Module No: 4

8051 SERIAL PORT PROGRAMMING IN ASSEMBLY AND C

Computers transfer data in two ways: parallel and serial. In parallel data transfers, often 8 or more lines (wire conductors) are used to transfer data to a device that is only a few feet away. Examples of parallel transfers are printers and hard disks; each uses cables with many wire strips. Although in such cases a lot of data can be transferred in a short amount of time by using many wires in parallel, the distance cannot be great. To transfer to a device located many meters away, the serial method is used. In serial communication, the data is sent one bit at a time, in contrast to parallel communication, in which the data is sent a byte or more at a time. The 8051 has serial communication capability built into it, thereby making possible fast data transfer using only a few wires.

BASICS OF SERIAL COMMUNICATION

When a microcontroller communicates with the outside world, it provides the data in byte-sized chunks. In some cases, such as printers, the information is simply grabbed from the 8-bit data bus and presented to the 8-bit data bus of the printer. This can work only if the cable is not too long, since long cables diminish and even distort signals. Furthermore, an 8-bit data path is expensive. For these reasons, serial communication is used for transferring data between two systems located at distances of hundreds of feet to millions of miles apart. Figure 10-1 diagrams serial versus parallel data transfers.

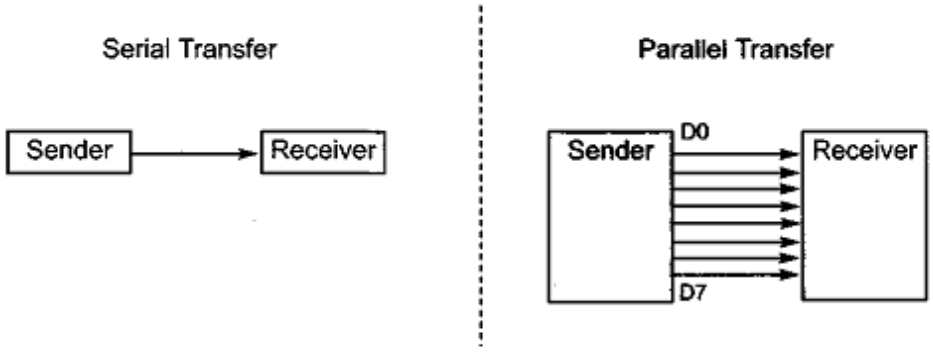


Fig: Serial vs Parallel transfer

The fact that serial communication uses a single data line instead of the 8-bit data line of parallel communication not only makes it much cheaper but also enables two computers located in two

different cities to communicate over the telephone. For serial data communication to work, the byte of data must be converted to serial bits using a parallel-in-serial-out shift register; then it can be transmitted over a single data line. This also means that at the receiving end there must be a serial-in-parallel-out shift register to receive the serial data and pack them into a byte. Of course, if data is to be transferred on the telephone line, it must be converted from Os and 1s to audio tones, which are sinusoidal-shaped signals. This conversion is performed by a peripheral device called a *modem*, which stands for "modulator/demodulator."

When the distance is short, the digital signal can be transferred as it is on a simple wire and requires no modulation. This is how IBM PC keyboards transfer data to the motherboard. However, for long-distance data transfers using communication lines such as a telephone, serial data communication requires a modem to *modulate* (convert from Os and 1 s to audio tones) and *demodulate* (converting from audio tones to Os and 1 s).

Serial data communication uses two methods, Asynchronous and Synchronous. The *synchronous* method transfers a block of data (characters) at a time, while the *asynchronous* method transfers a single byte at a time. It is possible to write software to use either of these methods, but the programs can be tedious and long. For this reason, there are special IC chips made by many manufacturers for serial data communications. These chips are commonly referred to as UART (universal asynchronous receiver-transmitter) and USART (universal synchronous-asynchronous receiver-transmitter). The 8051 chip has a built-in UART.

Half- and full-duplex transmission

In data transmission if the data can be transmitted and received, it is a *duplex* transmission. This is in contrast to *simplex* transmissions such as with printers, in which the computer only sends data. Duplex transmissions can be half or full duplex, depending on whether or not the data transfer can be simultaneous. If data is transmitted one way at a time, it is referred to as *half duplex.* If the data can go both ways at the same time, it *is full duplex.* Of course, full duplex requires two wire conductors for the data lines (in addition to the signal ground), one for transmission and one for reception, in order to transfer and receive data simultaneously.

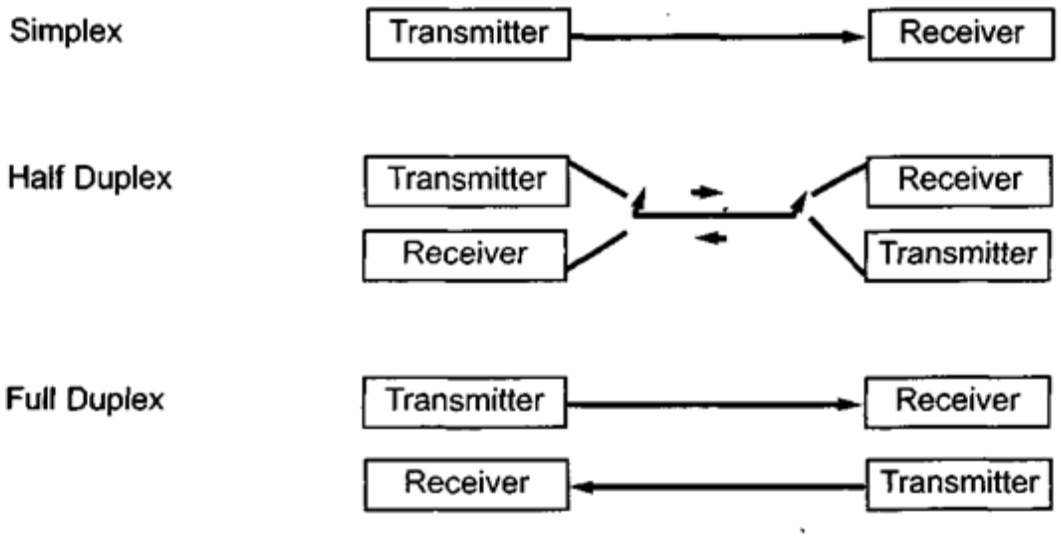


Fig: Simplex, duplex transfer

Asynchronous serial communication and data framing

Asynchronous serial data communication is widely used for character-oriented transmissions, while block-oriented data transfers use the synchronous method. In the asynchronous method, each character is placed between start and stop bits. This is called *framing.* In data framing for asynchronous communications, the data, such as ASCII characters, are packed between a start bit and a stop bit. The start bit is always one bit, but the stop bit can be one or two bits. The start bit is always a 0 (low) and the stop bit(s) is 1 (high). For example, look at Figure in which the ASCII character "A" (8-bit binary 0100 0001) is framed between the start bit and a single stop bit. Notice that the LSB is sent out first.

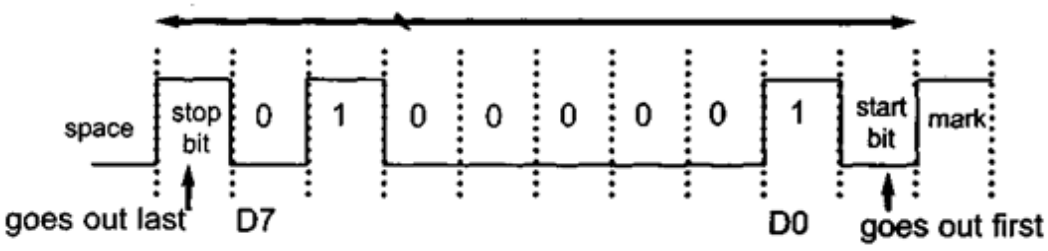


Fig: Data framing

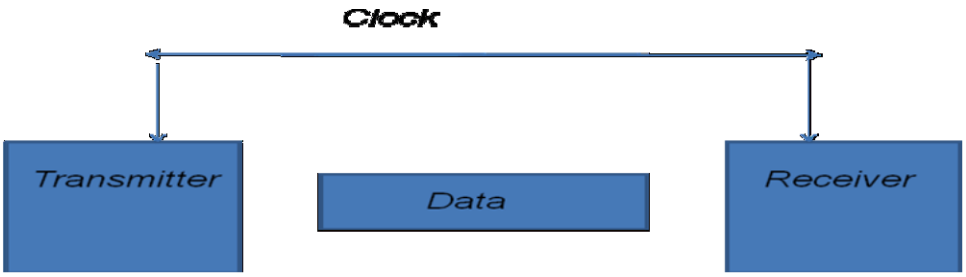
when there is no transfer, the signal is 1 (high), which is referred to as *mark.* The 0 (low) is referred to as *space.* Notice that the transmission begins with a start bit followed by DO, which

is the LSB, then the rest of the bits until the MSB (D7), and finally, the one stop bit indicating the end of the character "A".

In asynchronous serial communications, peripheral chips and modems can be programmed for data that is 7 or 8 bits wide. This is in addition to the number of stop bits, 1 or 2. While in older systems ASCII characters were 7-bit, in recent years, due to the extended ASCII characters, 8-bit data has become common. In some older systems, due to the slowness of the receiving mechanical device, two stop bits were used to give the device sufficient time to organize itself before transmission of the next byte. In modern PCs however, the use of one stop bit is standard. Assuming that we are transferring a text file of ASCII characters using 1 stop bit, we have a total of 10 bits for each character: 8 bits for the ASCII code, and 1 bit each for the start and stop bits. Therefore, for each 8-bit character there are an extra 2 bits, which gives 20% overhead.

In some systems, the parity bit of the character byte is included in the data frame in order to maintain data integrity. This means that for each character (7- or 8-bit, depending on the system) we have a single parity bit in addition to start and stop bits. The parity bit is odd or even. In the case of an odd-parity bit the number of data bits, including the parity bit, has an odd number of 1s. Similarly, in an even-parity bit system the total number of bits, including the parity bit, is even. For example, the ASCII character "A", binary 0100 0001, has 0 for the even-parity bit. UART chips allow programming of the parity bit for odd-, even-, and no-parity options.

In Asynchronous Serial Data Communication Pins TxD (P3.1) and RxD (P3.0) are used for transmitting and receiving the data serially. Figure below shows synchronous serial data communication which uses a common clock for synchronization of transmitter and receiver.



Data transfer rate

The rate of data transfer in serial data communication is stated in *bps* (bits per second). Another widely used terminology for bps is *baud rate*. However, the baud and bps rates are not

necessarily equal. This is due to the fact that baud rate is the modem terminology and is defined as the number of signal changes per second. In modems a single change of signal, sometimes transfers several bits of data. As far as the conductor wire is concerned, the baud rate and bps are the same, and for this reason in this book we use the terms bps and baud interchangeably.

The data transfer rate of a given computer system depends on communication ports incorporated into that system. For example, the early IBM PC/XT could transfer data at the rate of 100 to 9600 bps. In recent years, however, Pentium-based PCs transfer data at rates as high as 56K bps. It must be noted that in asynchronous serial data communication, the baud rate is generally limited to 100,000 bps.

RS232 standards

To allow compatibility among data communication equipment made by various manufacturers, an interfacing standard called RS232 was set by the Electronics Industries Association (EIA) in 1960. In 1963 it was modified and called RS232A. RS232B and RS232C were issued in 1965 and 1969, respectively. Today, RS232 is the most widely used serial I/O interfacing standard. This standard is used in PCs and numerous types of equipment. However, since the standard was set long before the advent of the TTL logic family, its input and output voltage levels are not TTL compatible. In RS232, a 1 is represented by -3 to -25 V, while a 0 bit is +3 to +25 V, making -3 to +3 undefined. For this reason, to connect any RS232 to a micro controller system we must use voltage converters such as MAX232 to convert the TTL logic levels to the RS232 voltage levels, and vice versa. MAX232 IC chips are commonly referred to as line drivers.

RS232 pins

Figure shows DB 25 connector. But IBM introduced the DB-9 version of the serial I/O standard, which uses 9 pins only,

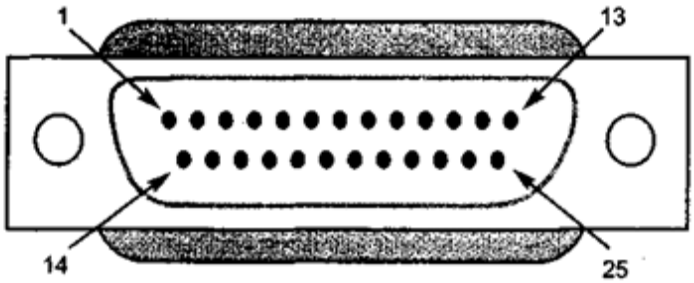


Fig: DB 25 connector

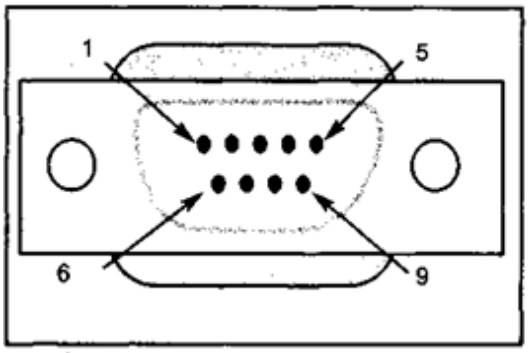


Fig: DB 9 connector

RS232 Pins (DB-25)

Pin	Description
1	Protective ground
2	Transmitted data (TxD)
3	Received data (RxD)
4	Request to send ($\overline{\text{RTS}}$)
5	Clear to send ($\overline{\text{CTS}}$)
6	Data set ready ($\overline{\text{DSR}}$)
7	Signal ground (GND)
8	Data carrier detect ($\overline{\text{DCD}}$)
9/10	Reserved for data testing
11	Unassigned
12	Secondary data carrier detect
13	Secondary clear to send
14	Secondary transmitted data
15	Transmit signal element timing
16	Secondary received data
17	Receive signal element timing
18	Unassigned
19	Secondary request to send
20	Data terminal ready ($\overline{\text{DTR}}$)
21	Signal quality detector
22	Ring indicator
23	Data signal rate select
24	Transmit signal element timing
25	Unassigned

fig: Pin description of DB-25

Pin	Description
1	Data carrier detect ($\overline{DCD}$)
2	Received data (RxD)
3	Transmitted data (TxD)
4	Data terminal ready (DTR)
5	Signal ground (GND)
6	Data set ready ($\overline{DSR}$)
7	Request to send ($\overline{RTS}$)
8	Clear to send ($\overline{CTS}$)
9	Ring indicator (RI)

fig: Pin description of DB-9

Data Communication

Current terminology classifies data communication equipment as DTE (data terminal equipment) or DCE (data communication equipment). DTE refers to terminals and computers that send and receivedata, while DCE refers to communication equipment, such as modems, that are responsible for transferring the data.

The simplest connection between a PC and microcontroller requires a minimum of three pins, TxD, RxD, and ground, Notice in that figure that the RxD and TxD pins are interchanged.

Examining RS232 handshaking signals

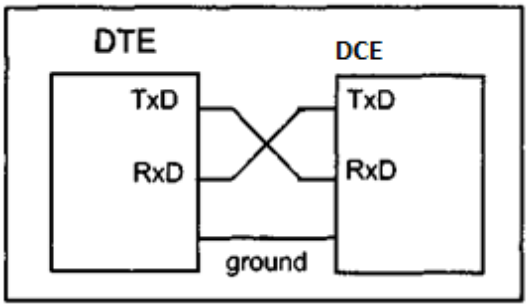


Fig:Null modem connection

The simplest connection between a PC and microcontroller requires a minimum of three pins, TxD,RxD, and ground, Notice in that figure that the RxD and TxD pins are interchanged.

Examining RS232 handshaking signals

To ensure fast and reliable data transmission between two devices, the data transfer must be coordinated. Just as in the case of the printer, because the receiving device in serial data communication may have no room for the data, there must be a way to inform the sender to stop sending data. Many of the pins of the RS-232 connector are used for handshaking signals.

1.DTR(data terminal ready). When a terminal (or a PC COM port) is turned on, after going through a self-test, it sends out signal DTR to indicate that it is ready for communication. If there is something wrong with the COM port, this signal will not be activated. This is an active-low signal and can be used to inform the modem that the computer is alive and kicking. This is an output pin from DTE (PC COM port) and an input to the modem.DSR (data set ready).

2. When DCE (modem) is turned on and has gone through the self-test, it asserts DSR to indicate that it is ready to communicate. Thus, it is an output from the modem (DCE) and input to the PC (DTE). This is an active- low signal. If for any reason the modem cannot make a connection to the telephone, this signal remains inactive, indicating to the PC (or terminal) that it cannot accept or send data.

3. RTS (request to send). When the DTE device (such as a PC) has a byte to transmit, it asserts RTS to signal the modem that it has a byte of data to transmit. RTS is an active-low output from the DTE and an input to the modem.

4.CTS (clear to send). In response to RTS, when the modem has room for storing the data it is to receive, it sends out signal CTS to the DTE (PC) to indicate that it can receive the data now. This input signal to the DTE is used by the DTE to start transmission.

5. DCD (carrier detect, or DCD, data carrier detect). The modem asserts signal DCD to inform the DTE (PC) that a valid carrier has been detected and that contact between it and the other modem is established. Therefore, DCD is an output from the modern and an input to the PC (DTE).

6.RI (ring indicator). An output from the modem (DCE) and an input to a PC (DTE) indicates that the telephone is ringing. It goes on and off in synchronization with the ringing sound. Of the six handshake signals, this is the least often used, due to the fact that modems take care of

answering the phone.

However, if the PC is in charge of answering the phone, this signal can be used.

While signals DTE and DSR are used by the PC and modem, respectively, to indicate that they are alive and well, it is RTS and CTS that actually control the flow of data. When the PC wants to send data it asserts RTS, and in response, if the modem is ready (has room) to accept the data, it sends back CTS. If, for lack of room, the modem does not activate CTS, the PC will deassert DTR and try again. RTS and CTS are also referred to as hardware control flow signals.

Serial Interface

The serial port of 8051 is full duplex, i.e., it can transmit and receive simultaneously. The register SBUF is used to hold the data. The special function register SBUF is physically two registers. One is, write-only and is used to hold data to be transmitted out of the 8051 via TXD. The other is, read-only and holds the received data from external sources via RXD. Both mutually exclusive registers have the same address 099H.

Serial Port Control Register (SCON)

SCON Register

SM0	SM1	SM2	REN	TB8	RB8	TI	RI

Serial control register: SCON

SM0, SM1 : Serial port mode specifier

Register SCON controls serial data communication. Address: 098H (Bit addressable)

Mode select bits

SM0	SM1	MODE
0	0	Mode0
0	1	Mode1
1	0	Mode2
1	1	M ode3

SM2:used for multiprocessor communication.

REN: set or cleared by software to enable/disable reception.

TB8: Transmitted bit 8,not widely used.

RB8:Received bit 8.

TI:Transmit interrupt flag –set by the hardware at the beginning of the stop bit in mode 1, must be cleared by software.

RI: Receive interrupt flag –set by the hardware halfway through the stop bit time in mode1, must be cleared by software.

Programming the 8051 to transfer data serially

In programming the 8051 to transfer character bytes serially, the following steps must be taken.

1. The TMOD register is loaded with the value 20H, indicating the use of Timer 1 in mode 2 (8-bit auto-reload) to set the baud rate.
2. The TH1 is loaded with one of the values in Table 10-4 to set the baud rate for serial data transfer (assuming XTAL = 11.0592 MHz).
3. The SCON register is loaded with the value 50H, indicating serial mode 1, where an 8-bit data is framed with start and stop bits.
4. TR1 is set to 1 to start Timer 1.
5. TI is cleared by the "CLR TI" instruction.
6. The character byte to be transferred serially is written into the SBUF register.
7. The TI flag bit is monitored with the use of the instruction " JNB TI, xx" to see if the character has been transferred completely.
8. To transfer the next character, go to Step 5.

Importance of the Tl flag

To understand the importance of the role of TI, look at the following sequence of steps that the 8051 goes through in transmitting a character via TxD.

1. The byte character to be transmitted is written into the SBUF register.
2. The start bit is transferred.
3. The 8-bit character is transferred one bit at a time.

4. The stop bit is transferred. It is during the transfer of the stop bit that the 8051 raises the TI flag (TI =1), indicating that the last character was transmitted and it is ready to transfer the next character.
5. By monitoring the TI flag, we make sure that we are not overloading the SBUF register. If we write another byte into the SBUF register before TI is raised, the untransmitted portion of the previous byte will be lost.

In other words, when the 8051 finishes transferring a byte, it raises the TI flag to indicate it is ready for the next character. 6. After SBUF is loaded with a new byte, the TI flag bit must be forced to 0 by the "CLR TI" instruction in order for this new byte to be transferred.

From the above discussion we conclude that by checking the TI flag bit, we know whether or not the 8051 is ready to transfer another byte. More importantly, it must be noted that the TI flag bit is raised by the 8051 itself when it finishes the transfer of data, whereas it must be cleared by the programmer with an instruction such as "CLR TI". It also must be noted that if we write a byte into SBUF before the TI flag bit is raised, we risk the loss of a portion of the byte being transferred. The TI flag bit can be checked by the instruction "JNB TI, . . ." .

Programming the 8051 to receive data serially

In the programming of the 8051 to receive character bytes serially, the following steps must be taken.

1. The TMOD register is loaded with the value 20H, indicating the use of Timer 1 in mode 2 (8-bit auto-reload) to set the baud rate.
2. TH1 is loaded with one of the values in Table 10-4 to set the baud rate (assum ing XTAL = 11.0592MHz).
3. The SCON register is loaded with the value 50H, indicating serial mode 1, where 8-bit data is framed with start and stop bits and receive enable is turned on.
4. TR1 is set to 1 to start Timer 1.RI is cleared with the "CLR RI" instruction.
5. The RI flag bit is monitored with the use of the instruction "JNB RI, xx" to see if an entire character has been received yet. When RI is raised, SBUF has the byte. Its contents are moved into a safe place.

6. To receive the next character, go to Step 5.

Importance of the Rl flag bit

In receiving bits via its RxD pin, the 8051 goes through the following steps.

1. It receives the start bit indicating that the next bit is the first bit of the charac ter byte it is about to receive.
2. The 8-bit character is received one bit at time. When the last bit is received, a byte is formed and placed in SBUF.
3. The stop bit is received. When receiving the stop bit the 8051 makes RI = 1, indicating that an entire character byte has been received and must be picked up before it gets overwritten by an incoming character.
4. By checking the RI flag bit when it is raised, we know that a character has been received and is sitting in the SBUF register. We copy the SBUF contents to a safe place in some other register or memory before it is lost.
5. After the SBUF contents are copied into a safe place, the RI flag bit must be forced to 0 by the "CLR RI" instruction in order to allow the next received character byte to be placed in SBUF. Failure to do this causes loss of the received character.

Data Transmission

Transmission of serial data begins at any time when data is written to SBUF. Pin P3.1 (Alternate function bit TXD) is used to transmit data to the serial data network. TI is set to 1 when data has been transmitted. This signifies that SBUF is empty so that another byte can be sent.

Data Reception

Reception of serial data begins if the receive enable bit is set to 1 for all modes. Pin P3.0 (Alternate function bit RXD) is used to receive data from the serial data network. Receive interrupt flag, RI, is set after the data has been received in all modes. The data gets stored in SBUF register from where it can be read.

Serial Data Transmission Modes:

Mode-0**:** In this mode, the serial port works like a shift register and the data transmission works synchronously v to the external circuitry for synchronization. The shift frequency or baud rate is always 1/12 of the oscillator freq

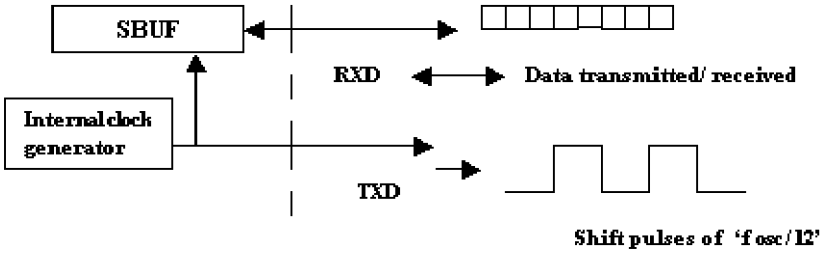
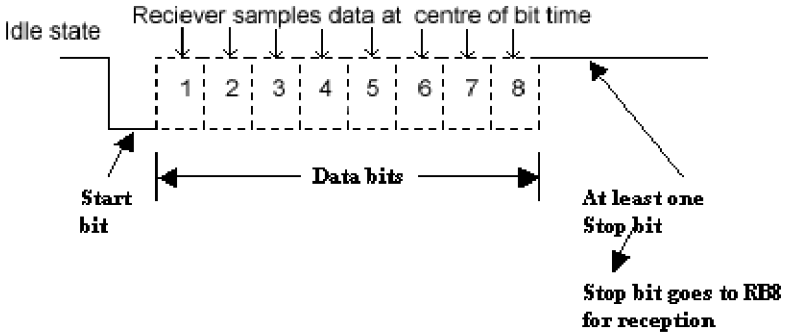


Fig : Data transmission/reception in Mode-0

In mode-1, the serial port functions as a standard Universal Asynchronous Receiver Transmitter (UART) mode received, the stop bit goes into RB8 in the special function register SCON. The baud rate is variable.

The following figure shows the way the bits are transmitted/ received.



Bit time= $1/f_{baud}$

In receiving mode, data bits are shifted into the receiver at the programmed baud rate.

The data word (8-bits) will be loaded to SBUF if the following conditions are true.

- RI must be zero. (i.e., the previously received byte has been cleared from SBUF)
- Mode bit SM2 = 0 or stop bit = 1.

After the data is received and the data byte has been loaded into SBUF, RI becomes one.

Mode-1 baud rate generation:

Timer-1 is used to generate baud rate for mode-1 serial communication by using overflow flag of the timer to determine the baud frequency. Timer-1 is used in timer mode-2 as an auto-reload 8-bit timer. The data rate is generated by timer-1 using the following formula.

Where,

SMOD is the

f_{osc} is the crystal oscillator frequency of the microcontroller

It can be noted that f_{osc}/ (12 X [256- (TH1)]) is the timer overflow frequency in timer mode-2, which is the auto-reload mode.

If timer-1 is not run in mode-2, then the baud rate is,

Timer-1 can be run using the internal clock, fosc/12 (timer mode) or from any external source via pin T1 (P3.5) (Counter mode).

Serial Data Mode-2 - Multiprocessor Mode :

In this mode 11 bits are transmitted through TXD or received through RXD. The various bits are as follows: a start bit (usually '0'), 8 data bits (LSB first), a programmable 9 th (TB8 or RB8)bit and a stop bit (usually '1').

While transmitting, the 9 th data bit (TB8 in SCON) can be assigned the value '0' or '1'. For example, if the information of parity is to be transmitted, the parity bit (P) in PSW could be moved into TB8. On reception of the data, the 9 th bit goes into RB8 in 'SCON', while the stop bit is ignored. The baud rate is programmable to either 1/32 or 1/64 of the oscillator frequency.

Mode-3 - Multi processor mode with variable baud rate :

In this mode 11 bits are transmitted through TXD or received through RXD. The various bits are: a start bit (usually '0'), 8 data bits (LSB first), a programmable 9 th bit and a stop bit (usually '1').

Mode-3 is same as mode-2, except the fact that the baud rate in mode-3 is variable (i.e., just as in mode-1).

$f_{baud} = (2^{SMOD}/32) * (f_{osc} / 12 (256-TH1))$.

This baudrate holds when Timer-1 is programmed in Mode-2.

Programming the 8051 to transfer data serially

Write a program for the 8051 to transfer letter "A" serially at 4800baud, continuously.

```
MOV TMOD,#20H ;timer 1, mode 2
MOV TH1,#-6 ;4800 baud rate
MOV SCON,#50H ;8-bit,1 stop,REN enabled
SETB TR1 ;start timer 1
AGAIN: MOV SBUF,#"A" ;letter "A" to be transferred
HERE: JNB TI,HERE ;wait for the last bit
CLR TI ;clear TI for next char
SJMP AGAIN ;keep sending A
```

Write a program to transfer the message "YES" serially at 9600 baud, 8-bit data, 1 stop bit. Do this continuously.

```
MOV TMOD,#20H ;timer 1, mode 2
MOV TH1,#-3 ;9600 baud
MOV SCON,#50H
SETB TR1
AGAIN: MOV A,#"Y" ;transfer "Y"
ACALL TRANS
MOV A,#"E" ;transfer "E"
ACALL TRANS
MOV A,#"S" ;transfer "S"
ACALL TRANS
SJMP AGAIN ;keep doing it
```

;serial data transfer subroutine

```
TRANS:                          MOV
HERE:           JNB             TI,HERE
CLR                 TI                      ;ge
RET
```

Baud Rates in the 8051

- Timer 1, mode 2 (8-bit, auto-reload)

- Define TH1 to set the baud rate.

XTAL = 11.0592 MHz

The system frequency = 11.0592 MHz / 12 = 921.6 kHz

Timer 1 has 921.6 kHz/ 32 = 28,800 Hz as source.

TH1=FDH means that UART sends a bit every 3 timer source.

Baud rate = 28,800/3= 9,600 Hz

Example

With XTAL = 11.0592 MHz, find the TH1 value needed to have the following baud rates. (a) 9600 (b) 2400 (c) 1200

Solution:

With XTAL = 11.0592 MHz, we have:

The frequency of system clock = 11.0592 MHz / 12 = 921.6 kHz

The frequency sent to timer 1 = 921.6 kHz/ 32 = 28,800 Hz

(a) 28,800 / 3 = 9600 where -3 = FD (hex) is loaded into TH1

(b) 28,800 / 12 = 2400 where -12 = F4 (hex) is loaded into TH1

(c) 28,800 / 24 = 1200 where -24 = E8 (hex) is loaded into TH1

Registers Used in Serial Trans1fer Circuit

SUBF (Serial data buffer)

SCON (Serial control register)

PCON (Power control register)

SBUF Register

Serial data register: **SBUF**

MOV SBUF,#'A' ;put char 'A' to transmit

MOV SBUF,A ;send data from A

MOV A, SUBF ;receive and copy to A

An 8-bit register

Set the usage mode for two timers

For a byte of data to be transferred via the TxD line, it must be placed in the SBUF.

SBUF holds the byte of data when it is received by the 8051's RxD line.

Write a C program for 8051 to transfer the letter "A" serially at 4800 baud continuously. Use 8-bit data and 1 stop bit.

Solution:

```
#include <reg51.h>
void main(void){
TMOD=0x20; //use Timer 1, mode 2
TH1=0xFA; //4800 baud rate
SCON=0x50;
TR1=1;
while (1) {
SBUF='A'; //place value in buffer
while (TI==0);
TI=0;
}
}
```

Write an 8051 C program to transfer the message "YES" serially at 9600 baud, 8-bit data, 1 stop bit. Do this continuously.

Solution:

```
#include <reg51.h>
void SerTx(unsigned char);
void main(void){
TMOD=0x20; //use Timer 1, mode 2
TH1=0xFD; //9600 baud rate
SCON=0x50;
TR1=1; //start timer
while (1) {
SerTx('Y');
SerTx('E');
SerTx('S');
}
```

```
}
void SerTx(unsigned char x){
SBUF=x; //place value in buffer
while (TI==0); //wait until transmitted
TI=0;
}
```

Program the 8051 in C to receive bytes of data serially and put them in P1. Set the baud rate at 4800, 8-bit data, and 1 stop bit.

Solution:

```
#include <reg51.h>
void main(void){
unsigned char mybyte;
TMOD=0x20; //use Timer 1, mode 2
TH1=0xFA; //4800 baud rate
SCON=0x50;
TR1=1; //start timer
while (1) { //repeat forever
while (RI==0); //wait to receive
mybyte=SBUF; //save value
P1=mybyte; //write value to port
RI=0;
}
}
```

Write an 8051 C Program to send the two messages "Normal Speed" and "High Speed" to the serial port. Assuming that SW is connected to pin P2.0, monitor its status and set the baud rate as follows:

SW = 0, 28,800 baud rate, SW = 1, 56K baud rate. Assume that XTAL = 11.0592 MHz for both cases.

Solution:

```
#include <reg51.h>
sbit MYSW=P2^0; //input switch
void main(void){
unsigned char z;
unsigned char Mess1[]="Normal Speed";
unsigned char Mess2[]="High Speed";
TMOD=0x20; //use Timer 1, mode 2
TH1=0xFF; //28800 for normal
SCON=0x50;
TR1=1; //start timer
if(MYSW==0) {
for (z=0;z<12;z++) {
SBUF=Mess1[z]; //place value in buffer
while(TI==0); //wait for transmit
TI=0;
}
}
else {
PCON=PCON|0x80; //for high speed of 56K
for (z=0;z<10;z++) {
SBUF=Mess2[z]; //place value in buffer
while(TI==0); //wait for transmit
TI=0;
}
}
}
```

8051 Interrupt

A computer has only two ways to determine the conditions that exist in internal and external circuits. One method uses software instructions that jump to subroutines on the status of flags and port pins. The second method responds to hardware signals, called interrupts that force the

program to call a subroutine. Most applications of microcontroller involve responding to events quickly enough to control the environment that generates the events termed real-time programming.

An interrupt is an external or internal event that interrupts the microcontroller to inform it that a device needs its service. A single microcontroller can serve several devices by two ways.

1) Interrupts

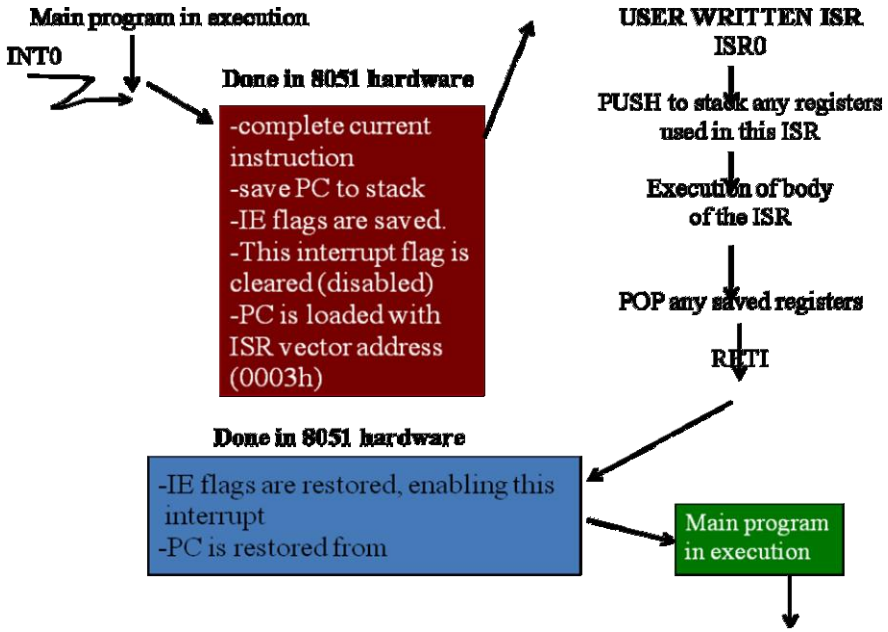
Whenever any device needs its service, the device notifies the microcontroller by sending it an interrupt signal. Upon receiving an interrupt signal, the microcontroller interrupts whatever it is doing and serves the device. The program which is associated with the interrupt is called the interrupt service routine(ISR) or interrupt handle

2) Polling

The microcontroller continuously monitors the status of a given device. When the conditions met, it performs the service. After that, it moves on to monitor the next device until everyone is serviced. Polling can monitor the status of several devices and serve each of them as certain conditions are met. The polling method is not efficient, since it wastes much of the microcontroller's time by polling devices that do not need service. ex. JNB TF,target.

The advantage of interrupts is that the microcontroller can serve many devices (not all at the same time). Each devices can get the attention of the microcontroller based on the assigned Priority. For the polling method, it is not possible to assign priority since it checks all devices in a round-robin fashion. The microcontroller can also ignore (mask) a device request for service. This is not possible for the polling method. For every interrupt, there must be an interrupt service routine (ISR), or interrupt handler. When an interrupt is invoked, the microcontroller runs the interrupt service routine. For every interrupt, there is a fixed location in memory that holds the address of its ISR. The group of memory locations set aside to hold the addresses of ISRs is called interrupt vector table.

Upon activation of an interrupt, the microcontroller goes through the following steps



1. It finishes the instruction it is executing and saves the address of the next instruction (PC) on the stack .
2. It also saves the current status of all the interrupts internally (i.e: not on the stack)
3. It jumps to a fixed location in memory, called the interrupt vector table, that holds the address of the ISR
4. The microcontroller gets the address of the ISR from the interrupt vector table and jumps to it. It starts to execute the interrupt service subroutine until it reaches the last instruction of the subroutine which is RETI (return from interrupt).
5. Upon executing the RETI instruction, the microcontroller returns to the place where it was interrupted. First, it gets the program counter (PC) address from the stack by popping the top two bytes of the stack into the PC. Then it starts to execute from that address.

Basically there are Six interrupts allocated as follows

1) Reset – power-up reset

2) Two interrupts are set aside for the timers: one for timer 0 and one for timer 1
3) Two interrupts are set aside for hardware. external interrupts P3.2 and P3.3 are for the external hardware interrupts INT0 (or EX1), and INT1 (or EX2).
4) Serial communication has a single interrupt that belongs to both receive and transfer.

Interrupt vector table

Interrupt	ROM Location (hex)	Pin
Reset	0000	9
External HW (INT0)	0003	P3.2 (12)
Timer 0 (TF0)	000B	
External HW (INT1)	0013	P3.3 (13)
Timer 1 (TF1)	001B	
Serial COM (RI and TI)	0023	

Upon reset, all interrupts are disabled (masked), meaning that none will be responded by the microcontroller if they are activated. The interrupts must be enabled by software in order for the microcontroller to respond to them.

What is difference between RET and RETI Instructions.

The instruction RET is a return from a function or a subroutine while RETI is return from an interrupt. The RETI instruction is executed at the end of interrupt subroutine. After the execution of the RETI instruction the PC address will be restored from the stack.

Comparison between Call instruction and the Interrupt action can be as:

Call Instruction	**Interrupt Action**
Completely under the control of programmer	Occurs at any time in the program
Placed in the program by the programmer	Enabled by the programmer
Execution is determined by where it is placed in the program	Calls the subroutine at any time and any place the program is executed

Table 1: Comparison of Call Instruction and Interrupt Action

Interrupt Enable (IE) SFR:

This is a bit addressable SFR with byte address A8H. The bits and addresses are shown in table 2. The bits are explained below.

IE.7	IE.6	IE.5	IE.4	IE.3	IE.2	IE.1	IE.0
EA	-	ET2	ES	ET1	EX1	ET0	EX0
AFH	AEH	ADH	ACH	ABH	AAH	A9H	A8H

Table 2: Interrupt Enable SFR

EA: This bit is a global interrupt enable/disable bit. When set to 1, it permits individual interrupts to be enable by their respective enable bits. When 0, it disables all interrupts.

IE.6: Not implemented

ET2: Reserved for future use.

ES: Enable serial port interrupt. Set to 1 by program to enable serial port interrupt. Cleared to 0 to disable serial port interrupt.

ET1: Enable (=1)/disable (=0) timer 1 interrupt

EX1: Enable (=1)/disable (=0) external interrupt 1

ET0: Enable (=1)/disable (=0) timer 0 interrupt

ET1: Enable (=1)/disable (=0) external interrupt 0

Interrupt Priority (IP):

This a bit addressable register, with byte address B8H. The addresses are shown in table 3. The priority of the interrupts is determined by the bits of IP SFR. The bits which are set to 1, have a high priority and bits with 0 have low priority. Interrupts with high priority can interrupt another interrupt with low priority. The lower priority interrupt is serviced after higher priority interrupt is finished.

IP.7	IP.6	IP.5	IP.4	IP.3	IP.2	IP.1	IP.0
-	-	PT2	PS	PT1	PX1	PT0	PX0
-	-	-	BC	BB	BA	B9	B8

Table 3: Interrupt Priority (IP)

IP.7 & IP.6: Not implemented

IP.5: Reserved for future use

PS: Serial port priority interrupt

PT1: Priority of timer 1 interrupt

PX1: Priority of external interrupt 1

PT0: Priority of timer 0 interrupt

PX0: Priority of external interrupt 0

When two or more interrupts have the same priority the microcontroller has its own ranking of providing service which is as below:

- External Interrupt 0 (P3.2)
- Timer 0 Overflow
- External Interrupt 0 (P3.3)
- Timer 1 Overflow
- Serial port

To enable an interrupt, we take the following steps:

1. Bit D7 of the IE register (EA) must be set to high to allow the rest of register to take effect

2. The value of EA. If EA = 1, interrupts are enabled and will be responded to if their corresponding bits in IE are high. If EA = 0, no interrupt will be responded to, even if the associated bit in the IE register is high.

Example

Show the instructions to (a) enable the serial interrupt, timer 0 interrupt, and external hardware interrupt 1 (EX1),and (b) disable (mask) the timer 0 interrupt, then (c) show how to disable all the interrupts with a single instruction.

Solution:

(a) MOV IE,#10010110B ;enable serial,;timer 0, EX1 Another way to perform the same manipulation is SETB IE.7 ;EA=1, global enable

SETB IE.4 ;enable serial interrupt

SETB IE.1 ;enable Timer 0 interrupt

SETB IE.2 ;enable EX1

(b) CLR IE.1 ;mask (disable) timer 0

;interrupt only

(c) CLR IE.7 ;disable all interrupts

The timer flag (TF) is raised when the timer rolls over .In polling TF, we have to wait until the TF is raised. The problem with this method is that the microcontroller is tied down while waiting for TF to be raised, and cannot do anything else.

Using interrupts solves this problem and, avoids tying down the controller. If the timer interrupt in the IE register is enabled, whenever the timer rolls over, TF is raised, and the microcontroller is interrupted in whatever it is doing, and jumps to the interrupt vector table to service the ISR. In this way, the microcontroller can do other until it is notified that the timer has rolled over.

Ex**ample**

Write a program that continuously get 8-bit data from P0 and sends it to P1 while simultaneously creating a square wave of 200 µs period on pin P2.1. Use timer 0 to create the square wave. Assume that XTAL = 11.0592 MHz.

Solution:

We will use timer 0 in mode 2 (auto reload). TH0 = 100/1.085 us = 92;--upon wake-up go to main, avoid using ;memory allocated to Interrupt Vector Table

ORG 0000H

LJMP MAIN ;by-pass interrupt vector table;--ISR for timer 0 to generate square wave

ORG 000BH ;Timer 0 interrupt vector table

```
CPL P2.1 ;toggle P2.1 pin
RETI ;
The main program for initialization
ORG 0030H ;after vector table space
MAIN: MOV TMOD,#02H ;Timer 0, mode 2
MOV P0,#0FFH ;make P0 an input port
MOV TH0,#-92 ;TH0=A4H for -92
MOV IE,#82H ;IE=10000010 (bin) enable;Timer 0
SETB TR0 ;Start Timer 0
BACK: MOV A,P0 ;get data from P0
MOV P1,A ;issue it to P1
SJMP BACK ;keep doing it loop;unless interrupted by TF0
END
```

Example

Rewrite above Example to create a square wave that has a high portion of 1085 us and a low portion of 15 us. Assume XTAL=11.0592MHz. Use timer 1.

Solution:

Since 1085 us is 1000 × 1.085 we need to use mode 1 of timer 1.

```
;--upon wake-up go to main, avoid using
;memory allocated to Interrupt Vector Table
ORG 0000H
LJMP MAIN ;by-pass int. vector table
;--ISR for timer 1 to generate square wave
ORG 001BH ;Timer 1 int. vector table
LJMP ISR_T1 ;jump to ISR
The main program for initialization
ORG 0030H                ;after vector table space
MAIN: MOV TMOD,#10H      ;Timer 1, mode 1
MOV P0,#0FFH             ;make P0 an input port
MOV TL1,#018H            ;      TL1=18          low byte of -1000
```

```
MOV TH1,#0FCH          ;TH1=FC high byte of -1000
MOV IE,#88H                              ;10001000 enable Timer 1 int
SETB TR1               ;Start Timer 1
BACK: MOV A,P0         ;get data from P0
MOV P1,A               ;issue it to P1
SJMP BACK              ;keep doing it
;Timer 1 ISR. Must be reloaded, not auto-reload
ISR_T1: CLR TR1            ;stop Timer 1
MOV R2,#4                  ; 2MC
CLR P2.1               ;P2.1=0, start of low portion
HERE: DJNZ R2,HERE         ;4x2 machine cycle 8MC
MOV TL1,#18H           ;load T1 low byte value 2MC
MOV TH1,#0FCH              ;load T1 high byte value 2MC
SETB TR1                   ;starts timer1 1MC
SETB P2.1                    ;P2.1=1,back to high 1MC
RETI                   ;return to main
END
```

Example

Write a program to generate a square wave if 50Hz frequency on pin P1.2. This is similar to Example 9-12 except that it uses an interrupt for timer 0. Assume that XTAL=11.0592 MHz

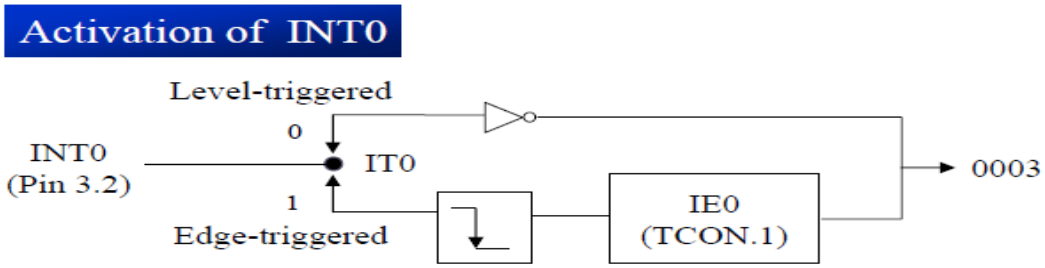
Solution:

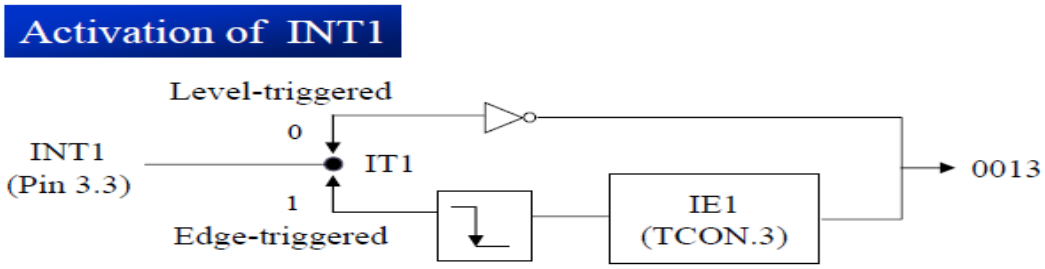
```
ORG 0
LJMP MAIN
ORG 000BH          ;ISR for Timer 0
CPL P1.2
MOV TL0,#00
MOV TH0,#0DCH
RETI
```

```
ORG 30H
;--------main program for initialization
MAIN:MOV TM0D,#00000001B ;Timer 0, Mode 1
MOV TL0,#00
MOV TH0,#0DCH
MOV IE,#82H ;enable Timer 0 interrupt
SETB TR0
HERE:SJMP HERE
END
```

The 8051 has two external hardware interrupts. Pin 12 (P3.2) and pin 13 (P3.3) of the 8051, designated as INT0 and INT1, are used as external hardware interrupts. The interrupt vector table locations 0003H and 0013H are set aside for INT0 and INT1. There are two activation levels for the external hardware interrupts.

1) Level trigged
2) Edge trigged





In the level-triggered mode, INT0 and INT1 pins are normally high.If a low-level signal is

applied to them, it triggers the interrupt. Then the microcontroller stops whatever it is doing and jumps to the interrupt vector table to service that interrupt. The low-level signal at the INT pin must be removed before the execution of the last instruction of the ISR, RETI; otherwise, another interrupt will be generated. This is called a level-triggered or level activated interrupt and is the default mode upon reset of the 8051.

TCON (Timer/Counter) Register (Bit-addressable)

D7							D0
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0

TF1	TCON.7	Timer 1 overflow flag. Set by hardware when timer/counter 1 overflows. Cleared by hardware as the processor vectors to the interrupt service routine
TR1	TCON.6	Timer 1 run control bit. Set/cleared by software to turn timer/counter 1 on/off
TF0	TCON.5	Timer 0 overflow flag. Set by hardware when timer/counter 0 overflows. Cleared by hardware as the processor vectors to the interrupt service routine
TR0	TCON.4	Timer 0 run control bit. Set/cleared by software to turn timer/counter 0 on/off

IE1	TCON.3	External interrupt 1 edge flag. Set by CPU when the external interrupt edge (H-to-L transition) is detected. Cleared by CPU when the interrupt is processed
IT1	TCON.2	Interrupt 1 type control bit. Set/cleared by software to specify falling edge/low-level triggered external interrupt
IE0	TCON.1	External interrupt 0 edge flag. Set by CPU when the external interrupt edge (H-to-L transition) is detected. Cleared by CPU when the interrupt is processed
IT0	TCON.0	Interrupt 0 type control bit. Set/cleared by software to specify falling edge/low-level triggered external interrupt