Serial Communication

Dinesh Sharma

Department Of Electrical Engineering
Indian Institute Of Technology, Bombay

February 16, 2021

Motivation for Serial Communication

Serial communications means sending data a single bit at a time. But why not send the data in parallel? Why bother with parallel to serial and serial to parallel conversions?

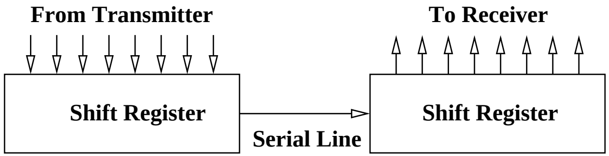
- There are several situations where serial communication is preferable.
- If the transmitter and the receiver are remote, it is expensive to lay a parallel cable with multiple wires in it.
- It may be difficult to ensure that all bits on a parallel bus have the same delay.

  If the difference in delay is comparable to the data rate, a ‘late’ bit from one transmission line might arrive at the same time as an ‘early’ bit from the other. The transmission will then get garbled.
- Also, if we are using an inherently single wire medium such as a phone line, serial communication would be required.

Serial Communication

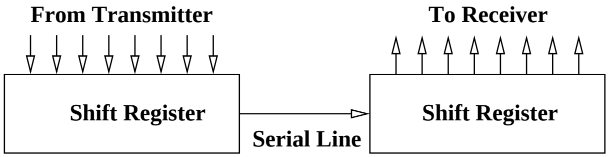
How do we send the data serially?

- At the transmitter, this involves loading data in parallel to a shift register, and shifting it out a bit at a time.



- At the receiving end, we have another shift register, which receives this data a single bit at a time and when all bits are received, the shift register can be read in parallel.
- The transmitter and the receiver must agree on the order in which the bits will be sent. It is common to send the least significant bit first.

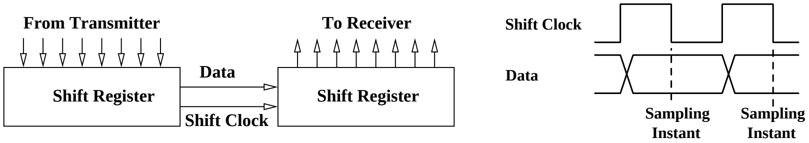
Synchronization



- Obviously, the rate at which the transmitter shifts the data out must be matched to the rate at which the receivers shifts the data in.
- Even if the clock frequencies at the transmitter and receiver are perfectly matched, the phase of the clock at the receiver should be carefully adjusted so that we sample the serial line when it is stable and not when the data on it is changing.

How can this be ensured?

Synchronization: Synchronous Serial I-O



- One way would be for the transmitter to send the data through one wire and the shift clock through another.
- The receiver then uses this same clock for sampling the data and shifting it in.
- If the transmitter places data on the positive clock edge and the receiver samples the data at the negative edge, the data will be stable at the sampling instant.

However, this doubles the wire cost.

Synchronization: Synchronous Serial I-O

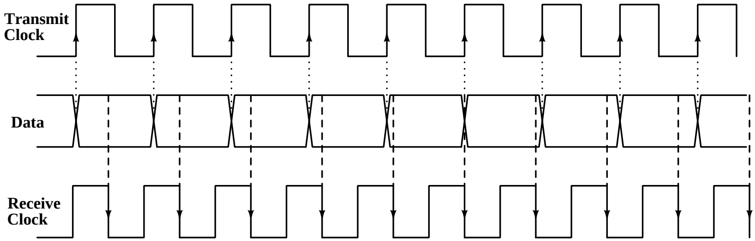
- We can use two wires instead of one and send data through one wire and the clock through the other.
- This way of sending serial data is called synchronous serial IO.
- It is useful for sending data over short distances.
  (Typically used for chip to chip communication on the same board).
- The advantage of this method is that it can achieve very high data rates.
- It is assumed that the delay over the clock line and the data line is reasonably well matched.
- 8051 provides support for this method through one of the modes defined for its serial IO (mode 0).

Asynchronous Serial I-O

- If the transmitter and the receiver are physically remote, they must use independent shift clock generators, set nominally to the same frequency.
- In this case, their clock frequencies will invariably have a small but non-zero mismatch.
- Unfortunately, this means that with time, the sampling time will drift further and further away from the ideal sampling instant.
- When the sampling time has shifted by more than a half period, the receiver might sample the wrong bit on the serial line, leading to errors.
- Obviously some means of synchronization must be provided to avoid this and to ensure reliable transfer of data.

Asynchronous Serial I-O

Phase drift due to slight difference between transmit and receive clocks



Asynchronous Serial I-O

- Since we don’t want to add any more wires, it is clear that the wire carrying data is the one which must carry the extra information required for synchronization.
- This involves sending extra bits on the wire along with the data. The data bits (called ‘payload’) along with the added bits constitute a **frame**.
- The clocks are synchronized at at the start of the frame. The frame is kept short enough, such that drift of the sampling instant is within acceptable limits.

Frame synchronisation in serial Asynchronous I-O

- Serial transmission often uses ‘non return to zero’ or NRZ encoding to minimize noise and power dissipation.
- If successive bits are identical, no edge may be generated by the data itself.
- To synchronize two clocks, we need to create an edge.
- To ensure that an edge occurs at the start of every frame, we add a bit before and a bit after the useful data.
- These bits are chosen to be different in value, thus ensuring that each frame starts with a value different from the one with which the previous one ended.
- These are called start and stop bits.
- It is common to use a ‘1’ for a stop bit and a ‘0’ for a start bit. Then each frame begins with a negative edge.

Timing models

At this time, we may have a general look at the timing models which are used in different systems for synchronization during data transmission and reception.

Fully synchronous: These are systems with a global clock such that the clock frequency is the same everywhere ($\Delta f = 0$) and the phase difference $\Delta\phi$ is zero, or at most, fixed and known.

Mesochronous: Here the clock source is the same everywhere, so ($\Delta f = 0$). However, wiring delays may introduce unknown delays. Thus $\Delta\phi$ is unknown, though bounded.

Design methods similar to synchronous design can be used for mesochronous systems by introducing compensating delays in data path.

Timing models

Pleisiochronous: Here the clock frequencies are nominally matched, but derived from independent oscillators. Thus ($\Delta f \neq 0$), but bounded. The phase will therefore drift with time. Hence, $\Delta\phi$ is unbounded.

The asynchronous serial data interface is an example of this timing model. The baud rate is agreed between communicating units, but these use independent oscillators. Thus ($\Delta f \neq 0$), though small. Depending on the difference in frequencies, $\Delta\phi$ will increase in magnitude continuously. Hence inherently, $\Delta\phi$ is unbounded.

Framing is used to make $\Delta\phi$ bounded. $\Delta\phi$ is made zero at the start of each frame. Since the frame size is restricted, the total phase drift over the frame is bounded.

Timing models

Rationally clocked: Here, the clocks for various sub-systems are derived by dividing a master clock by integral numbers. Thus ($\Delta f \neq 0$), and clock frequency of communicating systems are rationally related. $\Delta \phi$ is cyclic in nature, but known for different clock cycles.

Consider an application which receives a data packet, adds a header and outputs the enlarged data due to the header. To stream data without pausing, clock rates at input and output have to be different. Rational clocking is often used here.

Timing models

Fully Asynchronous: There is no clock in these systems.

- Data transfer takes place through request/acknowledge signals.
- A request from a listener invites the talker to place data on the data line(s).
- An acknowledge from the talker indicates that stable and valid data has been placed on data line(s) and will be kept there till the next request.
- The listener then uses the acknowledge signla to latch the data.
- This is called a pull system.

Timing models

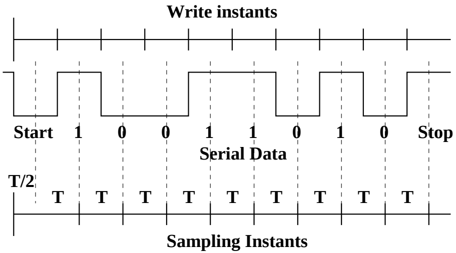
Fully Asynchronous: Data transfer through Request and Acknowledge.

- It is also possible for the talker to initiate a data transfer through a request. Now Request signals that valid data is available on data line(s).
- The listener uses the Req signal to latch data and then asserts acknowledge to indicate that the talker can now place fresh data on the data line(s).
- This is called a push system.

In either system (push or pull), request and acknowledge events need not occur at fixed time intervals.

Bit sampling in serial Asynchronous I-O

Each new bit is placed on the serial line by the transmitter after time T. (This is called bit time and the corresponding frequency the baud rate).



- Notice that the first sampling instant is offset from the start edge by half a bit time.
- This is to ensure that a bit is sampled in the middle of the bit interval T, where it is stable.

After sampling the start bit, each bit is sampled a time T after the previous bit.

Use of Parity bit for error checking

- Often a ‘Parity bit’ is also added to the data.
- The transmitter and the receiver agree to use either *even* or *odd* parity.
- The parity bit is chosen by the transmitter to be a ‘0’ or a ‘1’ so that the total number of ‘1’s in a frame is always even (or always odd).
- The receiver counts the number of ‘1’s and confirms that the count is ‘even’ (or odd) as previously agreed with the transmitter.
- If it is not, it can signal an error and the software can take appropriate action (such as a request for re-transmission).
- The use of parity bit is optional.

Bit sampling in serial Asynchronous I-O

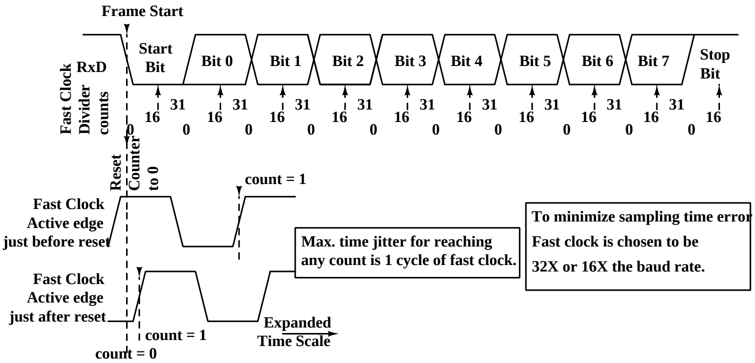
- The receiver knows that a frame will always begin with an edge.
- It uses a locally generated clock which is a multiple of the baud rate, (say 32X) for synchronization and timing.
- This fast clock is then divided by a counter to generate sampling events which are separated by a bit time.
- The fast clock is not aware when the negative edge will occur on the receive data line and so, the 'start of frame' edge on the serial data may not coincide with an edge on the locally generated clock.
- We take the next edge of the local clock (immediately following the expected edge at the start of the frame in the data stream) as the effective start of frame.
- This introduces a small error in timing – between 0 to 1 cycle of the local clock.
- Obviously, the local clock should not operate at the data rate. If the local clock was chosen to run at the baud rate itself, a possible timing error of 1 period of local clock will shift the data by 1 bit!

Bit sampling in serial Asynchronous I-O

To keep the synchronization error to a small value, the locally generated clock operates at a frequency much higher than the bit rate.

- This high frequency clock is divided down to get the actual shift clock.
- For example, the locally generated clock might operate at 32 times the shift clock.
- A 5 bit counter will be required to divide the local clock down to sampling clock rate.
- The worst case sampling error in timing will now be one cycle of the locally generated clock, which is 1/32 of the bit time.
- This is an acceptable error in sampling time. Instead of sampling the input data stream exactly in the middle of the bit time, we may sample it at a time which is shifted by T/32 from the optimum instant.

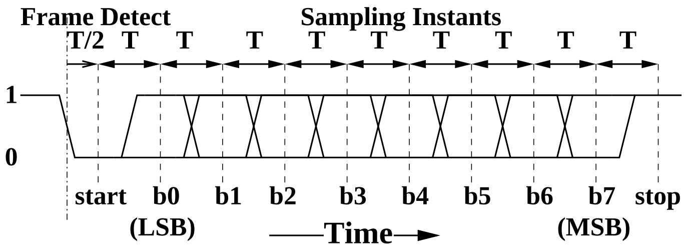
Bit sampling in serial Asynchronous I-O



Max. time jitter for reaching any count is 1 cycle of fast clock.

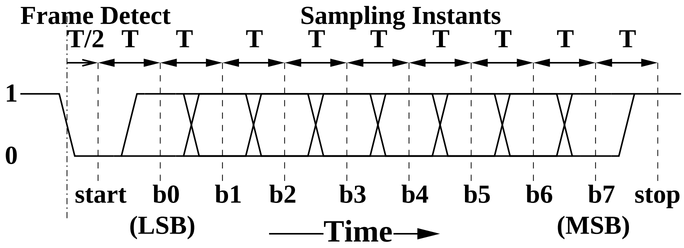
To minimize sampling time error Fast clock is chosen to be 32X or 16X the baud rate.

Bit sampling in serial Asynchronous I-O



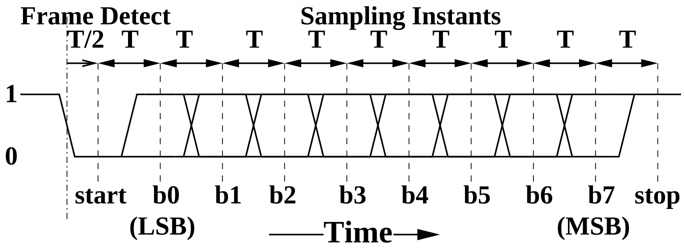
- Synchronization occurs at the beginning of each frame. Therefore timing errors do not accumulate from frame to frame.
- The counter used to divide the local clock is reset when the 'start of frame' edge is detected.
- In the example that we have chosen (internal clock = 32X bit rate), we reset the 5 bit counter to all zeros when we detect the frame edge.

Bit sampling in serial Asynchronous I-O



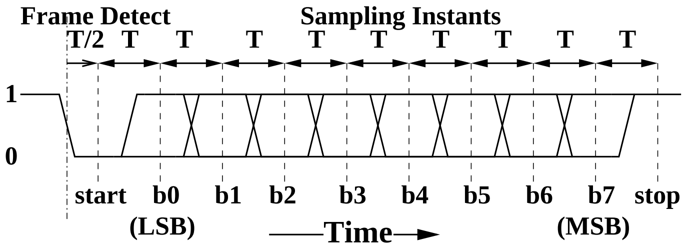
- The internal clock divider is reset to 00000 at Frame Detect.
- Now, after 16 cycles of the internal clock, (corresponding to time T/2 where T is the interval between bits) the start bit is sampled.
- If it is not ‘0’, a ‘framing error has occurred and should be handled in software – say by a request to re-transmit.

Bit sampling in serial Asynchronous I-O



- If the start bit has been correctly sampled, data bits will be sampled every 32 counts, till all data bits have been sampled.
- This might be 8 bits if no parity bit is appended and 9 bits if parity bit is used.
- All bits are sampled when the count is ‘16’. So the MSB of the counter can be used to latch the data.

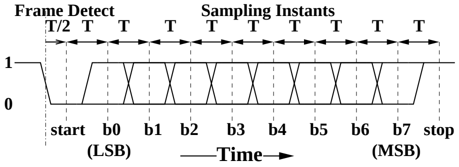
Bit sampling in serial Asynchronous I-O



- After all the data bits (and optionally, the parity bit) have been received, the next bit must be the stop bit (normally a ‘1’).
- If it is not the correct value, a frame error has occurred and must be handled in software appropriately.
- If the stop bit is correctly received, we accept the data and wait for the next ‘start of frame’.

Bit sampling in serial Asynchronous I-O

Notice that this scheme will work equally well, if additional delay is inserted between the end of one frame and the beginning of the next.



After the last bit has been sampled, no further timing needs to be performed. The serial line idles at the stop bit (normally ‘1’).

- The counter is reset to zero *whenever* the next frame edge is detected.
- Therefore it does not matter how much time elapses between the end of a frame and the beginning of the next.

Serial IO protocol

Before serial communications can be carried out, the transmitter and the receiver must agree on a number of parameters.

- Whether synchronous or asynchronous serial communication will be used.
- At what rate will the bits be sent.
- How many bits will be sent at a time (frame size).
- Whether parity will be used and if so, which kind of parity (even or odd) will be used.
- Whether simultaneous transmission and reception (duplex) will be possible or only one of these will be carried out at a time (simplex).

These parameters constitute the serial protocol.

Universal Asynchronous Receiver Transmitter (UART)

- A circuit which implements asynchronous serial communication protocol is known as a Universal Asynchronous Receiver Transmitter (UART).
- If it can also carry out synchronous serial communication, it is called a USART (Universal Synchronous Asynchronous Receiver Transmitter).
- Microprocessors often use dedicated chips like the Intel 8251 USART for implementing serial communication.
- The 8051 has a built in USART.

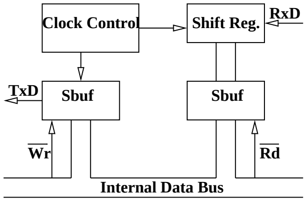
8051 Serial Port

- The 8051 architecture is designed to make sending and receiving serial data quite simple.
- It is capable of duplex communication, that is serial transmission and reception can be carried out simultaneously.
- Pins of port 3 perform a dual function. P3.0 can also serve as the serial data input (RxD) and P3.1 can serve as the serial data output (TxD) when serial IO has been enabled.

Additional functions of Port 3 lines

Port Line	P3.7	P3.6	P3.5	P3.4	P3.3	P3.2	P3.1	P3.0
Function	$\overline{\text{RD}}$	$\overline{\text{WR}}$	T1 in	T0 in	$\overline{\text{INT1}}$	$\overline{\text{INT0}}$	TxD	RxD

8051 Serial Port



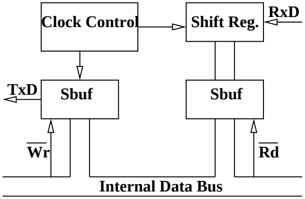
The 8051 contains two registers, one for receiving and the other for sending serial data. Both of these respond to the same address (99H) in the special function register area.

(The assembler aliases the address 99H to the name Sbuf).

- The receive register places data on the internal bus when a read is performed from Sbuf (99H).
- When a write is performed to the same address (99H), it is actually latched into the transmit register.
- The transmit register itself acts as the shift register for outputting serial data through the TxD pin (P3.1).
- When the data has been shifted out, a flag called TI is set, inviting the program to write fresh data to Sbuf.

8051 Serial Port

The serial data input pin RxD (P3.0) is connected to an independent shift register, whose output is transferred to the receive buffer after a complete frame has been shifted in.



When a complete frame has been shifted in, the data is transferred to the receive sbuf register and a flag (RI) is set to inform the processor that serial data is available for reading at Sbuf.

- Flags RI and TI are parts of a bit addressable register SCON, which resides at address 98H in the special function register area.
- SCON is used for configuring serial communication.

Configuring the Serial Port

- Configuration of the serial port involves choosing the mode of communication, deciding rate of sending data and managing the parity bit etc.
- Much of this work is done by the special function register SCON at byte address 98H in the SFR area. SCON is bit addressable.

SCON register at BYTE address 98H								
Bit No.	7	6	5	4	3	2	1	0
Bit Addr	9F	9E	9D	9C	9B	9A	99	98
Bit Name	SM0	SM1	SM2	REN	TB8	RB8	TI	RI

Baud rate is set using internal timers.

Configuring the Serial Port

SPECIAL FUNCTION REGISTERS

ADDR	0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F	
F8									FF
F0	B								F7
E8									EF
E0	ACC								E7
D8									DF
D0	PSW								D7
C8	T2CON	T2MOD	RCAP2L	RCAP2H	TL2	TH2			CF
C0									C7
B8	IP								BF
B0	P3								B7
A8	IE								AF
A0	P2								A7
98	SCON	SBUF							9F
90	P1								97
88	TCON	TMOD	TL0	TL1	TH0	TH1			8F
80	P0	SP	DPL	DPH				PCON	87
	0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F	

SCON resides at byte address 98H in the SFR area. It is bit addressable. Two physically distinct registers responding to the byte address 99H are used as transmit and receive buffers during serial IO.
Additionally, the most significant bit of the PCON register at address 87H is used for doubling the baud rate when set.

Configuring the Serial Port

SCON register at BYTE address 98H								
Bit No.	7	6	5	4	3	2	1	0
Bit Addr	9F	9E	9D	9C	9B	9A	99	98
Bit Name	SM0	SM1	SM2	REN	TB8	RB8	TI	RI

- The upper nibble of SCON configures the operation of the serial IO, while the lower nibble is used for auxiliary data arising *per frame* during communication.
- SM0 and SM1 decide the mode of communication. (Note the somewhat non-standard convention of calling the more significant bit SM0 and the less significant bit SM1).
  SM2 is used for multi-cast transmission.

Configuring the Serial Port

SCON register at BYTE address 98H								
Bit No.	7	6	5	4	3	2	1	0
Bit Addr	9F	9E	9D	9C	9B	9A	99	98
Bit Name	SM0	SM1	SM2	REN	TB8	RB8	TI	RI

- SM0 determines if 8bits of data will be sent per frame (SM0=0) or 9 (SM0=1).
- 9 bits are used when we want to send a parity bit in addition to 8bit data.
- 9 bit communication is also used in a special way with SM2 during multi-cast communication.

Configuring the Serial Port

SCON register at BYTE address 98H								
Bit No.	7	6	5	4	3	2	1	0
Bit Addr	9F	9E	9D	9C	9B	9A	99	98
Bit Name	SM0	SM1	SM2	REN	TB8	RB8	TI	RI

SM0 determines if 8bits of data will be sent per frame (SM0=0) or 9 (SM0=1).

SM1 determines if fixed or variable baud rate will be used.

SM2 is used for multi-cast transmission and will be explained later. In normal use, it should be kept cleared.

REN must be set to enable reception of data on the RxD line. (What would happen if serial reception is permanently enabled?)

Configuring the Serial Port

SCON register at BYTE address 98H								
Bit No.	7	6	5	4	3	2	1	0
Bit Addr	9F	9E	9D	9C	9B	9A	99	98
Bit Name	SM0	SM1	SM2	REN	TB8	RB8	TI	RI

- If we use a parity bit, we need to send and receive 9 bits of data along with the start and stop bits in every frame.
- Sbuf can hold only 8 bits each in the send and receive buffers. Where can the 9th bit be accommodated?
- Bits TB8 and RB8 in SCON are used to accommodate the 9th bit during transmission and reception respectively.

Configuring the Serial Port

SCON register at BYTE address 98H								
Bit No.	7	6	5	4	3	2	1	0
Bit Addr	9F	9E	9D	9C	9B	9A	99	98
Bit Name	SM0	SM1	SM2	REN	TB8	RB8	TI	RI

- When all bits of a frame have been received serially, the hardware in 8051 sets the flag RI. This is an invitation to the processor to read Sbuf (and if required, RB8).
- Similarly, TI is set when the frame corresponding to the data written to Sbuf (and if required, TB8) have been shifted out of the TxD line. This is an invitation to write fresh data to Sbuf if more data needs to be sent out on the serial line.
- Notice that TI and RI are not automatically cleared and must be cleared by software which reads/writes Sbuf.

Mode 0: Synchronous Serial Communication

SM0 and SM1 determine the mode of serial IO. Synchronous serial communication can be carried out using mode 0.

- Mode 0 corresponds to SM0=0 and SM1=0.
- This is a Shift Register mode: Data is transmitted *as well as* received through the RxD pin and the shift clock is supplied through the TxD pin.
- Since shift clock is supplied separately, there is no need for a start and a stop bit.
- 8 bits of data are transmitted or received (since SM0 = 0).
- Data is communicated at a fixed rate (since SM1 = 0) given by the crystal oscillator frequency divided by 12.

Mode 1:Asynchronous Serial Communication

- This mode corresponds to SM0 = 0 and SM1 = 1, and is a duplex UART mode.
- Data is sent through the TxD pin and received through the RxD pin. A 10 bit frame containing 8 bit data (since SM0=0), a start bit (=0) and a stop bit (=1) are used.
- SM1 = 1 implies variable data rate. The rate of sending and receiving data is set by the overflow rate of timer T1 (or T2 in case of 8052).
  Details of how T1 or T2 are used will be discussed later.
- The data rate can be doubled by setting the most significant bit of the special function register PCON at address 87H. This bit is called SMOD.

Mode 1:Asynchronous Serial Communication

- In this mode, we use 8 bit data (no parity) and set the baud rate using overflow rates on T1 (or in case of 8052, T2).
- Which timer will be used is decided by a bit in T2CON register. In 8051 there is no T2, so we always use T1.
- The data rate can be doubled by setting the most significant bit of the special function register PCON at address 87H. This bit is called SMOD. If this bit is set, the sampling clock runs at 16 times and not at 32 times the baud rate.
- The stop bit sampled on the serial input line is copied to bit RB8 in SCON in this mode.

Mode 2: Asynchronous Serial Communication with Parity

- Mode 2 is set by making SM0 = 1 (11 bit frame containing 8 bits of data, Parity, start and stop bit) and SMO = 0 (fixed baud rate).
- This is an asynchronous mode using a start and stop bit.
- Serial data is transmitted using the TxD pin and received using the RxD pin.
- The baud rate is fixed at $f_{osc}/64$. The baud rate can be doubled to $f_{osc}/32$ by setting the SMOD bit.

Mode 3: Asynchronous with Parity and variable baud rate

- This mode is set by making SM0 = 1 (11 bit frame containing 8 bits of data, Parity, start and stop bit) and SMO = 1 (variable baud rate).
- This is an asynchronous mode using Parity, a start and a stop bit (identical to mode 2) except that the baud rate is determined from the overflow rate of timer T1 or T2.
- Serial data is transmitted using the TxD pin and received using the RxD pin.
- The baud rate in modes 1,2 and 3 can be doubled by setting the bit SMOD (PCON.7).
- Notice that PCON is not bit addressable and masking must be used to set or clear this bit without disturbing the rest of PCON, which controls the sleep and idle modes to reduce power dissipation during times of low activity.

Serial communications using Interrupts

Interrupt operation for Serial IO is enabled using bits in the IE register, which is at byte address A8H. This register is bit addressable.

SFR IE at byte address A8H								
Bit No.	7	6	5	4	3	2	1	0
Function	IE	U	U	SI	TF1	Ex1	TF0	Ex0
Bit Addr	AF	AE	AD	AC	AB	AA	A9	A8
Bit Name	EA	-	-	ES	ET1	EX1	ET0	EX0

(The assembler gives special mnemonics to each bit address.)

The most significant bit of the register is a global interrupt enable flag. This bit must be set in order to enable any interrupt.
Bit 4 (ES) specifically enables interrupt from the serial interface.

To do serial I-O under interrupt control, we need to enable serial interrupts by setting both bits 4 (ES) and 7 (EA) of the interrupt enable special function register IE

Serial communications using Interrupts

- Once enabled, the serial interface causes interrupts due to a receive event (RI Flag set) or due to a transmit event (TI Flag set). RI and TI are in the SCON register.
- The receive event occurs when the input buffer of the serial line (Sbuf in) is full and a byte needs to be read from it.
- The transmit event indicates that a byte has been sent and a new byte can be written to output buffer for the serial line (Sbuf out).
- The interrupt service routine is supposed to examine both the flags and take appropriate action accordingly.

Serial communications using Interrupts

- When a serial interrupt occurs, control is transferred to address 0023H of the program memory, which is the vector address for serial interrupts.
- The interrupt handler routine is supposed to be placed here. Since only 8 bytes are available in this region for each handler, it is conventional to place the interrupt service routine elsewhere and to place an ljump instruction at address 0023H to the handler.
- Either or both of the RI and TI flags might be set, requiring a read from or write to the serial buffer Sbuf (or both).
- The interrupt service routine should check which of these events caused the interrupt.
- The RI and TI flags are *not* automatically cleared when an interrupt is serviced. Therefore, the interrupt service routine must clear them before returning.

Serial communications using Interrupts

Here is an example handler for serial interrupts:

```
SerialISR:                          ; put ljmp SerialISR at 0023H
          PUSH    PSW               ; Save flags and context
          JNB     RI,output         ; If RI not set, check for TI
          MOV     InChar, Sbuf      ; If set, Save this character
          CLR     RI                ; clear receive interrupt flag
output:                             ; Check if output is required
          JNB     TI, done          ; If no transmit flag, leave
          MOV     sbuf, outchar     ; Else send the character
          CLR     TI                ; Clear Transmit interrupt flag
done:     POP     PSW               ; Restore flags etc.
          RETI                      ; and return
```

Multi-cast Serial IO

- The earlier discussion assumed there is a single transmitter and a single receiver.
- However, at times, we need multi-cast serial IO, where there is a single transmitter and multiple receivers.
- This can be managed using the SM2 flag and using the 9th bit for this purpose, rather than for Parity.
- When SM2 is set, the receive interrupt flag is *not* set if the received character does not have the 9th bit set.
- This can be used to send data only to selected listeners.
- To do this, we need to send ‘commands’ to activate those listeners which should receive data and to de-activate those who should not.

Multi-cast Serial IO

- When SM2 is set, the receive interrupt flag is *not* set if the received character does not have the 9th bit set.
- The way this characteristic is used is as follows:

Several ‘slaves’ are connected in parallel to a single ‘master’. Each slave is given a unique address.
We use mode 2 or mode 3, which uses 11 bit frames containing 9 bit data, the start bit and the stop bit.

However, the 9th data bit is not used for parity here. It is used in a special way to enable reception of data only by selected receivers.

Multi-cast Serial IO

- Initially, all ‘slaves’ have their SM2 bits set.
- ‘Commands’ are sent by the master with the 9th bit set. (The master can easily do this by loading a ‘1’ in its ‘TB8’ bit in SCON).
- Because of this, when commands are sent, all ‘slaves’ receive an interrupt and read the command.

To illustrate the procedure with an example, we define a specific command format. (This is just for this example).
Assume that the upper nibble of the command encodes what command it is and the lower nibble contains the address of slave who should execute it.
(Thus, up to 16 slaves can be managed in this example).

Multi-cast Serial IO

Let us say that a 0001 in the upper nibble means “become an active listener” while 0010 means “stop listening”.

- In order to make a particular slave (with address = 7) an active listener, the master can send the command 0001 0111 with TB8 set.
- All slaves receive this command because the 9th bit was set.
- slave 7 finds that its address matches the lower nibble.
- Since the command asks it to become an active listener, It *clears* its SM2 bit. (This is the action associated with this command).
- The other slaves, for whom the lower nibble of the command did not match their address, do not execute the command and thus, continue to keep the SM2 bit set.
- Data is now sent by the master with the 9th bit (TB8) cleared.

Multi-cast Serial IO

- The selected slave has its SM2 bit cleared. Therefore, it will receive an interrupt whether the 9th data bit is set or not.
- On the other hand, no interrupt is generated in other slaves (because SM2 bit is set and the RB8 bit is not set).
- Therefore they automatically ignore the data.
- The selected listener can be de-selected by sending the command 0010 0111 with TB8 set.
- All slaves receive the command (because RB8 is set).

Multi-cast Serial IO

- Slave 7 knows that this is a command and not data (because RB8 is set).
- Since the lower nibble matches its address, it has to execute this command.
- The upper nibble is 0010, which means “de-select”. So slave 7 sets its SM2 bit again and now will be interrupted only by commands (which have the 9th bit set) and not by data (where the 9th bit is cleared).
- Thus, the master can select which of the listeners will receive its data and which will not.
- Of course this is just an example and different command conventions and actual commands may be defined.

Baud rate generation

- In modes 1 and 3, the baud rate is generated by the overflow rate of timer T1 (for 8051).
- While T1 can be used in any mode, it is convenient to use it in the auto reload mode (timer mode 2).
- This timer mode uses TH1 to store the auto-reload value and uses TL1 as an 8 bit counter.
- Once configured, T1 runs continuously and needs no further software effort.
- Its input is the oscillator frequency divided by 12.
- (Interrupts are not required and in fact must not be enabled for T1).

Baud rate generation

- In mode 2, T1 is an 8 bit counter. It will overflow when TL1 reaches 256. At this time, TL1 will be reloaded automatically with the value stored in TH1.
- Suppose the value n is loaded in TH1 and f_c = crystal frequency. TL1 loads the value n and counts up. It reaches 256 after 256-n cycles. Then it reloads itself with n and starts counting up again.
- So, TL1 will overflow and reload the value n every (256-n) inputs.
- The frequency of the input to the counter is $f_c/12$.
- Therefore the frequency of the output is $f_c/12(256-n)$
  This constitutes the 32X clock for synchronization.
- Therefore the baud rate is: $f_c/(12 \times 32 \times (256-n))$
  Thus, baud rate = $\frac{f_c}{384(256-n)}$

Baud rate generation

In practice, we are given the baud rate and must determine n.

$$\text{Since baud rate} = \frac{\mathrm{f}_c}{384(256-n)}, \quad n = 256 - \frac{\mathrm{f}_c}{384 \times \text{baud rate}}$$

If SMOD bit is set, the internal clock is just 16X the baud rate. In that case,

$$\text{baud rate} = \frac{\mathrm{f}_c}{192(256-n)} \qquad \text{so } n = 256 - \frac{\mathrm{f}_c}{192 \times \text{baud rate}}$$

Baud rate generation example

Let us illustrate it with an example. A crystal frequency of 11.059 MHz is often used because it gives convenient integral values for n for most standard baud rates. Consider a baud rate of 9600.

$$n = 256 - \frac{11059000}{384 \times 9600} = 256 - 2.9999 \simeq 256 - 3 = 253$$

If we back calculate the baud rate with this value of n,

$$\text{baud rate} = \frac{11059000}{384 \times 3} = 9599.2$$

which is close enough. However, if we want 19200 baud,

$$n = 256 - \frac{11059000}{384 \times 19200} = 256 - 1.49997 \simeq 256 - 1.5$$

In this case we do not get an integral value for n.

Baud rate generation example

To generate a clock for 19200 baud,

$$n = 256 - \frac{11059000}{384 \times 19200} = 256 - 1.49997 \simeq 256 - 1.5$$

In this case we do not get an integral value for n. If we use 256-1 = 255, we get

$$\text{baud rate} = \frac{11059000}{384 \times 1} = 28799.48$$

and if we use 256-2 = 254, we get

$$\text{baud rate} = \frac{11059000}{384 \times 2} = 14399.74$$

neither of which is anywhere close to what we want.

Baud rate generation example

We don't get a convenient divisor value to load into T1H if we want a baud rate of 19200 baud. However, in this case, we can set the SMOD bit (PCON.7). Then,

$$n = 256 - \frac{11059000}{192 \times 19200} = 256 - 2.9999 \simeq 256 - 3 = 253$$

This gives a baud rate of

$$\text{baud rate} = \frac{11059000}{192 \times 3} = 19199.62$$

Which is acceptable.
This is about the highest standard baud rate possible with a standard 8051 which is rated to run at clock frequencies up to about 12 MHz.

Baud rate generation example

The lowest baud rate we can get using T1 in 8bit auto reload mode is about 110. A count of 0 will give

$$\text{baud rate} = \frac{11059000}{384 \times 256} = 112.4979$$

which is marginally acceptable for 110 baud communication.

(This is because at such low baud rates, we can afford to be a fair bit away from the middle of the bit time and still not run the risk of sampling the wrong bit).
However, if we need even lower baud rates this timer mode will not do.

In these cases, we can either operate the micro-controller at a lower crystal frequency, or use a 16 bit timer mode and reload the timer in software with a calculated value every time T1 overflows.

Baud rate generation

- To get better resolution and range of baud rates which can be generated, we can use a 16 bit timer mode and reload the timer in software with a calculated value every time T1 overflows.
- Calculation of the count now will require us to include the number of clock cycles needed to respond to the interrupt and then to reload the counter.
- Fortunately, the use of baud rates below 110 is extremely rare.
- Of course, these problems have been solved by the additional timer in 8052 and later chips. Using this timer, we can have 16 bit resolution, as well as auto-reload without losing resolution.
- We must use a higher crystal clock frequency to use the better resolution.Modern chips of the 8051 family do allow that – for example the processor in our lab kit operates at 24 MHz.

Putting it all together

Here is the list of things we need to do in order to communicate serially.

For configuring asynchronous serial communication at a given baud rate to be set by T1:

- Write the interrupt service routine for serial interrupts. (An example has been provided earlier).
- Place an ljump to it at the vector address 0023H.

Putting it all together

- Set SCON bits to desired values.

SCON register at BYTE address 98H								
Bit No.	7	6	5	4	3	2	1	0
Bit Addr	9F	9E	9D	9C	9B	9A	99	98
Bit Name	SM0	SM1	SM2	REN	TB8	RB8	TI	RI

- For communication without the parity bit, clear SM0.
  Set SM0 for communication with parity.
- Since baud rate is to be decided by T1, set SM1.
- Clear SM2, as we are not using multi-cast transmission.
- We should (eventually) set REN to enable data reception, but it is better to do this as the last step to avoid receiving invalid data before configuration is complete.

Putting it all together

Load TH1 count to provide the desired baud rate.

- If we get a good integer value for the divisor n for the given baud rate and crystal frequency, clear SMOD (PCON.7) by ANDing it with #7FH. (This is because PCON is not bit addressable).
  Calculate the value to be loaded in TH1 as
  $$n = 256 - \frac{\mathrm{f}_c}{384 \times \text{baud rate}}$$
- If we need to double the baud rate to get an integer value for n, set SMOD (PCON.7) by ORing it with #80H. (This is because PCON is not bit addressable).
  Calculate the value to be loaded in TH1 as
  $$n = 256 - \frac{\mathrm{f}_c}{192 \times \text{baud rate}}$$
- To Load TH1 with n, include:
  MOV TH1, #n   in the initialization code.

Putting it all together

Configure T1 in TMOD

TMOD register at BYTE address 89H								
Bit No.	7	6	5	4	3	2	1	0
Timer:	T1				T0			
Bit Name	G1	C/T1	T1M1	T1M0	G0	C/T0	T0M1	T0M0

We need to disable hardware gating for T1 (G1 = 0), use it as a timer (C/T1 = 0), and put it in 8bit auto reload mode (T1M1 = 1, T1M0 = 0). Thus, we want to load 0010 in the upper nibble. Since TMOD is not bit addressable and we don't want to disturb T0 settings, we do it by ORing TMOD with #20H (to set bit 5) and ANDing it with #2FH (to clear bits 7,6 and 4).

Putting it all together

Configure T1 in TCON

TCON register at BYTE address 88H								
Bit No.	7	6	5	4	3	2	1	0
Bit Name	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
Bit Addr	8F	8E	8D	8C	8B	8A	89	88

We set TR1 in TCON to allow T1 to run. Since TR1 is bit addressable, this is easily done with SetB TR1.

Putting it all together

We want to disable interrupts from T1 (it is in auto re-load mode). and enable serial interrupts.

This is done by manipulating the special function register IE.

Interrupt Enable Register IE at A8H								
Bit No.	7	6	5	4	3	2	1	0
Bit Addr	AF	AE	AD	AC	AB	AA	A9	A8
Interrupt on	IE	U	U	SI	TF1	Ex1	TF0	Ex0
Bit Name	EA	-	-	ES	ET1	EX1	ET0	EX0

We need to set EA (IE.7), ES (IE.4) and clear ET1. Since IE is bit addressable, this is set easily by by sETB and CLR instructions.

Now we are ready: clear RI and TI to remove any stale flags and set REN in SCON to enable serial I-O.

Using the serial port

Once the configuration is done, it is easy to use the serial port.

On a receive interrupt, we just read SBUF and clear RI.

On a transmit interrupt, we write the byte to SBUF.

If parity is being used, we can just add 0 to the byte in A and copy the parity flag (PSW.0) to TB8 in SCON before writing to SBUF.