Serial Communications
Serial Data (8 or 9 bit) is asynchronous (not synchronized to system clock):

- bytes can arrive at any time
- the CPU has two status bits in PSW to inform the user when either of these conditions occur:
  - RI – (Receive) - set if a new byte has arrived
  - TI – (Transmit) - set if another byte may be send

Handling Unpredictable Events

- Polling – the software periodically checks the status bits to see if:
  - a new byte has arrived – RI will be set
  - a new byte may be sent – TI will be set

- Interrupts – the CPU calls a subroutine automatically if either RI or TI are set:
  - one subroutine for two conditions:
  - the subroutine must the check to see which condition (RI/TI/both) caused the interrupt

Sending Data Serially & Asynchronously
To send parallel data serially (one bit at a time), a shift register can be used:

If only two wires are used, the clock can’t be sent BUT if clock period is known, then using START and Stop bits and synchronising on falling edge of START bit will suffice.

8051 Serial I/O
The 8051 has:
- sending and receiving shift registers
- logic for generating the clocks
- logic to detect if the STOP bit is present
- status bits to indicate when data has been received,
- status but showing sending data has finished being shifted out

Serial Interface is controlled by two registers:

- SCON - serial control register
- SBUF - serial data buffer register

SCON (memory address 98H)
Has bits to define:
- number of bits sent (mode bits)
- optional 9th send and receive bits
- Receive enable bit (REN)
- TI – the transmit interrupt flag
  -(TI is set when char sent)
- RI – the receive interrupt bit
  -(RI is set when character has been received)

SBUF (99H) – one name - two registers
- when read, accesses last character received
- when written, stores byte to be sent via the Send Shift Register (here just part of the Serial port subsystem)

SCON – Serial Port Control Register

<table>
<thead>
<tr>
<th>SM0</th>
<th>SM1</th>
<th>SM2</th>
<th>REN</th>
<th>TB8</th>
<th>RB8</th>
<th>TI</th>
<th>RI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SM0-1 Mode Bit Definition

<table>
<thead>
<tr>
<th>SM0</th>
<th>SM1</th>
<th>Mode</th>
<th>Description</th>
<th>Baud Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>0</td>
<td>Shift Register</td>
<td>Clock/12</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>1</td>
<td>8 Bit UART</td>
<td>Variable</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>2</td>
<td>9 Bit UART</td>
<td>Clock/32 or 64</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>3</td>
<td>9 Bit UART</td>
<td>Variable</td>
</tr>
</tbody>
</table>

UART – Universal Asynchronous Receiver/Transmitter
Serial: Shift-registers & Stop/Start & control logic
Polling Serial I/O Demo

```assembly
org 00h
; Set up Timer 1 to act as clock for Serial Port
mov tm0d,#20h ; Set Timer0 to Mode 2
mov th1,#06dh ; Reload value
; SETUP SERIAL PORT
mov sc0,#50h ; Serial Mode
mov tc0,#40h ; Start Timer
; Save char to memory - as a Log
mov r0,#$60 ; Serial Buffer ptr
; Setup Done, Main Program starts here
lup: acall getch ; Get char from keybd
mov @r0,a ; put into Log buffer
inc r0 ; inc log pointer
clr c
subb a,#32 ; convert to upper case
acall putch ; Display Character
sjmp lup ; do again
; GETCH - get a character
getch: jnb ri,getch ; Character arrived?
mov a,sbuf ; Get it
clr ri ; Clear Ch Ready flag
ret ; return char in A
; PUTCH - Output a character
putch: clr ti ; Reset TI bit
mov sbuf,a ; Store ch in A to Output
gone: jnb ti,gone ; Wait until it's gone
ret ; return
```

### Interrupts

**Polling**

The processor continually checks the status of the device (polls) to see if some condition is satisfied.

**Polling Advantages**

- simple
- no unexpected events
- easy to program/debug
- Fast response

**Polling Disadvantages**

- Wastes CPU
- Nothing else happens until event has occurred
- Hard to have more than one pending event (programming becomes complex)

**Interrupts**

- **Polling**
- **Interrupts**

**Interrupts map Events to Subroutines**

Either:

- Different subroutine address called for each event
- each event has a (set of) distinct status bits
  these bits are tested by subroutine to determine what action to take.

**8031 processor supports both methods**

### Sources of Interrupt on 8051

<table>
<thead>
<tr>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IE0</td>
<td>External pin A changed state</td>
</tr>
<tr>
<td>TF0</td>
<td>Timer Flag 0 (timer counter overflow)</td>
</tr>
<tr>
<td>IE1</td>
<td>External Pin B changed state</td>
</tr>
<tr>
<td>TF1</td>
<td>Timer Flag 1 (timer 1 counter overflow)</td>
</tr>
<tr>
<td>RI+TI</td>
<td>Serial Port needs attention</td>
</tr>
</tbody>
</table>

**Interrupt Vectors**

The address jumped to by the hardware for each source of interrupt is fixed.

<table>
<thead>
<tr>
<th>Interrupt</th>
<th>Vector Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>0000H (not really an Interrupt)</td>
</tr>
<tr>
<td></td>
<td>// Start address of Program code</td>
</tr>
<tr>
<td>IE0</td>
<td>0003H</td>
</tr>
<tr>
<td>TF0</td>
<td>000BH</td>
</tr>
<tr>
<td>IE1</td>
<td>0013H</td>
</tr>
<tr>
<td>TF1</td>
<td>001BH</td>
</tr>
<tr>
<td>SBUF (TI or RI)</td>
<td>0023H</td>
</tr>
</tbody>
</table>

**Multiple Events on One Interrupt**

- There is one vector for the serial I/O device, but two sources of interrupt
- The Serial Interrupt Handler must test either RI or TI to determine what caused the interrupt.
CPU Action upon an Interrupt

- May clear interrupt flag
- Push PC onto stack (so can get back)
- Jump to location corresponding to interrupt source (replace PC with vector address)
- Start executing Interrupt Service Routine (ISR)

Returning from Interrupt

The RETI instruction must be used (not RET)

The RETI Instruction:

- pops PC off the stack (restores the PC)
- informs the interrupt subsystem that the handler has completed.

When Can Interrupts Occur?

An interrupt will be acknowledged at the end of the current instruction IF:

Device’s Flag is set
→ the Device has Requested Attention

AND
Interrupts are Enabled
→ Global Interrupt Enable Bit (EA/IE.7) is set

AND
Device’s Interrupt Enable bit is set
→ programmer is allowing this Device to Interrupt

AND
NONE of the following are true:

- an interrupt of same/greater priority is being serviced
  → stop Interrupt subroutines from being interrupted

- an RETI is in progress
  → let the Interrupt handler return to the mainline

- the current instruction is modifying IP or IE
  → interrupt enable/priority registers being updated

- the current instruction has not completed
  → only acknowledge interrupts between instructions

Minimal Serial Interrupt Demo

```asm
;-------------------------------------------
; Everything under interrupt - there is no mainline
; It uppercases characters and echoes them
; and also stores them at internal data ram location 60 ...
;-------------------------------------------
org 00h
sjmp main
;-------------------------------------------
; Put an code at serial interrupt vector
org $23
sjmp serint
;-------------------------------------------
org 30h
main: mov tmmod,#20h ; Timer1 Mode 2
mov th1,#0fdh ; Reload value
mov scon,#50h ; Serial Mode
mov tcon,#40h ; Start Timer
mov r0,$#60h ; Display pointer
mov sp,#10h ; Define stack area
;-------------------------------------------
; INTERRUPT INITIALISATION
setb ES ; Serial interrupts ON
setb EA  ; Master interrupt ON
; INTERRUPTS ARE NOW ACTIVE
;-------------------------------------------
; Main Program
dead: sjmp dead ; Sit in Infinite Loop
; or a more useful program
```

Pending Interrupts

Although a device may request attention, an interrupt may not happen immediately, or at all.

Interrupts will be delayed:

- until end of current instruction
- OR
- until end of higher priority interrupt handler

The usual sources of Interrupt (RI, TI, Timer Flag bits) won’t cause an interrupt if:

Global interrupt bit (EA) = 0 // All Interrupts Suppressed

OR

Device interrupt enable bit = 0 // Specific Device // Interrupt suppressed

The Device Status bits are Always Valid

Changes in the state of the processor's I/O devices will always be shown by the status bits (eg RI or TI),
→ but these conditions may not cause Interrupts.
Serial Interrupt Handler
There will be a Serial Interrupt whenever the user presses a key:

```asm
; What caused Interrupt
jb   ri, RCV_ch ; Is RI set, Yes - char arrived
clr  ti ; No - must be TI
re ti ; Ignore Transmit; Interrupts

;-------------------------------------------
; RECEIVE INTERRUPT HANDLER
RCV_ch: push psw ; Save Registers
push acc
mov a, sbuf ; Get the char
clr ri ; Clear Rcv flag
mov &r0, a ; Put char in IRAM
inc r0
clr c
subb a, #32 ; ch to uppercase
mov sbuf, a ; now send char so
; user can see it

Exit: pop acc ; Restore Acc
pop psw ; and PSW
re ti ; return to mainline
```

Interrupts – System Setup

Initialisation
Setting up conditions that will allow interrupts to happen

Interrupt Handler
Taking appropriate action on each interrupt

Termination
Ensuring that interrupts won’t happen if they’re no longer needed (eg a TI interrupt must be deactivated if there’s no more data to send)

Initialisation
- Define the interrupt handler subroutine
- it must preserve any registers that it uses
  - Save registers on entry to routine
  - restore them just before RETI
- Put a Jump to subroutine instruction at the interrupt vector location for that specific interrupt
- Make sure Stack pointer is defined
- Set Device-specific Interrupt-enable bit

All is now set up:
- Allow Interrupts:
  - Set Master Interrupt Enable bit // System is LIVE

Common problems using Interrupts
- not allowing for increased stack use caused by interrupt handler
- leaving interrupts active when no longer required
- interrupt happen unexpectedly
  → must protect shared variables (simplest technique is to temporarily disable interrupts)

Danger of Shared Variable Access
If specific measures aren’t taken to stop unexpected access to variables shared between (or accessed by both) Interrupt Subroutines and the main program, unexpected results will almost certainly be obtained.

This is because, mentally, the programmer expects statements to execute in the order they’re written, but between any pair of statements, an Interrupt may occur.

This produces an unexpected (and unpredictable) change in the order statements are executed.

This can result in:
- unexpected changes in data values
- erratic program behaviour (different each time you run the program)

These bugs are very difficult to diagnose. It is best to stop such bugs happening rather than try and find them later

Example of Shared Variables & Interrupts
Consider a Clock Interrupt Routine invoked via interrupt every second, using globals to hold the current time-of-day:

```pascal
var hours, minutes, seconds;
//Globals: current time

procedure Clock_Interrupt_Handler;
// called every second
begin
  inc(seconds)
  if seconds = 60
  second:= 0
  inc(minutes);
  if minutes = 60 then
  minutes:= 0;
  inc(hours);
  if hours = 24 then
  hour:= 0;
end;

// User access to Time
Procedure GetTime(var h,m,s);
begin
  h:= hours;
  m:= minutes;
  s:= seconds;
end;
```

If time is 8:59:59:
after: h:=hours
a Clock Interrupt occurs!!!
Interrupt Handler then updates all of hours, minutes, & seconds to 9:0:0
Returned time will be will 8:0:0
WRONG!

Guarantee Exclusive Access to Hours, Minutes & Seconds:

```pascal
Procedure GetTime(var h,m,s);
begin
  Disable_Interrupts;
  h:= hours; m:= minutes s:= seconds;
  Enable_Interrupts;
end;
```