......b_0. Assume that the operation OP_{sl} is shift left by 2 bits to multiply it by 4 and find y = 4 × x. Let OP_{sl} done non-atomically in four suboperations, OPA_{sl}, OPB_{sl}, OPC_{sl} and OPD_{sl} for b_{31}......b_0 , b_{63}......b_{32} , b_{95}......b_{64} and b_{127}......b_{96}, respectively. Assuming at an instance that suboperations OPA_{sl}, OPB_{sl} and OPC_{sl} are completed and OPD_{sl} remained incomplete. Now interrupt *I* occurs at that instance. The *I* calls some function which uses x if x is the global variable. It modifies x = b'_{127}......b'_0. On return from interrupt, as OPD_{sl} is not complete, OPD_{sl} operates on b'_{127}......b'_{96} not on b_{127} ... b_{96}.

(*b*) Consider date d and time t. Let d and t be taken in the program as global variables. Assume that a thread *Update_Time_Date* is updating t and d information on system clock tick interrupt I_S. The thread *Display_Time_Date* in Example 7.2 displays that t and d information.

1. Assume that when *Update_Time_Date* ran t = 23:59:59 and date d = 17 July 2007.
2. The *Display_Time_Date* gets interrupted and assumes that display d and operation t are non-atomic. Display of d was completed but display of t was incomplete when interrupt I_S occurred.
3. After a while, the t changes to t = 00:00:00 and d = 18 July 2007 when the thread *Update_Time_Date* runs. The display will show t = 00:00:00 and date d = 17 July 2007 on the return from interrupt and re-start of blocked thread *Display_Time_Date*.

The use of shared variables d and t in two threads *Update_Time_Date* and *Display_Time_Date* causes the display error.

7.8.2 Shared Data Problem Solutions

One solution is the use of atomic operation for solving the shared data problem. We need atomic operations because of the following.

1. An interrupt can occur at the end of an instruction cycle, not at the end of a high-level instruction.
2. A DMA operation can occur at the end of a machine cycle itself and a compiler or program may not taking these atomic-level details into account (DMA operation means direct memory access, an IO device loads into the memory using the system address and data buses when CPU is not performing any bus-operation).
3. A context switch operation can occur at the end of an instruction for calling a new function, cycle itself and a compiler or program may not be taking into account these atomic-level details.

The following are the steps that, if used together, almost eliminate a likely bug in the program because of the shared data problem:

1. *Use modifier volatile* with a declaration for a variable that returns from the interrupt. This declaration warns the compiler that certain variables can modify because the ISR does not consider the fact that the variable is also shared with a calling function.

2. *Use reentrant functions* with atomic instructions in that part of a function that needs its complete execution before it can be interrupted. This part is called the critical section. For example, the suboperations of shifting of x in Example 7.15 are in critical section.
3. *Put a shared variable in a circular queue*. A function that requires the value of this variable always deletes (takes) it from the queue *front*, and another function, which inserts (writes) the value of this variable, always does so at the queue *back*. Now a problem can occur in case there are a large number of functions that post the values into and take the values but the maximum required queue size is not provided.

Example 7.16

Example shows how shared data problem get solved by using queue. Consider the Example 7.15(b). Assume that variables t for time and d for date are shared variables and there is a queue Q_{TD} into which a thread inserts the shared variables and another thread deletes these variables. [*Note:* Insertion into a queue means writing a value at the queue tail and then changing the pointer to the queue tail for the next insertion. Deletion from a queue means reading the value
from the queue head and then changing the pointer to the queue head for the next read.]

1. Assume that when *Update_Time_Date* runs the t = 23:59:59 and date d = 17 July 2007 and inserts these in a queue Q_{TD}.
2. The *Display_Time_Date* reads t and d from Q_{TD} and displays. When the thread gets interrupted after reading d, display of d, the Q_{TD} is still holding t when interrupt I_S occurs. Display of t will complete on return from the interrupt.
3. After a while, the t changes to t = 00:00:00 and date d = 18 July 2007 and the thread *Update_Time_Date* runs and inserts the new values of t and d at the back of the earlier values in Q_{TD}. The display will show d = 17 July 2007 and t = 23:59:59 and on return from the interrupt and in the next cycle run the thread will show d = 18 July 2007 and t = 00:00:00.

The use of queue for the shared variables d and t in two threads when *Update_Time_Date* inserts these into the queue and *Display_Time_Date* deletes these from the Q_{TD} causes no display error.

4. *Disable the interrupts before a critical section starts executing* and *enable* the interrupts on its completion. It is a powerful but drastic option. An interrupt, even if of higher priority than the present critical function, gets disabled. Advantage of it is that the semaphore functions have greater computational overhead than disabling of the interrupt. The difficulty with this option is that it increases in the interrupt latency period for all the tasks. The latency increases by the time taken in executing the codes of the section. A deadline may be missed for an interrupt service by that task which does not share the critical section or the resource.

As an alternative to disabling interrupts, Section 7.7.2 described using of semaphores for the shared data problem. A software designer must not use the drastic option of disabling interrupts in all the critical sections. [*Note:* In the OS for automobile applications, the disabling of interrupts is used before entering any critical section to avert any unintended action because of improper use of semaphores.] Another alternative is use of lock or spin-lock functions in a scheduler. (Section 7.11)

The use of disabling the switching of task from one to another and other steps and use of semaphores (Section 7.7) must eliminate the shared data problem completely from a multitasking, multi-ISRs and multiple shared variable cases. Each of the step has its own inherent benefits in solving the problem. A programmer must utilize the various steps optimally suited to solve the problem.

Shared data problem can arise in a system when another higher priority task finishes an operation and modifies the data or a variable. Using reentrant functions, disabling interrupt mechanism, using semaphores and IPCs such as mailbox and queue are the solutions which are used for taking care of shared data problem.

7.8.3 Applications of Semaphores and Shared Data Problem

Use of mutex facilitates mutually exclusive access by two or more processes to the resource (CPU). The same variable, *sem_m*, is shared between the various processes. Let process 1 and process 2 share *sem_m* and its initial value = 1.

1. Process 1 proceeds after *sem_m* decreases and equals 0 and gets the exclusive access to the CPU.
2. Process 1 ends after *sem_m* increases and equals 1; process 2 now gets exclusive access to the CPU.
3. Process 2 proceeds after *sem_m* decreases and equals 0 and gets exclusive access to CPU.
4. Process 2 ends after *sem_m* increases and equals 1; process 1 now gets the exclusive access to the CPU.

The *sem_m* is like a resource key and shared data within the processes 1 and 2 is the resource. Whosoever first decreases it to 0 at the start gets the access to and prevents the other to run with whom this key shares.

Mutex is a semaphore that provides at an instance two tasks mutually exclusive access to resources and is used in solving shared data problem.

7.8.4 Elimination of Shared Data Problem

The use of semaphores does not eliminate the shared data problem completely. Software designers may not take the drastic option of disabling interrupts in all the critical sections by using semaphores. When using semaphores, the OS does not disable the interrupts. Alternatively, task-switching flags can be used (Section 8.10.3). The following problems that can arise when using semaphores.

1. Sharing of two semaphores creates a deadlock problem (Refer to Section 7.8.5).
2. Suppose the semaphore taken is never released? There should therefore be some time-out mechanism after which the error message is generated or an appropriate action taken. There is some degree of similarity with the watchdog timer action on a time-out. A watchdog timer on timeout resets the processor. Here, after the time out, the OS reports an error and runs an error-handling function. Without a time out, an ISR worst-case latency may exceed the deadline.
3. A semaphore not taken, and another task uses a shared variable.
4. What happens when a train takes a signal for a wrong track? When using the multiple semaphores, if an unintended task takes the semaphore, it creates a problem.
5. There may be priority inversion problem (Refer to Section 7.8.5).

7.8.5 Priority Inversion Problem and Deadlock Situations

Let the priorities of tasks be in an order such that task *I* is of the highest priority, task *J* is of a lower and task *K* of the lowest. Assume that only tasks *I* and *K* share the data and *J* does not share data with *K*. Also let tasks *I* and *K* alone share a semaphore and not *J*. Why do only a few tasks share a semaphore? Can't all share a semaphore? The reason is that the worst-case latency becomes too high and may exceed the deadline if all tasks are blocked when one task takes a semaphore. The worst-case latency will be small only if the time taken by the tasks that share the resources is relevant. Now consider the following situation.

At an instant t_0, suppose task *K* takes a semaphore, the OS does not block task *J* and blocks task *I*. This happens because only tasks *I* and *K* share the data and *J* does not. Consider the problem that now arises on selective sharing between *K* and *I*. At the next instant t_1, let task *K* become ready to run first on an interrupt. Now, assume that at the next instant t_2, task *I* becomes ready on an interrupt. At this instant, *K* is in the critical section. Therefore, task *I* cannot start at this instant due to *K* being in the critical region. Now, if at next instant t_3, some action (event) causes the unblocked higher than the *K* priority task *J* to run. After instant t_3, running task *J* does not allow the highest priority task *I* to run. This is because even though *K* is not running and thus unable to release the semaphore that it shares with *I*. Further, the code of task *J* may be such that even when the semaphore is released by task *K*, it may not let *I* run (*J* runs the codes as if it is in critical section all the time). The *J* action is now as if *J* has higher priority than *I*. This is because *K*, after entering the critical section and taking the semaphore when the OS is letting *J* run, but did not share the priority information about *I*—that task *I* is of higher priority than *J*. The priority information of another higher-priority task *I* should have also been inherited by *K* temporarily, if *K* waits for *I* but *J* does not and *J* runs when *K* has still not finished the critical section codes. This did not happen because the given OS design was such that it did not provide for temporary priority inheritance in such situations.

This situation is also called *priority inversion problem*. An OS must provide for a solution for the priority inversion problem. Some OSes provide for priority inheritance in these situations and thus priority inheritance problem does not occur when using them. Refer to Section 7.7.2 for use of a mutex for resources sharing. A mutex should be a mutually exclusive Boolean function, by using which the critical section is protected from interruption in such a way that the problem of priority inversion does not arise. Mutex is automatically provided in certain RTOS so that the priority inversion problem does not arise. Mutex use may also be just analogous to a semaphore defined in Section 7.7.2 in another RTOS and which does not solve the priority inversion problem.

Consider another problem. Assume the following situation.

1. Let the priorities of tasks be such that task *H* is of highest priority. Then task *I* has a lower priority and task *J* has the lowest.
2. There are two semaphores, *SemTok1* and *SemTok2*. This is because the tasks *I* and *H* have a shared resource through *SemTok1* only. Tasks *I* and *J* have two shared resources through two semaphores, *SemTok1* and *SemTok2*.
3. Let *J* interrupt at an instant t_0 and first take both the semaphores *SemTok1* and *SemTok2* and run.

Assume that at a next instant t_1, being now of a higher priority, the task *H* interrupts the tasks *I* and *J* after it takes the semaphore *SemTok1*, and thus blocks both *I* and *J*. In-between the time interval t_0 and t_1, the *SemTok1* was released but *SemTok2* was not released during the run of task *J*. But the latter did not matter as the tasks *I* and *J* do not share *SemTok2*. At an instant t_2, if *H* now releases the *SemTok1*, allows the task *I* to take it. Even then it cannot run because it is also waiting for task *J* to release the *SemTok2*. The task *J* is waiting at a next instant t_3, for either *H* or *I* to release the *SemTok1 because it needs this to again enter a critical section*. Neither task *I* can run after instant t_3 nor task *J*. There is a circular dependency established between *I* and *J*.

This situation is also called a *deadlock* situation. On the interrupt by *H*, the task *J*, before exiting from the running state, should have been put in queue-front so that later on, it should first take *SemTok1*, and the task *I* put in queue next for the same token, the deadlock would not have occurred (refer to Section 7.12 for queuing of messages).

The use of mutex solves the deadlock problem in certain OSes. Its use may be just analogous to a semaphore defined in Section 7.2.1. In other OSes, the mutex use may not solve the deadlock situation.

Priority becomes inverted and deadlock (circular dependency) develops in certain situations when using semaphores. Certain OSes provide the solution to this problem of semaphore use by ensuring that these situations do not arise during the concurrent processing of multitasking operations.

7.9 INTERPROCESS COMMUNICATION

Is it possible to send through the kernel an output data (a message of a known size with or without a header) or processing by another task? One way is the use of global variables. Use of these now creates two problems.

One is the shared data problem (Section 7.8). The other problem is that the global variables do not prevent (encapsulate) a message from being accessed by other tasks.

IPCs in a multiprocessor system are used to generate information about certain sets of computations finishing on one processor and to let the other processors waiting for finishing those computations take note of the information.

IPC means that a process (scheduler, task or ISR) generates some information by signal or value or generates an output so that it allows another process to take note or use it through the kernel functions for the IPCs. IPCs in a multitasking system are used to set or reset a signal or token or flag or generate message from the certain sets of computations finishing on one task and to let the other tasks take note of the signal or get the message.

OSes provide the software programmer the following IPC functions, which can be used.

1. Signals
2. Semaphores *as token or mutex* or counting semaphores for the intertask communication between tasks sharing a common buffer or operations
3. Queues and mailboxes
4. Pipes and sockets
5. Remote procedure calls (RPCs) for distributed processes.

Section 7.7 described two IPC functions, OSSemPend ( ) and OSSemPost ( ) and use of semaphores for the IPCs. The following shows an application for the printing from buffer by a task.

Example 7.17

Consider using a mutex semaphore in the tasks, which needs to use the *print* for the buffer data from one task at an instance. A task runs a *print* function, which reads from a print buffer and prints. The kernel should let the other tasks share this task. The *print* task can be shared among the multiple tasks, which use the mutex semaphore IPCs in their critical sections.

When the printer buffer becomes available for new data, an IPC from the *print* task is generated and the kernel takes note of it. Other tasks then take note of it. A task takes note of it by the OSSemPend ( ) function of the kernel used at the beginning of the critical section and the task gets mutually exclusive access to the section to send messages into the print buffer by using the OSSemPost ( ) function of the kernel at the end of the section (Sections 7.7.2, 7.8.3 and 7.11.1).

Consider Example 7.18, which shows use of the functions for semaphore and mailbox IPCs (semaphore and mailbox IPC functions will be described in detail in Sections 7.11 and 7.13).

Example 7.18

Consider a mobile phone device (Section 1.10.5) *Update_Time task* (Example 7.8). Assume that there is a task, *Task_Display* for a multiline display of outputs which displays current time on the last line. When the multiline display task finishes the display of the last but one line, an IPC semaphore *supdateTD* from the *display* task is generated and the kernel takes note of it. The task—continuously updating time—then can take the *supdateTD* and generate an IPC as a mailbox output for the current time and date.

When the task for updating time t on a signal posted on system clock-tick interrupt I_S starts, on taking the *supdateTD* posted by the Task_Display, it can write the td into a *mailbox* using an IPC for posting mailbox message, *timeDate*.

Assume OSSemPost ( ) is an OS IPC function for posting a semaphore and assume OSSemPend ( ) is another OS IPC function for waiting for the semaphore. Let *supdateTD* be the binary semaphore posted at *Task_Display* and pending at *Task_Display* section for displaying t and d. Let *supdateTD* initial value = 1.

Let the IPC function be OSMboxPost ( ) for posting the mailbox IPC message from *Update_Time task* and OSMboxPend ( ) for waiting for the mailbox IPC at *Task_Display* section. Let timeDate initial value = null.

The following will be the codes:

```
static void Task_Display (void *taskPointer) {
.
while (1) {
.
/* IPC for waiting time and date in the mailbox */
TimeDateMsg = OSMboxPend (timeDate) /* Wait for the mailbox message timeDate. The timeDate becomes
    null after the mailbox is posted time and date by Task_ Update_Time and TimeDateMsg equals the
    updated time and date */
/* Code for display TimeDateMsg Time: hr:mm Date: month:date */
.
/* IPC for requesting TimeDate */
OSSemPost (supdateTD) /* Post for the semaphore supdateTD. supdateTD becomes 1
};
static void Task_ Update_Time (void *taskPointer) {
.
while (1) { /* wait for system clock inter interrupt signal or semaphore notification from ISR of I_s */
OSSemPend (supdateTD) /* Wait the semaphore supdateTD. This means that OS function decrements
    supdateTD in corresponding event control block. supdateT becomes 0 */
/* Codes for updating time and date as per the number of clock interrupts received so far */
/* Codes for writing into the mailbox */
OSMboxPost (timeDate) /* Post for the mailbox message and timeDate, which equaled null
    now equals newupdated time and date*/
.
    };
```

The need for IPC and thus intertask communications also arises in a client-server network.

IPC means that a process (scheduler or task or ISR) generates some information by setting or resetting a token or value, or generates an output so that it lets another process take note or use it under the control of OS.

7.10 SIGNAL FUNCTION

One way for messaging is to use an OS function *signal* ( ). It is provided in Unix, Linux and several RTOSes. Unix and Linux OSes use *signals* profusely and have 31 different types of *signals* for the various events. Section 9.3 will describe *signal* in VxWorks RTOS. Just a hardware mechanism sends the interrupt to the OS, task (or process) or the OS itself sends *signal*. The task or process sending the signal uses a function signal ( ) having an integer number n in the argument. A signal is function, which executes a software interrupt instruction INT n or SWI n.

A 'signal' provides the shortest communication. The *signal* ( ) sends a output n for a process, which enables the OS to unmask a signal mask of a process or task as per the n. The task is called signal handler and has coding similar to the ones in an ISR. The handler runs in a way similar to a highest priority ISR. An ISR runs on an hardware interrupt provided that the interrupt is not masked. The handler runs on the signal provided that the signal is not masked.

The signal ( ) forces the OS to first run a signalled process or task called signal handler. When there is return from the signalled or forced task or process, the process, which sent the signal, runs the codes as happens on a return from an ISR. A signal mask is the software equivalent of the flag at a register that sets on masking a hardware interrupt. Unless masked by a *signal* mask, the *signal* allows the execution of the signal-handling function and allows the handler to run just as a hardware interrupt allows the execution of an ISR.

An integer number (for example n) represents each signal and that number associates a function (or process or task) signal handler, an equivalent of the ISR. The signal handler has a function called whenever a process communicates that number.

A signal handler is not called directly by a code. When the signal is sent from a process, OS interrupts the process execution and calls the function for signal handling. On return from the signal handler, the process continues as before.

For example, signal (5). The signal mask of signal handler 5 is reset. The signal handler and connect function associate the number 5. The function represented by number 5 is forced run by the signal handler.

An advantage of using it is that unlike semaphores it takes the shortest possible CPU time to force a handler to run. The *signals* are the interrupts that can be used as the IPC functions of synchronizing.

A *signal* is unlike the semaphore. The semaphore has use as a token or resource key to let another *task process* block, or which locks a resource to a particular *task process* for a given section of the codes. A signal is just an interrupt that is shared and used by another *interrupt-servicing process*. A *signal* raised by one process forces another process (signal handler) to interrupt and catch that *signal* in case the *signal* is not masked (use of that signal handler is not disabled). Signals are to be handled only for forcing the run of very high priority processes as it may disrupt the usual schedule and priority inheritance mechanism. It may also cause reentrancy problems.

An important application of the *signals* is to handle *exceptions*. (An exception is a process that is executed on a specific reported run-time condition.) A *signal* reports an error (called 'exception') during the running of a task and then lets the scheduler initiate an error-handling process or function or task. An error-handling signal handler handles the different error login of other task. The device driver functions also use the signals to call the handlers (ISRs).

The following are the signal related IPC functions, which are generally not provided in the RTOS such as µCOS-II and provided in RTOS such as VxWorks or OS such as Unix and Linux.

1. *SigHandler* ( ) to create a signal handler corresponding to a signal identified by the signal number and define a pointer to the signal context. The signal context saves the registers on signal.
2. *Connect* an interrupt vector to a signal number, with signalled handler function and signal-handler arguments. The interrupt vector provides the PC value for the signal-handler function address.
3. A function *signal ( )* to send a signal identified by a number in the argument to a task.
4. *Mask* the signal.
5. *Unmask* the signal.
6. *Ignore* the signal.

1. The simplest IPC for messaging from processes that forces a handler function to run (provided unmasked) is the use of 'signal'.
2. A 'signal' provides the shortest communication. Signals are used for initiating exceptions and error-handling processes.
3. IPC function for a signal is signal (n) for signalling signal-handler associated with n to run if not masked.

7.11 SEMAPHORE FUNCTIONS

The OS provides for semaphore as notice or token for an event occurrence. Semaphore facilitates IPC for notifying (through a scheduler) a waiting task section change to the running state upon event at presently running code section at an ISR or task. A semaphore as binary semaphore is a token or resource key (Sections 7.7.1 and 7.7.2). The OS also provides for mutex access to a set of codes in a task (or thread or process) (Sections 7.7.2 and 7.8.3). The use of mutex is such that the priority inversion problem is not solved in some OSes while it is solved in other OSes. The OS also provides for counting semaphores. The OS may provide for POSIX standard P and V semaphores which can be used for notifying event occurrence or as mutex or for counting. The timeout can be defined in the argument with wait function for the semaphore IPC. The error pointer can also be defined in the arguments for semaphore IPC functions.

The following are the functions, which are generally provided in an OS, for example, µCOS-II for the semaphores.

1. *OSSemCreate*, a semaphore function to create the semaphore in an event control block (ECB). Initialize it with an initial value.
2. *OSSemPost*, a function which sends the semaphore notification to ECB and its value increments on event occurrence (used in ISRs as well as tasks).
3. *OSSemPend*, a function, which waits the semaphore from an event, and its value decrements on taking note of that event occurrence (used in tasks not in ISRs).
4. *OSSemAccept*, a function, which reads and returns the present semaphore value and if it shows occurrence of an event (by non-zero value) then it takes note of that and decrements that value (no wait; used in ISRs as well as tasks).
5. *OSSemQuery*, a function, which queries the semaphore for an event occurrence or non-occurrence by reading its value and returns the present semaphore value, and returns pointer to the data structure OSSemData. The semaphore value does not decrease. The OSSemData points to the present value and table of the tasks waiting for the semaphore (used in tasks).

An OS provides the IPC functions create, post, pend, accept and query for using semaphores. The time-out can be provided with 'pend' function arguments. A pointer to error-handling function can also be specified in the arguments.

7.11.1 Mutex, Lock and Spin Lock

An OS, using a mutex blocks a critical section in a task on taking the mutex by another task's critical section. Other task unlocks on releasing the mutex. The mutex wait by blocked task can be for a specified timeout.

There is a function in kernel called lock ( ). It locks a process to the resources till that process executes unlock ( ). A wait loop creates and when wait is over the other processes waiting for the lock starts. Use of lock ( ) and unlock ( ) involves little overhead compared to uses of OSSemPend ( ) and OSSemPost ( ) when using a mutex. Overhead means number of operations needed for blocking one process and starting another. However, *a resource of high priority should not lock the other processes by blocking an already running task in the following situation.* Suppose a task is running and a little time is left for its completion. The running time left for it is less compared with the time that would be taken in blocking it and context switching. There is an innovative concept of spin locking in certain OS schedulers. A *spin lock* is a powerful tool in the situation described before. [Refer to 'Multithreaded Programming with Java' by

Bil Lewis and Daniel J. Berg, Sun Microsystems Inc., 2000.] The scheduler locking process for a task *I* waits in a loop to cause the blocking of the running task first for a time interval t, then for (t - δt), then (t - 2δt) and so on. When this time interval spin-downs to 0, the task that requested the lock of the processor now unlocks the running task *I* and blocks it from further running. The request is now granted to task *J* to unlock and start running provided that task is of higher priority. *A spin lock does not let a running task to be blocked instantly, but first successively tries with or without decreasing the trial periods before finally blocking a task.* A spin-lock obviates need of context-switching by pre-emption and use of mutex function-calls to OS.

An OS provides the IPC functions for creating and accessing the resource using mutex for a process and to prevent the resource for the other processes. An OS may also provides for lock or spin-locks at the scheduler.

7.12 MESSAGE QUEUE FUNCTIONS

Some OSes do not distinguish, or make little distinction, between the use of queues, pipes and mailboxes during the message communication among processes, while other OSes regard the use of queues as different.

A message *queue* is an IPC with the following features.

1. An OS provides for inserting and deleting the message pointers or messages.
2. Each queue for the message or message-pointers needs initialization (creation) before using functions in kernel for the queue.
3. Each created queue has an ID.
4. Each queue has a user-definable size (upper limits for number of bytes).
5. When an OS call is to insert a message into the queue, the bytes are as per the pointed number of bytes. For example, for an integer or float variable as a pointer, there will be 4 bytes inserted per call. If the pointer is for an array of eight integers, then 32 bytes will be inserted into the queue. When a message-pointer is inserted into queue, the 4 bytes inserts, assuming 32-bit addresses.
6. When a queue becomes full, there is error handling function to handle that.

Figure 7.8(a) shows functions for the queues in the OS. Figure 7.8(b) shows a queue-message block with the messages or message-pointers. Two pointers, *QHEAD and *QTAIL are for queue head and tail memory locations.

The OS functions for a queue, for example, in µCOS-II, can be as follows:

1. OSQCreate, a function that creates a queue and initializes the queue.
2. OSQPost, a function that sends a message into the queue as per the queue tail pointer, it can be used by tasks as well as ISRs.
3. The OSQPend waits for a queue message at the queue and reads and deletes that when received (wait, used by tasks only, not used by ISRs).
4. OSQAccept deletes the present message at queue head after checking its presence yes or no and after the deletion the queue head pointer increments. (no wait; used by ISRs as well as tasks)
5. OSQFlush deletes the messages from queue head to tail. After the *flush* the queue head and tail points to QTop, which is the pointer at start of the queuing. (used by ISRs and tasks)
6. OSQQuery queries the queue message block but the message is not deleted. The function returns pointer to the message queue *Qhead if there are the messages in the queue or else returns NULL. It returns a pointer to the structure of the queue data structure for *QHEAD, number of queued messages, size and table of tasks waiting for the messages from the queue. (query is used in tasks)
7. OSQPostFront sends a message to front pointer, *QHEAD. Use of this function is made in the following situations. A message is urgent or is of higher priority than all the previously posted message into the queue (used in ISRs and tasks).

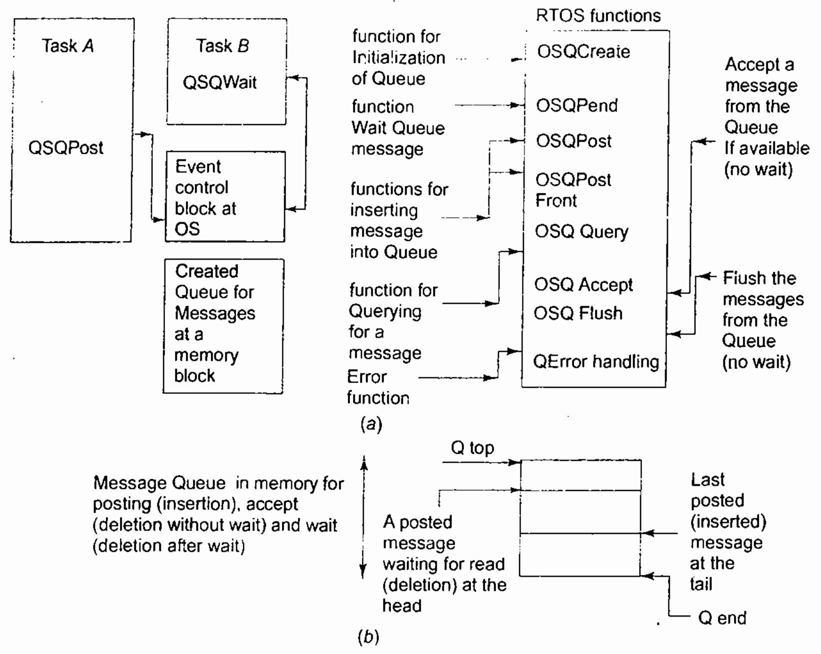

Task A
QSQPost
Task B
QSQWait
Event control block at OS
Created Queue for Messages at a memory block
RTOS functions
function for Initialization of Queue
function Wait Queue message
functions for inserting message into Queue
function for Querying for a message
Error function
OSQCreate
OSQPend
OSQPost
OSQPost Front
OSQ Query
OSQ Accept
OSQ Flush
QError handling
Accept a message from the Queue If available (no wait)
Flush the messages from the Queue (no wait)
(a)
Message Queue in memory for posting (insertion), accept (deletion without wait) and wait (deletion after wait)
Q top
A posted message waiting for read (deletion) at the head
Last posted (inserted) message at the tail
Q end
(b)

Fig. 7.8 (*a*) OS functions for queue and use of post and wait functions by Tasks *A* and *B* (*b*) Queue message block in memory

Example 7.19

Consider Orchestra Playing Robots (Example 1.10.7). Task_*Director_Output* puts the musical notes into the queue at conducting and directing robot. OSQEntries equals the number of queue entries and OSQSize equals the maximum number of notes that can be put into the queue.

```
static void Task_Director_Output (void *taskPointer) {
.
while (1) {
.
/* Codes for inserting musical notes into the queue */
for (OSQEntries = 0; OSQEntries < OSQSize; OSQEntries ++)
{OSQPost (QDirector, note)} /* Post for the Queue QDirector messages upto the OSQSize /*
.
};
static void Task_Player_Input (void *taskPointer) {
.
while (1) {
.
```

```
/* Codes for deleting notes from the queue */
for (OSQEntries = OSQSize; OSQEntries >0 ; OSQEntries --)
note (i) = OSQPend (QDirector, 0, err) /* wait for the message */
.
};
```

In certain RTOS, a queue is given select option and the option is provided for priority or FIFO. The task having priority if started deletes a queue message first in case the priority option is selected. The task pending since longer period deletes a queue message first in case the FIFO option is selected.

An OS provides the IPC functions create, post, postfront, pend, accept, flush and query for using message queues. The timeout can be provided with 'pend' function argument. The error-pointer can also be provided in the argument.

7.13 MAILBOX FUNCTIONS

A message-mailbox is for an IPC message that can be used only by a single destined task. The mailbox message is a message-pointer or can be a message. (µCOS-II provides for sending message-pointer into the box). The source (mail sender) is the task that sends the message pointer to a created (initialized) mailbox (the box initially has the NULL pointer before the message posts into the box). The destination is the place where the OSMBoxPend function waits for the mailbox message and reads it when received.

A mobile phone LCD display task is an example that uses the message mailboxes as an IPC. In the mailbox, when the time and date message from a clock-process arrives, the time is displayed at side corner on top line. When the message from anther task to display a phone number, it is displayed the number at middle at a line. When the message from anther task to display the signal strength at antenna, it is displayed at the vertical bar on the left.

Another example of using a mailbox is the mailbox for an error-handling task, which handles the different error logins from other tasks.

Figure 7.9(a) shows three mailbox-types at the different RTOSes. Figure 7.9(b) shows the initialization and other functions for a mailbox at an OS. The following may be the provisions at an OS for IPC functions when using the mailbox.

1. A task may put into the mailbox only a pointer to the message-block or number of message bytes as per the OS provisioning.
2. There are three types of the mailbox provisions.

A queue (Section 7.12) may be assumed a special case of a mailbox with provision for multiple messages or message pointers. An OS can provide for *queue* from which a read (deletion) can be on a FIFO basis or alternatively an OS can provide for the multiple mailbox messages with each message having a priority parameter. The read (deletion) can then only be on priority basis in case mailbox message has multiple messages with priority assigned to high priority ones. Even if the messages are inserted in a different priority, the deletion is as per the assigned priority parameter.

An OS may provision for *mailbox* and *queue* separately. A mailbox will permit one message pointer per box and the queue will permit multiple messages or message pointers. µCOS-II RTOS is an example of such an OS.

The RTOS functions for mailbox for the use by the tasks can be the following:

1. OSMBoxCreate creates a box and initializes the mailbox contents with a NULL pointer.
2. OSMBoxPost sends (writes) a message to the box.

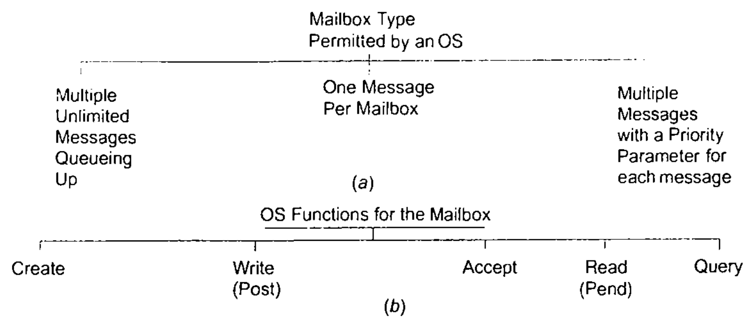

Mailbox Type
Permitted by an OS
Multiple Unlimited Messages Queueing Up
One Message Per Mailbox
Multiple Messages with a Priority Parameter for each message
(a)
OS Functions for the Mailbox
Create
Write (Post)
Accept
Read (Pend)
Query
(b)

Fig. 7.9 (*a*) Mailbox types at the different operating systems (OSes) (*b*) Initialization and other functions for a mailbox at an OS

3. OSMBoxWait (pend) waits for a mailbox message, which is read when received.
4. OSMBoxAccept reads the current message pointer after checking the presence yes or no (no wait). Deletes the mailbox when read.
5. OSMBoxQuery queries the mailbox when *read* and not needed later.

An ISR can post (but not wait) into the mailbox of a task.

Example 7.20

(*a*) Consider an AVCM (Section 1.10.2). Assume that a message pointer IPC posts into the mailbox the *amount* collected by *Task Read_Amount* and the *Chocolate_delivery_task* section waits for taking message into the mailbox to make the amount equal to NULL after delivering the chocolate. Assume *mboxAmt* is a pointer to mailbox and fullAmount is a string *full amount* which should be made NULL after delivering the chocolate. OSMboxPost (mboxAmt, full Amount) is an OS IPC function for posting a message pointer into the mailbox and assume OSSemPend ( ) is another OS IPC function for waiting for the message pointer, *fullAmount*.

(*b*) Also assume that the OSMboxPost ( ) is also used by keypad for posting the mailbox IPC message for *userInput* into mailbox *mboxUser* from *Task User Keypad Input* and OSMboxPend ( ) for waiting for *userInput* for display. A mailbox IPC 'pend' is for mboxUser message in *Task_Display*.

The following will be the codes for (*a*) and (*b*).

```
(a) static void Task Read_Amount (void *taskPointer) {
.
while (1) {
.
/* Codes for reading the coins inserted into the machine */
/* Codes for writing into the mailbox full amount message if cost of chocolate is received*/
OSMboxPost (mboxAmt, fullAmount) /* Post for the mailbox message and fullAmount, which equaled
    NULL now equals fullAmount */
.
};
static void Chocolate_delivery_task (void *taskPointer) {
.
while (1) {
```

```
.
/* IPC for requesting full amount message */
fullAmountMsg = OSMboxPend (mboxAmt, 20, *err) /* Wait for the mailbox mboxAmt message for
    20 clock-ticks and error if message not found. mboxAmt becomes NULL after message is read.
.
};
(b) static void Task_User_Keypad_Input (void *taskPointer) {
.
while (1) {
.
/* Codes for reading keys pressed by the user before the enter key */
/* Codes for writing into the mailbox */
OSMboxPost (mboxUser, userInput) /* Post for the mailbox message and userInput, which equaled NULL
    now equals userInput */
.
};
static void Task_Display (void *taskPointer) {
.
while (1) {
.
/* IPC for waiting for User input message */
UserInputMsg = OSMboxPend (mboxUser, 20, *err) /* Wait for the mailbox mboxUser
    message for 20 clock ticks and error if message not found. mboxUser becomes null after
    message is read.
.
/* Code for display of user Input */
TimeDateMsg = OSMboxPend (timeDate, 20, err) /* Wait for the mailbox message timeDate.
/* Code for display TimeDateMsg Time: hr:mm Date: month:date */
.
};
```

An OS provides the IPC functions *create*, *post*, *pend*, *accept* and *query* for using the mailbox. The time-out and error-pointer can be provided with the 'pend' function arguments.

7.14 PIPE FUNCTIONS

The OS pipe functions are unlike message queue functions. The difference is that pipe functions are similar to the ones used for devices such as file.

A message-pipe is a device for inserting (writing) and deleting (reading) from that between two given interconnected tasks or two sets of tasks. Writing and reading from a pipe is like using a C command *fwrite with a file name* to write into a named file, and *fread with a file name* to read from a named file. Pipes are also like Java *PipeInputOutputStreams*.

1. One task using the function *fwrite* in a set of tasks, can insert (write) *to* a pipe at the back pointer address, *pBACK.
2. Another task using the function *fread* in a set of tasks can delete (read) from a pipe at the front pointer address, *pFRONT.

3. In a pipe there may be no fixed number of bytes per message but there is end-pointer. A pipe can therefore be inserted limited number of bytes and have a variable number of bytes per message between the initial and final pointers.
4. Pipe is unidirectional. One thread or task inserts into it and the other one deletes from it.

An example of the need for messaging and thus for IPC using a pipe is a network stack.

The OS functions for pipe are the following:

1. *pipeDevCreate* for creating a device, which functions as pipe.
2. *open* ( ) for opening the device to enable its use from beginning of its allocated buffer, its use is with options and restrictions (or permissions) defined at the time of opening.
3. *connect* ( ) for connecting a thread or task inserting bytes to the thread or task deleting bytes from the pipe.
4. *write* ( ) function for inserting (writing) from the bottom of the empty memory space in the buffer allotted to it.
5. *read* ( ) function for deleting (reading) from the pipe from the bottom of the unread memory spaces in the buffer filled after writing into the pipe.
6. *close* ( ) for closing the device to enable its use from beginning of its allocated buffer only after opening it again.

Figure 7.10(a) shows functions at an OS. A function is for initialization and creating a pipe. It defines pipe ID, length, maximum length (not defined in some OSes) and initial values of two pointers. These are *pFRONT and *pBACK for pipe message destination (head) and pipe message source (tail) memory locations, respectively. A function is for pipe connecting and thus defining source ID and destination ID. A function is for the error handling. Figure 7.10(b) shows pipe messages in a message buffer.

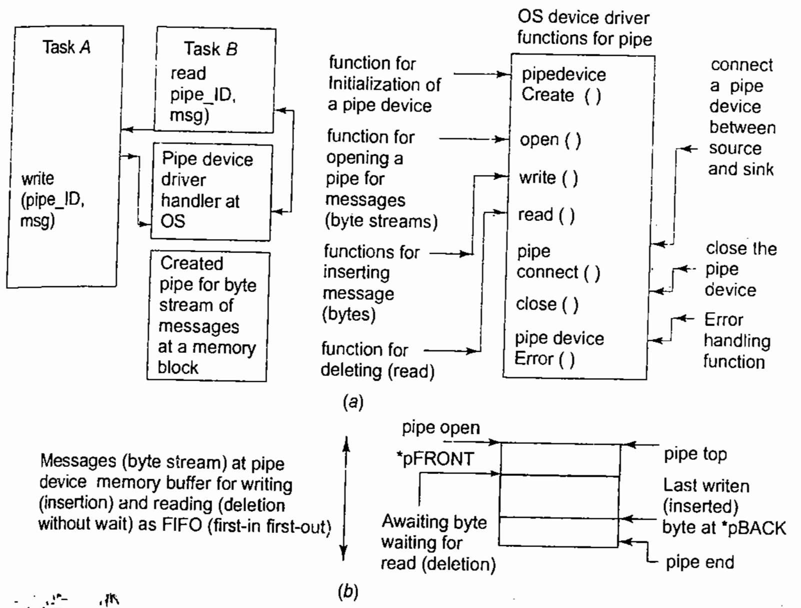

Task A
write (pipe_ID, msg)
Task B
read pipe_ID, msg)
Pipe device driver handler at OS
Created pipe for byte stream of messages at a memory block
function for Initialization of a pipe device
function for opening a pipe for messages (byte streams)
functions for inserting message (bytes)
function for deleting (read)
OS device driver functions for pipe
pipedevice Create ( )
open ( )
write ( )
read ( )
pipe connect ( )
close ( )
pipe device Error ( )
connect a pipe device between source and sink
close the pipe device
Error handling function
(a)
Messages (byte stream) at pipe device memory buffer for writing (insertion) and reading (deletion without wait) as FIFO (first-in first-out)
pipe open
*pFRONT
Awaiting byte waiting for read (deletion)
pipe top
Last writen (inserted) byte at *pBACK
pipe end
(b)

Fig. 7.10 (*a*) Functions at operating system (*initialization, connect, read, write* and *error-handling* functions) and the use of write and read functions by tasks *A* and *B* (*b*) Pipe messages in a message buffer

Example 7.21

Consider a smart card. [Section 1.10.3 and Example 7.4] When it is inserted into a card reader host machine, it gets the radiation and charges up.

1. Assume that the main program runs an OS function pipeDevCreate ( ) to create a task.
2. Assume that a pipe is used by the task, Task_Send_Card_Info for writing card information to the host using the pipe.

The codes at the card can be as follows.

```
.
pipeDevCreate ("/pipe/pipeCardInfo", 4, 32) /* Create a pipe pipeCardInfo, which can save four messages
    each of 32 bytes maximum */
fd = open (("/pipe/pipeCardInfo",O_WR, 0) /* Open a write only device. First argument is pipe ID /pipe/
    pipeCardInfo, second argument is option O_WR for specifying write only and third argument is 0 for
    unrestricted permission.*/
.
static void Task_Send_Card_Info (void *taskPointer) {
.
while (1) {
.
.
cardTransactionNum = 0; /* At start of the transactions with the machine*/
write (fd, cardTransactionNum, 1) /* Write 1 byte for transaction number after card insertion */
write (fd, cardFabricationkey, 16) /* Write 16 bytes for fabrication key */
write (fd, cardPersonalisationkey, 16) /* Write 16 bytes for personalisation key */
write (fd, cardPIN, 16) /* Write 16 bytes for PIN, personal identification number
    granted by the authorising bank */
.
};
```

An OS provides the IPC functions pipeDevcreate, open, connect, write, read and close. The pipe ID, limit of the total number of messages and the maximum size per message are also provided when creating a pipe.

7.15 SOCKET FUNCTIONS

Example 7.21 showed that a pipe could be used for inserting the byte steam by a process and deleting the bytes from the stream by another process. However, the use of pipe between a process at the card and a process at the host will have the following problems.

1. We need the card information to be transferred from a process *A* as bytes stream to the host machine process *B* and the *B* sends messages as bytes stream to *A*. There is need for bi-directional communication between *A* and *B*.
2. We need the *A* and *B*'s ID or port as well as address information when communicating. These must be specified either for the destination alone or for both source and destination (It is similar to sending the messages in a letter along with address specification).

A protocol provides for communication along with the byte stream information of the address or port of the destination alone or addresses or ports of both source and destination. A protocol may provide for the addresses as well as ports of both source and destination in case of the remote processes (for example, in IP protocol). Also, there are two types of the protocols.

1. There may be the need of using a *connectionless protocol* when sending and receiving message streams. An example of such a protocol is UDP (user datagram protocol). UDP protocol requires a UDP header, which contains source port (optional) and destination port numbers, length of the datagram and checksum for the header-bytes. Port means a process or task for specific application. The number specifies the process. Connectionless means there is no connection establishment between source and destination before actual transfer of data stream can take place. Datagram means a set of data, which is independent and need not be in sequence with the previously sent data. Checksum is sum of the bytes to enable the checking of the erroneous data transfer. For remote communication, the address, for example, IP address is also required in the header.
2. There may be the need of using a *connection-oriented protocol*, for example, TCP. Connection-oriented protocol means a protocol, which provides that there must first a connection establishment between the source and destination, and then the actual transfer of data stream can take place. At end, there must be connection termination.

Socket provides a device-like mechanism for bi-direction communication. It provides for using a protocol between the source and destination processes for transferring the bytes; it provides for establishing and closing a connection between the source and destination processes using a protocol for transferring the bytes; it may provide for listening from multiple sources or multicasting to multiple destinations. Two tasks at two distinct places are locally interconnect through the sockets. Multiple tasks at multiple distinct places interconnect through the sockets to a socket at a server process. The client and server sockets can run on the same CPU or at distant CPUs on the Internet.

Sockets can be using a different domain. For example, a socket domain can be TCP, another socket domain may be UDP, the card and host example socket domains are different.

The use of a socket for IPC is analogous to the use of sockets for an Internet connection between the browser and webserver. *A socket provides a bi-directional transfer of messages and may also send protocol header. The transfer is between two or between the multiple clients and server-process.* Each socket may have the task source address (similar to a network or IP address) and a port (similar to a process or thread) number. The source and destination sets of tasks (addresses) may be on the same computer or on a network. Figure 7.11 shows the initialized sockets between the client set of tasks and a server set of tasks at an OS.

Example 7.22

Assume that using the OS socket functions, a socket interconnects a byte stream between the source set of processes *I* and destination targeted set of processes *J*. Let there be the four process threads: *a*, *b*, *c* and *d* in process set *I*. There are two threads, *x* and *y* in a set of processes, *J*. Let the socket be used to send a byte stream from (process set *I*, process thread *c*) to a (process set *J*, process thread *x*). Now, the socket at the source (process 1 socket) is specified as the socket at (*I*, *c*) and socket at the process 2 as the socket at (*J*, *x*). When the bytes are sent or received at the socket at (*I*, *c*) from or to the socket at (*J*, *x*), the protocol specifies (*I*, *c*) and (*J*, *x*). The protocol may also specify the length of the bytes being communicated. The protocol may also specify the checksum of the bytes being communicated so that if any bit is lost in communication to remote then retransmission request can be sent.

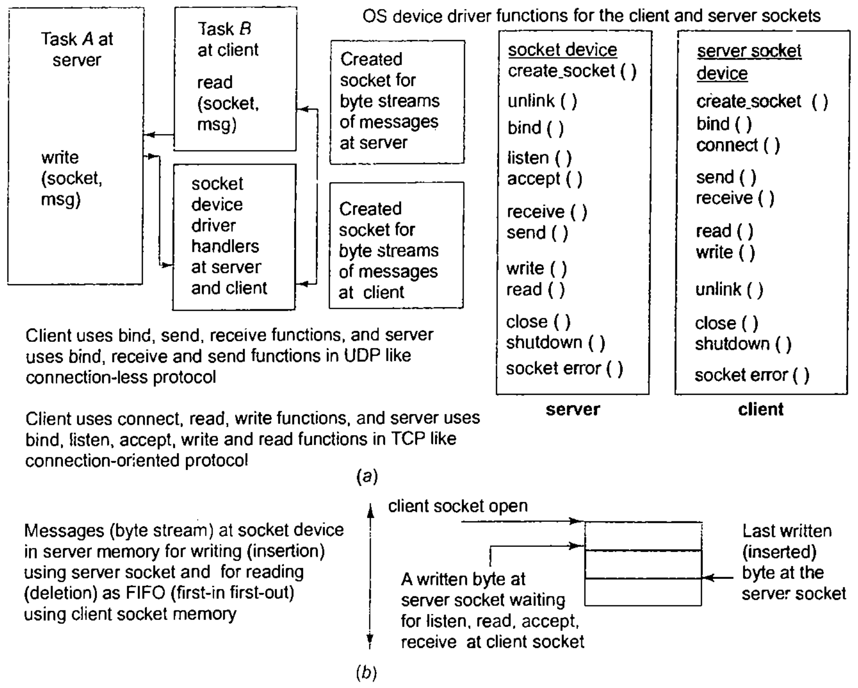

OS device driver functions for the client and server sockets
Task *A* at server
write (socket, msg)
Task *B* at client
read (socket, msg)
socket device driver handlers at server and client
Created socket for byte streams of messages at server
Created socket for byte streams of messages at client
socket device
create_socket ( )
unlink ( )
bind ( )
listen ( )
accept ( )
receive ( )
send ( )
write ( )
read ( )
close ( )
shutdown ( )
socket error ( )
server
server socket device
create_socket ( )
bind ( )
connect ( )
send ( )
receive ( )
read ( )
write ( )
unlink ( )
close ( )
shutdown ( )
socket error ( )
client
Client uses bind, send, receive functions, and server uses bind, receive and send functions in UDP like connection-less protocol
Client uses connect, read, write functions, and server uses bind, listen, accept, write and read functions in TCP like connection-oriented protocol
(a)
client socket open
Messages (byte stream) at socket device in server memory for writing (insertion) using server socket and for reading (deletion) as FIFO (first-in first-out) using client socket memory
A written byte at server socket waiting for listen, read, accept, receive at client socket
Last written (inserted) byte at the server socket
(b)

Fig. 7.11 (*a*) Initialized virtual sockets between the client set of tasks and a server set of tasks and the operating system provisions for the socket-functions (*b*) Byte stream between client and server

Example 7.23

Consider orchestra-playing robots (Example 1.5.7).

1. A process. Task_Master in the director robot creates a socket using a statement as follows:
   sfd1 = socket ("/socket/serversocket1", playStream, 0).
   The sfd1 is an unsigned integer for a socket descriptor. "/socket/serversocket1" is the path and file from which the stream, playStream will be sent or received from play robots. 0 represents unrestricted permission to use the file.
2. Task_Master socket binds the sfd1 and data structure at the socket address, sockAddr by using the function as follows:
   bind (sfd1, (struct sockaddr *)&local, lBytes).
   The lBytes is length of the bytes in the play stream;
3. Task_Master socket listens to eight orchestra-playing robots by using function as follows:
   listen (sfd1, 8)
4. Task_Master socket accepts bytes (from a playing robot process socket) from the socket with descriptor sfd2 using function as follows: sfd2 = accept (sfd1, &playRobotSockAddress, &playRlBytes)

The &playRIBytes refers to address for the maximum length of bytes from the playing robot. The playRobotSockAddres is the address of the data structure for the client (playing robot) socket.

5. Task_Master socket sends the bytes by using the function as follows:
send (sfd2, & playBuffer, playstreamlen, 0); /* Send total playstreamlen bytes from playBuffer using the socket sfd2.
6. Task_Master socket receives the bytes from the playing robots (e.g., for acknowledgement) by using the function as follows:
whiie (streamlength > 0 && streamlength < = playRIBytes) {streamlength = recv (sfd2, &ackBuffer, 200, 0)}; /* The recv () returns –1 if no more bytes are to be received. Bytes are received in ackBuffer */
7. Task_Master socket closes the socket by using a function as follows:
close ( );
8. A process, Task_Client in the playing robot creates a client 1 socket using a statement as follows:
sfd2 = socket ("/socket/clientsocket1", sockStream, 0).
The sfd2 is an unsigned integer for a socket descriptor. 'socket/serversocket1' is the path and file from which stream, socStream will be sent or received from play robots. 0 represents unrestricted permission to use the file.
9. Task_Client socket connects the sfd1 and data structure at the socket address, sockClientAddr by using the function at the server process as follows:
connect (sfd2, (struct sockClientaddr *)&remote, CIIBytes).
The CIIBytes is maximum length of the bytes in the play stream.
10. Task_Client socket sends the acknowledgement bytes by using the function as follows:
send (sfd2, & ackBuffer, playRIBytes, 0); /* Send total ackstreamlen bytes from ackBuffer using the socket sfd.
11. Task_Client socket receives the bytes from the playing robots (e.g., for acknowledgement) by using the function as follows:
while (streamlength > 0 && streamlength < = playstreamlen) { streamlength = recv (sfd2, &playBuffer, 200, 0)}; /* The recv () returns –1 if no more bytes are to be received. Bytes are received in playBuffer */

Application of sockets are as follows:

1. An *application* of sockets is to connect the tasks in the distributed environment of embedded systems. For example, a network interconnection or a card process connects a host-machine process.
2. A TCP/IP socket is another common *application* for the Internet. Another exemplary *application* of socket is a task receiving a byte stream of TCP/IP protocol at a mobile internet connection.
3. An exemplary *application* is when a task writes into a file at a computer or at a network using NFS protocol (network file system protocol).
4. Another *application* of the socket is the interconnection of a task or a section in a source set of tasks in an embedded system with another task at a destined set of tasks. The kernel has the socket-connecting functions with the codes specifying the source and destination sets and tasks.

The OS functions for socket in Unix are as following.

1. The socket ( ) [in place of open ( ) in case of pipe] gives a socket descriptor sfd. The socket ( ) enables its use from beginning of its allocated buffer at the socket address, its use with option and restrictions or permissions defined at the time of opening. A socket can be a stream, SOCK_STREAM or UDP datagram SOCK_DGRAM.
2. The unlink ( ) before the bind ( ).

3. The bind ( ) for binding a thread or task inserting bytes into the socket to the thread or task and deleting bytes from the socket. bind ( ) the socket descriptor to an address in the Unix domain. bind (sfd, (struct sockaddr *)&local, len); where len is string length. sockaddr is a data structure with a record of 16-bit unsigned num and a path for the file and a data structure struct sockaddr_un {unsigned short num; char path[108]; }
4. The listen (sfd, 16) function for listening 16 queued connections from the client socket.
5. The accept ( ) accepts the client connection and gives a second socket descriptor.
6. The recv ( ) function for deleting (reading) and receiving from the socket from the bottom of unread memory spaces in buffer. The buffer has messages after writing into the socket.
7. The send ( ) function for inserting (writing) and sending from the socket from the bottom of the memory spaces in the buffer filled after writing into the socket.
8. The close ( ) for closing the device to enable its use from beginning of its allocated buffer only after opening it again.

1. A Socket is an IPC for sending a byte stream or datagram from one or multiple task sockets to another task or server process socket as a bi-direction FIFO-like device using a protocol for transferring the bytes. Datagram provide protocol-header bytes along with the byte stream.
2. The sockets can be a client-server set of sockets (multiple processes and single server process) or peer-to-peer sockets IPC. A socket has a number of applications. An Internet connection socket is for virtual connection between two ports: one port at an IP address to another port at another IP address.
3. An OS provides the IPC functions for creating socket, unlinking, binding, listening, accepting, receiving, sending and closing.

7.16 RPC FUNCTIONS

RPC is a method used for connecting two remotely placed methods by first using a protocol for connecting the processes. It is used in the cases of distributed tasks.

The OS can provide for the use of RPCs. These permit distributed environment for the embedded systems. The RPC provides the IPC when a task is at system 1 and another task is at system 2. Both systems work in the peer-to-peer communication mode. Each system in peer-to-peer can make RPCs. The OS IPC function allows a function or method to run at another address space of shared network or an other remote computer. The process 1 makes the call to the function that is local or remote and the process 2 response is either remote or local in the process 1 to process 2 method call.

Summary

- A process is a computational unit that processes on a CPU under the state-control of a kernel in OS.
- A process may consist of multiple threads that define thread as a minimum unit for a scheduler to schedule it to run the CPU and provide other system resources. Unix provides for processes and their threads as light-weight processes. Light weight mean functions not dependent on functions like memory-management functions. Java use the threads.
- A single CPU system runs one process (or one thread of a process) at a time. A scheduler is a must to schedule a multitasking or multithreading system.

- A task is a computational unit or set of codes, actions or functions that processes on a CPU under the state-control of a kernel in OS.
- A task is similar to a process or thread. Each task is an independent process that takes control of the CPU when scheduled by a scheduler at an OS. No task can call another task. Each task and its state is recognized by its TCB (memory block) that holds information of the PC, memory map, the signal (message) dispatch table, signal mask, task ID, CPU state (registers and so on) and a kernel stack (for executing system calls and so on). A task is in one of the four states: idle, ready, blocked and running, that are controlled by the scheduler.
- Often the same data is used in two different tasks (or processes) and if another task interrupts before without completing the operation on that data, then the shared data problem arises. Disabling of interrupts till the completion of the operation by the first task, and then re-enabling interrupts, is one solution. Use of *semaphores* (as *tokens, mutex or counting semaphores*) is another efficient way for solving the shared data problem and running critical section codes. Use of lock functions and spin-locks are also provided in the OSes.
- A buffer is a memory block for a queue or stream of bytes between an output source and input sink. [For example, between the tasks, files, computer and printer, physical devices and network.] It has to be bounded between two limits. It cannot be unlimited or infinite. For example, a print buffer cannot accept unlimited output from a computer. The bounded buffer problem is of synchronizing the source and sink. A *producer cannot keep on producing beyond a limit if consumers do not consume. The consumers cannot keep consuming unless the producer keeps producing.* Counting semaphores provide solution to this problem.
- There has been POSIX IEEE standardization of the OS and IPC functions, for example, the P and V semaphore POSIX functions. These function are event notifying, resource key, mutex and counting semaphores.
- *The priority inversion* problem and deadlock situation can arise in certain situations when using a semaphore. An OS should be such that it can take care of it by having the appropriate provisions to avoid these situations. Certain OSes provide mutex semaphores such that priority inversion problem does not arise.
- The OS functions handle IPCs between the multiple tasks.
- The OS provides for the following IPCs: signals, semaphores, queues, mailboxes, pipes, sockets and RPCs.
- A mailbox may either provide for only one message or multiple messages in an RTOS.
- A pipe is a queue or stream of messages that connects the two tasks and that uses the functions as used in a device.
- The sockets are used in networks or client-server-like communication between the tasks using functions as used for the devices. The RPCs are used for the case of distributed tasks.

Keywords and their Definitions

Buffer	:	A memory block for a queue or network stack or pipe or stream of bytes between an output source and input sink, for example, between the tasks, files, computer and printer, physical devices and network.
Counting semaphore	:	A semaphore in which the value of which can be initialized to an 8- or 16- or 32-bit integer and that is decremented and incremented. A task does not block if its value is found to be >0 and the task blocks if its value is found to be 0.
Critical section	:	A section in a task the execution of which should block execution of another such section in another task, for example, when a buffer in printer is shared between two or more tasks.
Deadlock situation	:	A task waiting for the release of a semaphore from a task and another a different task waiting for another semaphore release to run. None of these is able to proceed further due to circular dependency. An OS can take care of this by appropriate provisions.
Interprocess communication	:	A mechanism from one task (or process) sending signal or messages or event notification from one task to the system and which the OS communicates to another task. Using IPC mechanism and functions a task uses signals, exceptions, semaphores, queues, mailboxes, pipes, sockets and RPCs.
Mailbox	:	A message(s) from a task that is addressed for another task.
Message queue	:	A task sending the multiple messages into a queue for use by another task(s) using queue messages as an input.
Mutex	:	The special variable and mechanism used to take note of certain actions to prevent any task or process from proceeding further and at the same time let another task exclusively proceed further. Mutex helps in mutual exclusion of one task with respect to another by a scheduler in the multitasking operations.
P and V semaphores	:	The semaphore functions defined in a POSIX IEEE standard to be used as event notification or mutex or counting semaphore, or to solve the classical producer–consumer problem when using a bounded buffer.
Pipe	:	A device for use by the task for sending the messages and another task using the device receives the messages as stream. A pipe is a unidirectional device.
Priority inversion	:	A problem in which a low priority task inadvertently does not release the process for a higher priority task. An operating system can take care of this by appropriate provisions.
Process	:	A code that has its independent PC values and an independent stack. A single CPU system runs one process (or one thread of a process) at a time. A *process* is a concept (abstraction). It defines *a sequentially executing (running) program and its state. A state*, during the running of a process, is represented by its status (running, blocked or finished), its control block, called process control block (PCB) or *process structure*, its data, objects and resources.
Remote procedure call	:	A method used for connecting two remotely placed methods by first using a protocol for connecting the processes. It is used in the cases of distributed tasks.
Semaphore	:	A special variable operated by the OS functions which are used to take note of certain actions to prevent another task or process from proceeding further.
Shared data problem	:	If a variable is used in two different processes (tasks) and if another task interrupts before the operation on that data is completed, then the shared data problem arises.
Signal	:	A function to call a signal handler by interrupting the processes. It uses INT n SWI n instruction, where n defines the handler, which should run.
Socket	:	It provides the logical link using a protocol between the tasks in a client-server or peer-to-peer environment. It enables a bi-directional stream or datagram or network stack.
Synchronization	:	To let each section of codes, tasks and ISRs run and gain access to the CPU one after the other sequentially or concurrently, following a scheduling strategy, so that there is a predictable operation at any instance.

Task	A task is for the service of specific actions and may also correspond to the codes, which execute for an interrupt. A task is an independent process that takes control of the CPU when scheduled by a scheduler at an OS. Every task has a TCB.
Task control block	A memory block that holds information of the PC, memory map, the signal (message) dispatch table, signal mask, task ID, CPU state (registers and so on), and a kernel stack (for executing system calls and so on).
Task state	A state of a task that changes on scheduler directions. A task at an instance can be in one of the four states, *idle*, *ready*, *blocked* and *running* that are controlled by the scheduler.
Thread	A minimum unit for a scheduler to schedule the CPU and other system resources. A process may consist of multiple threads. A thread has an independent process control block like a TCB and a thread executes codes under the control of a scheduler. It is a light weight process.

Review Questions

1. How does a data output generated by a process transfer to another using an IPC?
2. What are the parameters at a TCB of a task? Why should each task have distinct TCB?
3. What are the states of a task? Which is the entity controlling (scheduling) the transitions from one state to another in a task?
4. Define critical section of a task. What are the ways by which the critical section run by blocking other process(es)?
5. How is data (shared variables) shielded in a critical section of a process before being operated and changed by anther higher priority process that starts execution before the process finishes?
6. How does use of a counting semaphore differ from a mutex? How is a counting semaphore used?
7. Give an example of a deadlock situation during multiprocessing (multitasking) execution.
8. What is the advantage and disadvantage of disabling interrupts during the running of a critical section of a process?
9. Explain the term multitasking OS and multitasking scheduler.
10. Each process or task has an endless (infinite) loop in a pre-emptive scheduler. How does the control of resources transfer from one task to another?
11. What is an *exception* and how is an error-handling task executed on throwing the exception?
12. How do functions differ from ISRs, tasks, threads and processes? Why is an ISR not permitted to use the IPC pend wait functions.
13. List the features of P and V semaphores and how these are used as a resource key, as a counting semaphore and as a mutex.
14. What are the situations, which lead to priority inversion problems? How does an OS solve this problem by a priority inheritence mechanism?
15. What is meant by a pipe? How does a pipe may differ from a queue?
16. What is meant by a spinning lock? Explain the situation in which the use of the spin lock mechanism would be highly useful to lock the transfer of control to an higher priority task?
17. What is a mailbox? How does a mailbox pass a message during an IPC?
18. When are the sockets used for IPCs? List four examples. When are RPCs used? List two examples.
19. What are the analogies between process, task and thread? Also list the differences between the process, task and thread.

Practice Exercises

20. Design a table to clearly distinguish the cases when there is concurrent processing of processes, when tasks and when threads by using a scheduler.
21. Make a table similar to Table 7.1 to clearly distinguish ISRs, ISTs and tasks.
22. What is the advantage of using a *signal* as an IPC? List the situations which warrant use of signals.
23. List five exemplary applications of solutions to the bounded buffer problem using P and V mutex semaphores.
24. Every tenth second a burst of 64 kB arrives at 512 kbps in an interval of 100 seconds. Is an input buffer required? If yes, then how much? If yes, then write a program to use the buffer using P and V semaphores.
25. Use Web search to understand an IEEE-accepted standard POSIX 1000.3b in detail.
26. Can different IPCs be used? Given the choice, how will you select an IPC from signal, semaphore, queue or mailbox?
27. List the tasks in the automatic chocolate-vending machine (Example 1.10.2). List the IPC functions required and their uses in the ACVM.
28. List the processes used in smart card (Example 1.10.3). How does the card communicate with the host using the sockets? List the IPC functions required and their uses in the smart card.
29. List the tasks in the digital camera (Example 1.10.4). List the IPC functions required and their uses in the camera.
30. List the processes in the smart mobile phone (Example 1.10.5). The display process has multiple threads in the phone. List the threads. List the IPC functions required and their uses in the phone.
31. List the processes in the PDA (Example 1.10.6). Assume that PDA services the events by the ISRs and signal handlers using a queue of the events. How can this be done? Show it by a diagram.
32. List the processes in the director of OPRs (Example 1.10.7). List the processes in eight playing robots in OPRs.