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

Input and Output Devices

Input and output devices allow the computer system to interact with the outside world by moving data *into* and *out of* the system. An *input device* is used to bring data into the system. Some input devices are:

- Keyboard
- Mouse
- Microphone
- Bar code reader
- Graphics tablet

An *output device* is used to send data out of the system. Some output devices are:

- Monitor
- Printer
- Speaker

Input/output devices are usually called I/O devices. They are directly connected to an electronic module inside the systems unit called a **device controller**. For example, the speakers of a multimedia computer system are directly connected to a device controller called an audio card (such as a Soundblaster), which in turn is connected to the rest of the system.

Sometimes secondary memory devices like the hard disk are called I/O devices (because they move data in and out of main memory.) What counts as an I/O device depends on context. To a user, an I/O device is something outside of the system box. To a programmer, everything outside of the processor and main memory looks like an I/O devices. To an engineer working on the design of a processor, everything outside of the processor is an I/O device.

A computer that is dedicated to running a program that controls another device is an **embedded system**. An embedded system is usually embedded inside the device it controls. Usually they run just one program that is permanently kept in a special kind of main memory called ROM (for Read Only Memory). More processor chips are sold per year for embedded systems than for all other purposes.

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.

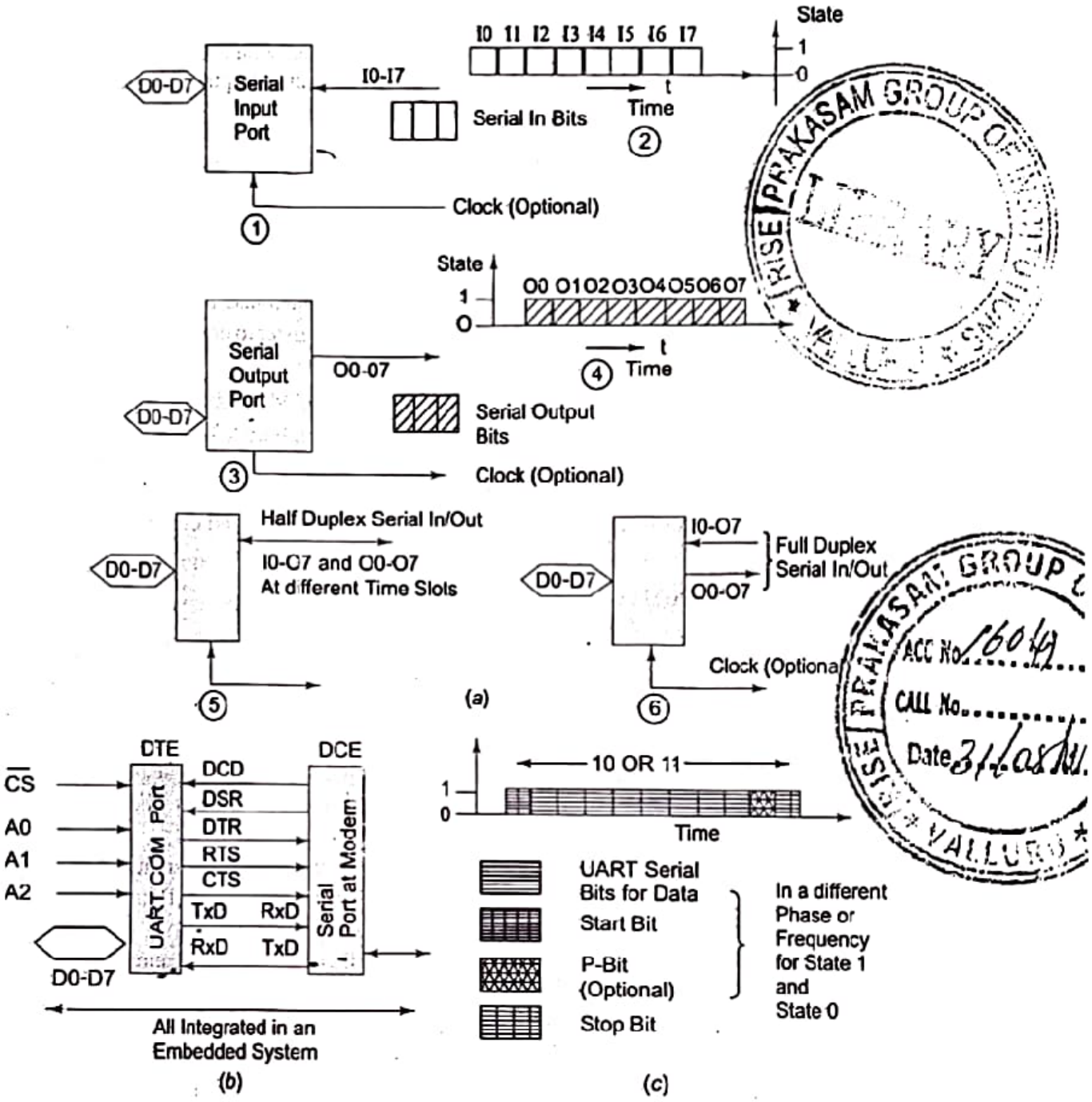


Fig. 3.1 (*a*) Input serial port, Output Serial port, Bi-directional half-duplex serial port, and Bi-directional full-duplex serial port (*b*) Handshaking signals at COM port in computer and (*c*) a UART serial port bits

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.

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.

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.ΔT[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.

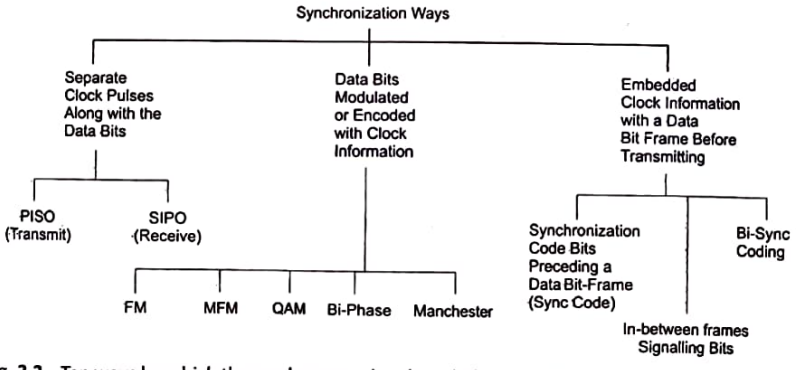


Fig. 3.2 Ten ways by which the synchronous signals with the clocking information transmit from a master device to slave device

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).

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].

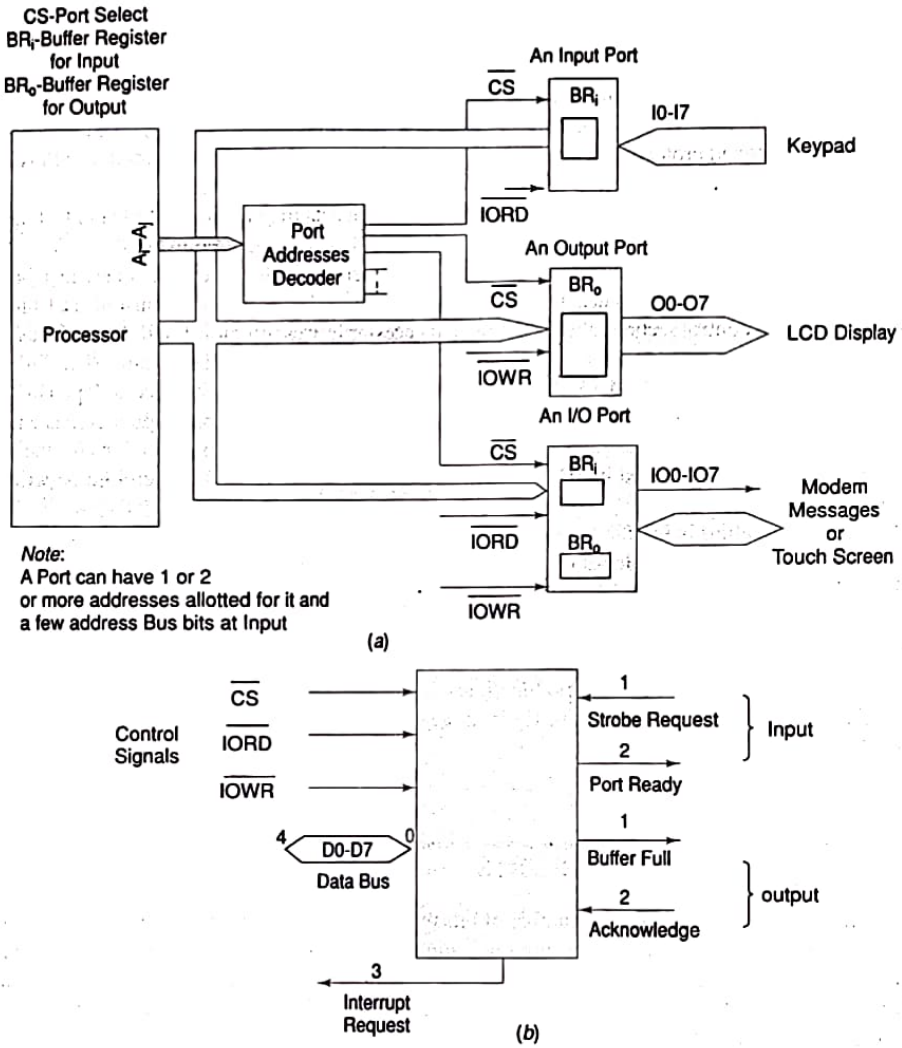


Fig. 3.4 (*a*) Parallel input port, output port, and a bi-directional port for connecting the device (*b*) The handshaking signals when used by the IO ports

Example 3.4

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) $\times$ 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 $p \times 2^n \times \delta T$ 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*)

ISR Concept

* Interrupt means Event, Which invites the attention of processor for some action on the hard Ware or soft Ware 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 hard Ware interrupt
2. When soft Ware run-time exception conditions is detected, either proce -ssor h/w or s/w instruction generates an interrupt. This interrupt is called Software interrupt or trap or exception.

* In response to the interrupt, the routine or program, Which is running at present gets interrupted and an ISR is executed.

Features of ISR

1. An ISR call due to interrupt excutes an event. Event can occur at any moment and event occurences are asynchronous.
2. ISR call is event-based diversion from the current sequence of instructions to the another sequence of instructions. This sequence of instructions executes till the return instruction.
3. Event can be a device or port event or s/w computational exceptional condition detected by H/w or detected by program, Which throws the exception.

4. An event can be signalled by s/w interrupt instruction SWI Used in device driving functions Create( ), Open( ), etc.
5. An interrupt service mechanism exists in a System to call the ISRs from multiple Sources.
6. Diversion to ISR may or may not take place on finishing the execution of any instruction in the presently running routine.

Interrupt Sources

* Hard Ware sources can be from internal devices or external pheripherals, Which interrupt the ongoing routine and there by cause diversion to corresponding ISR.
* software sources for interrupt are related to
  (i) processor detecting computational error for an illegal op-code during execution or
  (ii) execution of an SWI instruction to cause processor-interrupt of on going routine.
* There may be some other special types of sources provided in the System are

1. Hardware Interrupts Related to Internal devices

There are no. of H/w interrupt sources which can interrupt an on going program. These are processor or micro controller or internal devices h/w specific.

Ex:- Parallel port, UART serial receiver port, ADC start of conversion, ADC end, 3. synchronous receiver byte completion, Etc.

2. Hardware Interrupts related to External Devices - 1

There can be external hardware interrupt source for interrupting an on going program that also provides the ISR address or Vector address or interrupt - type information through the data bus.

Ex:- INTR in 8086 and 80x86.

3. Hardware interrupts Related to External Devices - 2

External hardware interrupts with their ISR Vector address are process or micro controller - specific interrupt of an on going program. External interruption - source does not send interrupt - type or ISR address - related information.

Ex:- Non - maskable pin, within first few clock cycles unmaskable declarable pin but otherwise maskable, Maskable pin [M for] INT0 & INT1

4. software Error - Related Hardware interrupts

These 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 s/w) 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).

Ex:- Division by zero detection (or trap) by hard ware, over flow by h/w, Under flow by h/w, Illegal opcode by H/w.

5. Soft ware Instruction - Related Interrupts sources

A program can also handle specific computational errors or run - time conditions or signalling some conditions. processors provide for software instructions related to the traps, signals or exceptions.

(i) There are certain software instructions for interrupting and then diverting to the ISR also called the signal handler. These are used for signalling to another routine from an on going routine or task or thread.

(ii) Software instructions are also used for trapping some run - time error conditions and executing exceptional handlers on catching the exceptions.

INTERRUPT SERVICING (HANDLING) MECHANISM

* Each system has an interrupt Servicing (handling) Mechanism.
* The os also provides for mechanism for interrupt handling.

(i) 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.
* Vectoring is as per the provisions in interrupt-handling mechanism. The Various mechanism 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 instructim, 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.

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 micro controller, each internal device interrupt source or source-group has a seperate ISR-VECTADDR address. Each external interrupt pin has seperate ISR-VECTADDR.

2. In 80x86 processor architecture, a software instruction, for example 'INTn' explicitly also defines the type of interrupt and the type of defined the ISR-VECTADDR this mechanism results in the handling of n no. of exception handling routine or ISRs for n interrupt types.
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 by is a Common ISR-VECTADDR for Each exception or signal or trap generated using SWI instruction.

A group of interrupt sources having common Vector Address

A group of Interrupt sources having common Vector Address

A source group in the hardWare may have the same ISR-VECTADDR. 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.

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 or jump instruction to the ISR first instruction. or ISR short code With call to the full code of the ISR or for fetching the bytes for finding the ISR address.

classification of Interrupt

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 service may be temperarly disable to let high priority ISRs be executed first Uninterrupted.

3. Non-maskable only When defined so within few clock cycles after reset : For example, an external interrupt pin, XSIRQ interrupt, in 68HC11. XSIRQ interrupt is non-maskable only when defined so within few clock cycle after 68HC11 is reset.

Multiple Interrupts

Multiple Interrupt calls

When there are multiple interrupt sources, each occurance of interrupt from a source (or source group) is identifiable from a bit in the status register and/or in the IPR (Interrupt-pending resistor).

* There can be interrupt service calls in succession case higher priority interrupt sources activate in succession. Then return from high priority ISR is to lower priority pending ISR.

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

1. Certain processors donot provide for in-between routine diversion to higher priority interrupts and presume that all interrupts of priority greater than the presently running routine are masked till the end of the routine.
2. Certain processors permit in-between routine diversion to higher priority interrupts. These processors provide, in order to prevent diversion

greater than the presently running routine are ...
end of the routine.

2. Certain processors permit in-between routine diversion to higher priority interrupts. 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 levels bit. These process -ors also provisions selective diversion by provisioning for masking the interrupt service selectively by secondary level bits.

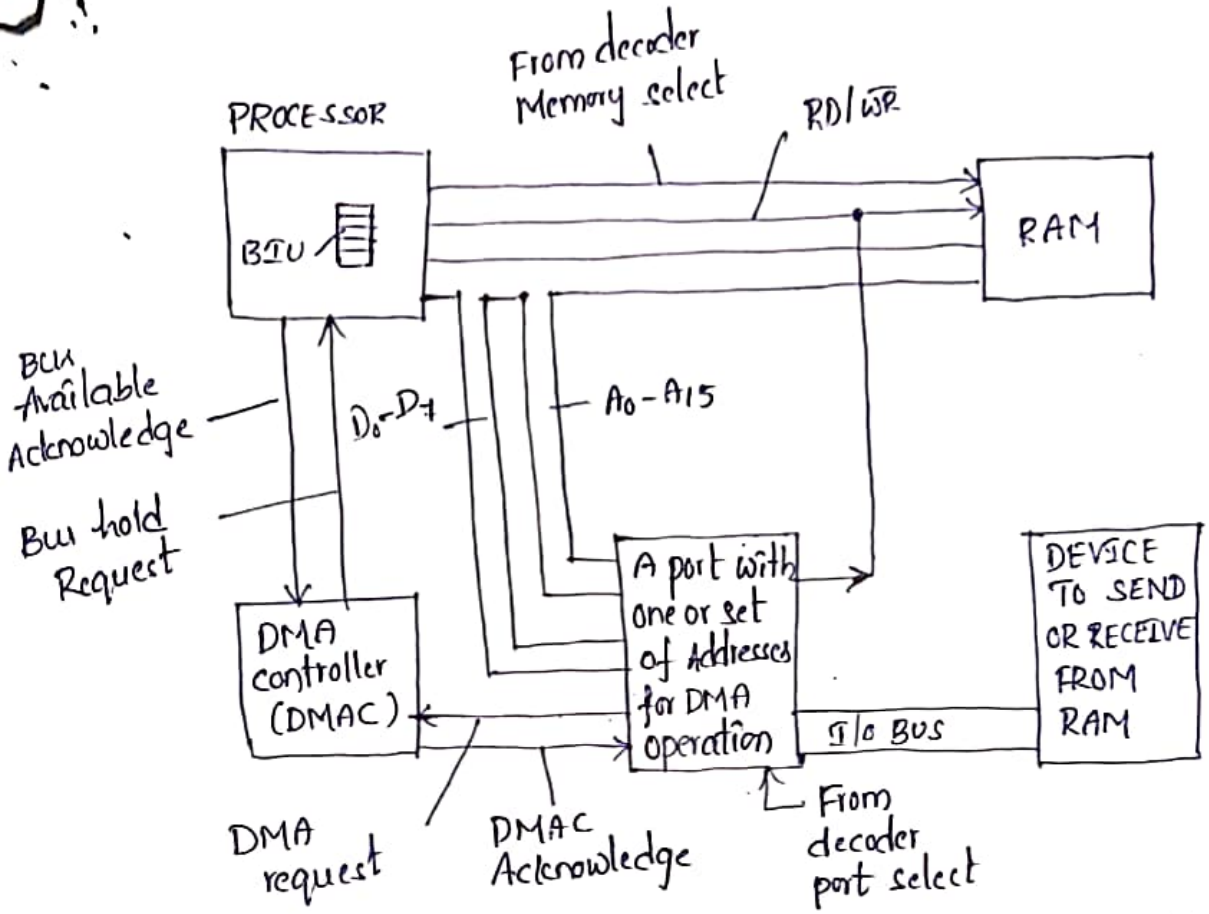
1-Hardware Assigned priorities

There is assigned priority order by hardware

* ARM7 provides for two types of external interrupt sources (request) IRQs and FIQs (fast interrupt requests).
* 8051 provides for priority order in order of interrupt vector address. lower address has highest and higher has the lower priority.
* Interrupts in 80x86 are assigned priority order according to Inte -rrupt-types. Interrupt of type '0' has highest priority and 225 has lowest assigned priority.

Direct Memory Access

* The DMA is used to transfer data b/w hard disk system memory.
* When the I/O's are needed for large amount data from a peripheral device to the memory address in the system or large amount of data is to be transferred by the I/O's, 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 b/w the external device and system or b/w two systems.
* The Device DMAC (DMA controller) data transfer occurs efficiently b/w the I/O Devices and system memory.
* The following fig. shows the interconnections using the DMAC. It also shows the buses and control signals b/w the processor, memory, DMAC and data transfering I/O device.



System buses not accessible by processor internal address and data buses during acknowledge active.

fig: The buses and contrd signals between the processor, memory, DMA Controller and data-transfering I/o device.

* The DMAC sends a hold request to the CPU and the CPU ackno -wledges 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.

Use of DMAC

Whenever a DMA request is made to the DMAC for the I/O's, the DMAC is first initialized. It is programmed for

(i) read or Write
(ii) mode (bytes, burst or bulk) of DMA transfer
(iii) Total No. of bytes to be transfered and
(iv) starting memory address.

* Whenever a DMA request by the external device is made to the DMAC, the CPU is requested the DMA transfers by DMAC at the start to initiate the DMA and at the end to notify the end of the DMA by DMAC.

* A DMAC may also provide memory access to multiple channels.

Device Driver programming

* A system has no. of physical devices, A device may have multiple functions each device function requires a driver.
* Examples of multiple functions in a device are as follows.
  (i) A timer device performs timing functions as well as counting functions. It also performs the delay function and periodic system calls.
  (ii) A transceiver device transmits as well as receives.
  (iii) Voice - data - fax modem device has transmitting as well as receiving as functions for voice, fax as well as data.
* The drivers has following features
  (i) The driver provides a software layer (Interface) b/w 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.
  Ex:- For mailing system b/w layers, Network interface Card is used.

(iii) The driver facilitates the use of device by executing an ISR: simple commands from a task or function can them 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, address, registers, bits and flags used by the device.
Ex:- The system clock is to be set to tick every 10,000μs.

* 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 I/o buses and the addresses of the device register in the specific hardware.
  → Writing physical device - Driving ISRs in a system
  → Virtual Device drivers
  → Parallel port drivers in a System
  → Serial port drivers in a system
  → Device drivers for Internal programmable Timing devices.
  → Linux Internals as device drivers and Network functions.

Programming in Embedded C

* Whenever the Convential 'c' language and its extensions are used for programming Embedded systems, it is referred as "Embedd C' program.
* Desktop application development using 'c' language for a particular os platformer.
* Desktop computers contain Working memory in the range of Megabyt and storage memory in the range of Giga bytes.
* Embedded systems are limited in which both storage and Working memory resources.

'C' V/s 'Embedded C':-

* 'c' is a Well structured, Well defined and standardised general purpose programming language with extensive bit manipulation support.
* 'c' offers a combination of the features of high level language and assembly and helps in hard Ware access programming as Well as business packages developments.
* The Conventional 'c' language follows ANSI standard ~~and~~ and it incorporates various library files for different operating systems.
* A plat form (os) specific application, known as compiler is used for the conversion of program Written in 'c' to the target processor specific binary files.
* Hence it is plat form specific development.
* Embedded 'c' can be considered as a subset of conventional 'c' language.
* Embedded 'c' supports all 'c' instructions and incorporates a few target processor specific functions / instructions.
* The Implementation of target processor / controller specific functions / instructions depends up on the processor / controller as well as the supported Cross - compiler for the particular Embedded 'c' language.
* A software program called 'cross - compiler' is used for the conversion of programs Written in Embedded 'c' to target proce -ssor / controller specific instructions. (machine language).
* The standard ANSI 'c' library implementation is always tailored to the target processor / controller library files in Embedded 'c'.