```
; To set the baud rate, use this formula or set to 0 for auto detection
; baud_const = 256 - (crystal / (12 * 16 * baud))

.equ    baud_const, 0           ;automatic baud rate detection
;.equ   baud_const, 255         ;57600 baud w/ 11.0592 MHz
;.equ   baud_const, 253         ;19200 baud w/ 11.0592 MHz
;.equ   baud_const, 252         ;19200 baud w/ 14.7456 MHz
;.equ   baud_const, 243         ;4808 baud w/ 12 MHz


;to do automatic baud rate detection, we assume the user will
;press the carriage return, which will cause this bit pattern
;to appear on port 3 pin 0 (CR = ascii code 13, assume 8N1 format)
;
;              0 1 0 1 1 0 0 0 0 1
;              | |             | |
; start bit----+ +--lsb   msb--+ +----stop bit
;
;we'll start timer #1 in 16 bit mode at the transition between the
;start bit and the LSB and stop it between the MBS and stop bit.
;That will give approx the number of cpu cycles for 8 bits.  Divide
;by 8 for one bit and by 16 since the built-in UART takes 16 timer
;overflows for each bit.  We need to be careful about roundoff during
;division and the result has to be inverted since timer #1 counts up.  Of
;course, timer #1 gets used in 8-bit auto reload mode for generating the
;built-in UART's baud rate once we know what the reload value should be.

autobaud:
        mov     tmod, #0x11     ;get timer #1 ready for action (16 bit mode)
        mov     tcon, #0x00
        clr     a
        mov     th1, a
        mov     tl1, a
        mov     a, #baud_const  ;skip if user supplied baud rate constant
        jnz     autoend
        mov     a, 0x7B         ;is there a value from a previous boot?
        xrl     0x7A, #01010101b
        xrl     0x79, #11001100b
        xrl     0x78, #00011101b
        cjne    a, 0x7A, autob2
        cjne    a, 0x79, autob2
        cjne    a, 0x78, autob2
        sjmp    autoend
autob2: jb      p3.0, *         ;wait for start bit
        jb      p3.0, autob2
        jb      p3.0, autob2    ;  check it a few more times to make
        jb      p3.0, autob2    ;  sure we don't trigger on some noise
        jb      p3.0, autob2
        jnb     p3.0, *         ;wait for bit #0 to begin
        setb    tr1             ;and now we're timing it
        jb      p3.0, *         ;wait for bit #1 to begin
        jnb     p3.0, *         ;wait for bit #2 to begin
        jb      p3.0, *         ;wait for bit #4 to begin
        jnb     p3.0, *         ;wait for stop bit to begin
        clr     tr1             ;stop timing
        mov     a, tl1
        mov     c, acc.6        ;save bit 6 for rounding up if necessary
        mov     f0, c
        mov     c, acc.7        ;grab bit 7... it's the lsb we want
        mov     a, th1
        rlc     a               ;do the div by 128
        mov     c, f0
        addc    a, #0           ;round off if necessary
        cpl     a               ;invert since timer #1 will count up
```

```
        inc     a               ;now acc has the correct reload value (I hope)
autoend:mov     0x7B, a
        mov     0x7A, a         ;store the baud rate for next warm boot.
        mov     0x79, a
        mov     0x78, a
        xrl     0x7A, #01010101b
        xrl     0x79, #11001100b
        xrl     0x78, #00011101b
        mov     th1, a
        mov     tl1, a
        mov     tmod, #0x21     ;set timer #1 for 8 bit auto-reload
        mov     pcon, #0x80     ;configure built-in uart
        mov     scon, #0x52
        setb    tr1             ;start the baud rate timer
        mov     r0, #0
        djnz    r0, *
        djnz    r0, *
        ret
```

```
;Serial I/O routines using the 8051's built-in UART.
;Almost all of these should use CIN and COUT, so they
;could pretty easily be adapted to other devices which
;could have similar single character I/O routines,
;including the 8051's UART using interrupts and buffers
;in memory.

;Much of this code appears in PAULMON1... see the
;PAULMON1.EQU file for an example of how to use some
;of these routines.


    ;timer reload calculation
    ; baud_const = 256 - (crystal / (12 * 16 * baud))

.equ    baud_const, 252          ;19200 baud w/ 15 MHz
;.equ    baud_const, 243 ;4800 baud w/ 12 MHz


;----------------------------------------------------------;
;                                                          ;
;               Subroutines for serial I/O                 ;
;                                                          ;
;----------------------------------------------------------;


cin:    jnb     ri, cin
        clr     ri
        mov     a, sbuf
        ret

cout:   jnb     ti, cout
        clr     ti
        mov     sbuf, a
        ret

newline:push    acc
        mov     a, #13
        acall   cout
        mov     a, #10
        acall   cout
        pop     acc
        ret

        ;get 2 digit hex number from serial port
        ; c = set if ESC pressed, clear otherwise
        ; psw.5 = set if return w/ no input, clear otherwise
ghex:
ghex8:  clr     psw.5
ghex8c:
        acall   cin             ;get first digit
        acall   upper
        cjne    a, #27, ghex8f
ghex8d: setb    c
        clr     a
        ret
ghex8f: cjne    a, #13, ghex8h
        setb    psw.5
        clr     c
        clr     a
```

```
        ret
ghex8h: mov     r2, a
        acall   asc2hex
        jc      ghex8c
        xch     a, r2           ;r2 will hold hex value of 1st digit
        acall   cout
ghex8j:
        acall   cin             ;get second digit
        acall   upper
        cjne    a, #27, ghex8k
        sjmp    ghex8d
ghex8k: cjne    a, #13, ghex8m
        mov     a, r2
        clr     c
        ret
ghex8m: cjne    a, #8, ghex8p
ghex8n: acall   cout
        sjmp    ghex8c
ghex8p: cjne    a, #21, ghex8q
        sjmp    ghex8n
ghex8q: mov     r3, a
        acall   asc2hex
        jc      ghex8j
        xch     a, r3
        acall   cout
        mov     a, r2
        swap    a
        orl     a, r3
        clr     c
        ret



        ;carry set if esc pressed
        ;psw.5 set if return pressed w/ no input
ghex16:
        mov     r2, #0          ;start out with 0
        mov     r3, #0
        mov     r4, #4          ;number of digits left
        clr     psw.5

ghex16c:
        acall   cin
        acall   upper
        cjne    a, #27, ghex16d
        setb    c               ;handle esc key
        clr     a
        mov     dph, a
        mov     dpl, a
        ret
ghex16d:cjne    a, #8, ghex16f
        sjmp    ghex16k
ghex16f:cjne    a, #127, ghex16g  ;handle backspace
ghex16k:cjne    r4, #4, ghex16e   ;have they entered anything yet?
        sjmp    ghex16c
ghex16e:acall   cout
        acall   ghex16y
        inc     r4
        sjmp    ghex16c
```

```
ghex16g:cjne    a, #13, ghex16i   ;return key
        mov     dph, r3
        mov     dpl, r2
        cjne    r4, #4, ghex16h
        clr     a
        mov     dph, a
        mov     dpl, a
        setb    psw.5
ghex16h:clr     c
        ret
ghex16i:mov     r5, a             ;keep copy of original keystroke
        acall   asc2hex
        jc      ghex16c
        xch     a, r5
        lcall   cout
        mov     a, r5
        push    acc
        acall   ghex16x
        pop     acc
        add     a, r2
        mov     r2, a
        clr     a
        addc    a, r3
        mov     r3, a
        djnz    r4, ghex16c
        clr     c
        mov     dpl, r2
        mov     dph, r3
        ret

ghex16x:  ;multiply r3-r2 by 16 (shift left by 4)
        mov     a, r3
        swap    a
        anl     a, #11110000b
        mov     r3, a
        mov     a, r2
        swap    a
        anl     a, #00001111b
        orl     a, r3
        mov     r3, a
        mov     a, r2
        swap    a
        anl     a, #11110000b
        mov     r2, a
        ret

ghex16y:  ;divide r3-r2 by 16 (shift right by 4)
        mov     a, r2
        swap    a
        anl     a, #00001111b
        mov     r2, a
        mov     a, r3
        swap    a
        anl     a, #11110000b
        orl     a, r2
        mov     r2, a
        mov     a, r3
        swap    a
        anl     a, #00001111b
        mov     r3, a
```

```
        ret


asc2hex:        ;carry set if invalid input
        clr     c
        push    b
        subb    a,#'0'
        mov     b,a
        subb    a,#10
        jc      a2h1
        mov     a,b
        subb    a,#7
        mov     b,a
a2h1:   mov     a,b
        clr     c
        anl     a,#11110000b     ;just in case
        jz      a2h2
        setb    c
a2h2:   mov     a,b
        pop     b
        ret


phex:
phex8:
        push    acc
        swap    a
        anl     a, #15
        add     a, #246
        jnc     phex_b
        add     a, #7
phex_b: add     a, #58
        acall   cout
        pop     acc
phex1:  push    acc
        anl     a, #15
        add     a, #246
        jnc     phex_c
        add     a, #7
phex_c: add     a, #58
        acall   cout
        pop     acc
        ret


PHEX16:
        PUSH    ACC
        MOV     A,DPH
        ACALL   PHEX
        MOV     A,DPL
        ACALL   PHEX
        POP     ACC
        RET


PSTR:                   ;print string @DPTR
        PUSH    ACC
PSTR1:  CLR     A
        MOVC    A,@A+DPTR
        JZ      PSTR2
```

```
        mov     c, acc.7
        anl     a, #01111111b
        acall   cout
        Jc      pstr2
        inc     dptr
        SJMP    PSTR1
PSTR2:  POP     ACC
        RET




;first we initialize all the registers we can, setting up
;for serial communication.

poweron:

        mov     sp, #0x30
        clr     psw.3           ;set for register bank 0 (init needs it)
        clr     psw.4
        orl     PCON,#10000000b   ; set double baud rate
        MOV     TMOD,#00010001b
        MOV     SCON,#01010000b  ; Set Serial for mode 1 &
                                 ; Enable reception
        ORL     TCON,#01010010b  ; Start timer 1 both timer

        mov     a, #baud_const
        mov     th1, a

        setb    ti              ;ti is normally set in this program
        clr     ri              ;ri is normally cleared

        ;jump to main program from here...



pint8u: ;prints the unsigned 8 bit value in Acc in base 10
        push    b
        push    acc
        sjmp    pint8b

pint8:  ;prints the signed 8 bit value in Acc in base 10
        push    b
        push    acc
        jnb     acc.7, pint8b
        mov     a, #'-'
        lcall   cout
        pop     acc
        push    acc
        cpl     a
        add     a, #1
pint8b: mov     b, #100
        div     ab
        setb    f0
        jz      pint8c
        clr     f0
        add     a, #'0'
        lcall   cout
```

```
pint8c: mov     a, b
        mov     b, #10
        div     ab
        jnb     f0, pint8d
        jz      pint8e
pint8d: add     a, #'0'
        lcall   cout
pint8e: mov     a, b
        add     a, #'0'
        lcall   cout
        pop     acc
        pop     b
        ret




        ;print 16 bit unsigned integer in DPTR, using base 10.
pint16u:;warning, destroys r2, r3, r4, r5, psw.5
        push    acc
        mov     a, r0
        push    acc
        clr     psw.5
        mov     r2, dpl
        mov     r3, dph

pint16a:mov     r4, #16 ;ten-thousands digit
        mov     r5, #39
        acall   pint16x
        jz      pint16b
        add     a, #'0'
        lcall   cout
        setb    psw.5

pint16b:mov     r4, #232;thousands digit
        mov     r5, #3
        acall   pint16x
        jnz     pint16c
        jnb     psw.5, pint16d
pint16c:add     a, #'0'
        lcall   cout
        setb    psw.5

pint16d:mov     r4, #100;hundreds digit
        mov     r5, #0
        acall   pint16x
        jnz     pint16e
        jnb     psw.5, pint16f
pint16e:add     a, #'0'
        lcall   cout
        setb    psw.5

pint16f:mov     a, r2           ;tens digit
        mov     r3, b
        mov     b, #10
        div     ab
        jnz     pint16g
        jnb     psw.5, pint16h
pint16g:add     a, #'0'
```

```
        lcall   cout

pint16h:mov     a, b            ;and finally the ones digit
        mov     b, r3
        add     a, #'0'
        lcall   cout

        pop     acc
        mov     r0, a
        pop     acc
        ret

;ok, it's a cpu hog and a nasty way to divide, but this code
;requires only 21 bytes!  Divides r2-r3 by r4-r5 and leaves
;quotient in r2-r3 and returns remainder in acc.  If Intel
;had made a proper divide, then this would be much easier.

pint16x:mov     r0, #0
pint16y:inc     r0
        clr     c
        mov     a, r2
        subb    a, r4
        mov     r2, a
        mov     a, r3
        subb    a, r5
        mov     r3, a
        jnc     pint16y
        dec     r0
        mov     a, r2
        add     a, r4
        mov     r2, a
        mov     a, r3
        addc    a, r5
        mov     r3, a
        mov     a, r0
        ret


upper:  ;converts the ascii code in Acc to uppercase, if it is lowercase
        push    acc
        clr     c
        subb    a, #97
        jc      upper2          ;is it a lowercase character
        subb    a, #26
        jnc     upper2
        pop     acc
        add     a, #224 ;convert to uppercase
        ret
upper2: pop     acc             ;don't change anything
        ret




pbin:   mov     r0, #8
pbin2:  rlc     a
        mov     f0, c
        push    acc
        mov     a, #'0'
        addc    a, #0
```

```
        lcall   cout
        pop     acc
        mov     c, f0
        djnz    r0, pbin2
        rlc     a
        ret


lenstr: mov     r0, #0    ;returns length of a string in r0
        push    acc
lenstr1:clr     a
        movc    a,@a+dptr
        jz      lenstr2
        mov     c,acc.7
        inc     r0
        Jc      lenstr2
        inc     dptr
        sjmp    lenstr1
lenstr2:pop     acc
        ret


.equ    str_buf, 0x20           ;16 byte buffer
.equ    max_str_len, 19

getstr: ;get a string and store in an internal ram buffer
        ;  str_buf = beginning of the buffer
        ;  max_str_len = max number of char to receive
        ;   (buffer must be one larger for null termination)

        mov     r0, #str_buf
gstrz:  mov     @r0, #0                 ;fill buffer with zeros
        inc     r0
        cjne    r0, #(str_buf+max_str_len+1), gstrz
        mov     r0, #str_buf
gstr_in:lcall   cin
        lcall   isascii
        jnc     gstr_ctrl
        cjne    r0, #(str_buf+max_str_len), gstradd
        sjmp    gstr_in
gstradd:lcall   cout
        mov     @r0, a
        inc     r0
        sjmp    gstr_in
gstr_ctrl:
        cjne    a, #13, gstrc2          ;carriage return
        clr     a
        mov     @r0, a
        ret
gstrc2: cjne    a, #8, gstrc3           ;backspace
gstrbk: cjne    r0, #str_buf, gstrbk2
        sjmp    gstr_in
gstrbk2:mov     a, #8
        lcall   cout
        mov     a, #' '
        lcall   cout
        mov     a, #8
        lcall   cout
        dec     r0
        sjmp    gstr_in
```

```
gstrc3: cjne    a, #127, gstrc4         ;delete
        sjmp    gstrbk
gstrc4:
        sjmp    gstr_in                 ;ignore all others



pstrbuf:  ;print the string in the internal ram buffer
        mov     r0, #str_buf
pstrbuf2:
        mov     a, @r0
        jz      pstrbuf3
        lcall   cout
        inc     r0
        sjmp    pstrbuf2
pstrbuf3:
        ret




        ;get unsigned integer input to acc
gint8u:
        mov     r0, #0          ;r0 holds sum so far
        mov     r1, #0          ;r1 counts number of characters
gi8_in: lcall   cin
        mov     r2, a           ;r2 is temp holding space for input char
        clr     c
        subb    a, #'0'
        jc      gi8_ctrl
        subb    a, #10
        jnc     gi8_ctrl
        mov     a, r0
        mov     b, #10
        mul     ab
        xch     a, b
        jnz     gi8_in
        mov     a, r2
        clr     c
        subb    a, #'0'
        add     a, b
        jc      gi8_in
        mov     r0, a
        mov     a, r2
        lcall   cout
        inc     r1
        sjmp    gi8_in
gi8_ctrl:
        mov     a, r2
        cjne    a, #13, gi8c2
        mov     a, r0
        ret
gi8c2:  cjne    a, #8, gi8c3
gi8bk:  cjne    r1, #0, gi8bk2
        sjmp    gi8_in
gi8bk2: mov     a, #8
        lcall   cout
        mov     a, #' '
```

```
        lcall   cout
        mov     a, #8
        lcall   cout
        mov     a, r0
        mov     b, #10
        div     ab
        mov     r0, a
        sjmp    gi8_in
gi8c3:  cjne    a, #127, gi8c4
        sjmp    gi8bk
gi8c4:
        sjmp    gi8_in



isascii:        ;is acc an ascii char, c=1 if yes, c=0 if no
        push    acc
        cjne    a, #0x7F, isasc2
        sjmp    isasc_no
isasc2: anl     a, #10000000b
        jnz     isasc_no
        pop     acc
        push    acc
        anl     a, #11100000b
        jz      isasc_no
        setb    c
        pop     acc
        ret
isasc_no:
        clr     c
        pop     acc
        ret
```

```
;==========================================================================
======

;                   Bootstrap Loader for Hexadecimal Files
;                    written by G. Goodhue, Signetics Co.

; This program downloading a hexadecimal program file over an
asynchronous
; serial link to a code RAM in an 80C51 system. The downloaded code may
then
; be executed as the main program for the system. This technique may be
used
; in a system that normally connects to a host PC so that the code may
come
; from a disk and thus be easily updated. The system RAM must be wired
to the
; 80C51 system so that it appears as both data and program memory (wire
the
; RAM normally, but use the logical AND of RD and PSEN for the output
enable.)

; To use the bootstrap program, an Intel Hex file is sent through the
serial
; port in 8-N-1 format at 9600 baud. The baud rate and format may be
altered
; by making small changes in the serial port setup routine (SerStart).

; Note that there is no hardware handshaking (e.g. RTS/CTS or XON/XOFF)
; implemented between the host and the bootstrap system. This was done
to keep
; the protocol between the two systems as simple as possible.

; Since the bootstrap program does not echo the data file, there is no
chance
; of an overrun unless the 80C51 is running very slowly and/or the
; communication is very fast. An 80C51 running at 11.0592 MHz (the most
; commonly used frequency in systems with serial communication) will be
able
; to easily keep up with 38.4K baud communication without handshaking.

;==========================================================================
======

; The download protocol for this program is as follows:

;  - When the bootstrap program starts up, it sends a prompt character
("=")
;    up the serial link to the host.

;  - The host may then send the hexadecimal program file down the serial
link.
;    At any time, the host may send an escape character (1B hex) to
abort and
;    restart the download process from scratch, beginning from the "="
prompt.
;    This procedure may be used to restart if a download error occurs.

;  - At the end of a hex file download, a colon (":") prompt is
returned. If
```

```
;    an error or other suspicious circumstance occurred, a flag value
will
;    also be returned as shown below. The flag is a bit map of possible
;    conditions and so may represent more than one problem. If an error
;    occurs, the bootstrap program will refuse to execute the downloaded
;    program.

;  Exception codes:
;    01 - non-hexadecimal characters found embedded in a data line.
;    02 - bad record type found.
;    04 - incorrect line checksum found.
;    08 - no data found.
;    10 - incremented address overflowed back to zero.
;    20 - RAM data write did not verify correctly.

;  - If a download error occurs, the download may be retried by first
sending
;    an escape character. Until the escape is received, the bootstrap
program
;    will refuse to accept any data and will echo a question mark ("?")
for
;    any character sent.

;  - After a valid file download, the bootstrap program will send a
message
;    containing the file checksum. This is the arithmetic sum of all of
the
;    DATA bytes (not addresses, record types, etc.) in the file,
truncated to
;    16 bits. This checksum appears in parentheses: "(abcd)". Program
;    execution may then be started by telling the bootstrap program the
;    correct starting address. The format for this is to send a slash
("/")
;    followed by the address in ASCII hexadecimal, followed by a
carriage
;    return. Example: "/8A31<CR>"

;  - If the address is accepted, an at sign ("@") is returned before
executing
;    the jump to the downloaded file.

; The bootstrap loader can be configured to re-map interrupt vectors to
the
; downloaded program if jumps to the correct addresses are set up. For
; instance, if the program RAM in the system where this program is to be
used
; starts at 8000 hexadecimal, the re-mapped interrupts may begin at 8003
for
; external interrupt 0, etc.


;=========================================================================
======


$Title(Bootstrap Loader for Hexadecimal Files)
$Date(04-13-92)
$MOD51
```

```
;=========================================================================
======
;                                   Definitions
;=========================================================================
======


LF          EQU     0Ah                 ; Line Feed character.
CR          EQU     0Dh                 ; Carriage Return character.
ESC         EQU     1Bh                 ; Escape character.
StartChar   EQU     ':'                 ; Line start character for hex
file.
Slash       EQU     '/'                 ; Go command character.
Skip        EQU     13                  ; Value for "Skip" state.

Ch          DATA     0Fh                ; Last character received.
State       DATA     10h                ; Identifies the state in
process.
DataByte    DATA     11h                ; Last data byte received.
ByteCount   DATA     12h                ; Data byte count from current
line.
HighAddr    DATA     13h                ; High and low address bytes from
the
LowAddr     DATA     14h                ;   current data line.
RecType     DATA     15h                ; Line record type for this line.
ChkSum      DATA     16h                ; Calculated checksum received.
HASave      DATA     17h                ; Saves the high and low address
bytes
LASave      DATA     18h                ;   from the last data line.
FilChkHi    DATA     19h                ; File checksum high byte.
FilChkLo    DATA     1Ah                ; File checksum low byte.

Flags       DATA     20h                ; State condition flags.
HexFlag     BIT      Flags.0            ; Hex character found.
EndFlag     BIT      Flags.1            ; End record found.
DoneFlag    BIT      Flags.2            ; Processing done (end record or
some
                                        ;   kind of error.

EFlags      DATA     21h                ; Exception flags.
ErrFlag1    BIT      EFlags.0           ;   Non-hex character embedded in
data.
ErrFlag2    BIT      EFlags.1           ;   Bad record type.
ErrFlag3    BIT      EFlags.2           ;   Bad line checksum.
ErrFlag4    BIT      EFlags.3           ;   No data found.
ErrFlag5    BIT      EFlags.4           ;   Incremented address overflow.
ErrFlag6    BIT      EFlags.5           ;   Data storage verify error.

DatSkipFlag BIT      Flags.3            ; Any data found should be
ignored.


;=========================================================================
======
;                           Reset and Interrupt Vectors
;=========================================================================
======


; The following are dummy labels for re-mapped interrupt vectors. The
```

```
; addresses should be changed to match the memory map of the target
system.

ExInt0       EQU     8003h              ; Remap address for ext interrupt
0.
T0Int        EQU     800Bh              ; Timer 0 interrupt.
ExInt1       EQU     8013h              ; External interrupt 1.
T1Int        EQU     801Bh              ; Timer 1 interrupt.
SerInt       EQU     8023h              ; Serial port interrupt.


             ORG      0000h
             LJMP     Start             ; Go to the downloader program.


; The following are intended to allow re-mapping the interrupt vectors
to the
; users downloaded program. The jump addresses should be adjusted to
reflect
; the memory mapping used in the actual application.

; Other (or different) interrupt vectors may need to be added if the
target
; processor is not an 80C51.

             ORG      0003h
;             LJMP     ExInt0            ; External interrupt 0.
             RETI

             ORG      000Bh
;             LJMP     T0Int             ; Timer 0 interrupt.
             RETI

             ORG      0013h
;             LJMP     ExInt1            ; External interrupt 1.
             RETI

             ORG      001Bh
;             LJMP     T1Int             ; Timer 1 interrupt.
             RETI

             ORG      0023h
;             LJMP     SerInt            ; Serial port interrupt.
             RETI


;=========================================================================
======
;                          Reset and Interrupt Vectors
;=========================================================================
======


Start:       MOV      IE,#0             ; Turn off all interrupts.
             MOV      SP,#5Fh           ; Start stack near top of '51
RAM.
             ACALL    SerStart          ; Setup and start serial port.
             ACALL    CRLF              ; Send a prompt that we are here.
             MOV      A,#'='            ;   "<CRLF> ="
             ACALL    PutChar
```

```
            ACALL   HexIn               ; Try to read hex file from
serial port.
            ACALL   ErrPrt              ; Send a message for any errors
or
                                        ;   warnings that were noted.
            MOV     A,EFlags            ; We want to get stuck if a fatal
            JZ      HexOK               ;   error occurred.

ErrLoop:    MOV     A,#'?'              ; Send a prompt to confirm that
we
            ACALL   PutChar             ;   are 'stuck'.  " ? "
            ACALL   GetChar             ; Wait for escape char to flag
reload.
            SJMP    ErrLoop

HexOK:      MOV     EFlags,#0           ; Clear errors flag in case we
re-try.
            ACALL   GetChar             ; Look for GO command.
            CJNE    A,#Slash,HexOK      ; Ignore other characters
received.

            ACALL   GetByte             ; Get the GO high address byte.
            JB      ErrFlag1,HexOK      ; If non-hex char found, try
again.
            MOV     HighAddr,DataByte ; Save upper GO address byte.

            ACALL   GetByte             ; Get the GO low address byte.
            JB      ErrFlag1,HexOK      ; If non-hex char found, try
again.
            MOV     LowAddr,DataByte  ; Save the lower GO address byte.

            ACALL   GetChar             ; Look for CR.
            CJNE    A,#CR,HexOK         ; Re-try if CR not there.

; All conditions are met, so hope the data file and the GO address are
all
;   correct, because now we're committed.

            MOV     A,#'@'              ; Send confirmation to GO. " @ "
            ACALL   PutChar
            JNB     TI,$                ; Wait for completion before
GOing.

            PUSH    LowAddr             ; Put the GO address on the
stack,
            PUSH    HighAddr            ;   so we can Return to it.
            RET                         ; Finally, go execute the user
program!


;=======================================================================
======
;                          Hexadecimal File Input Routine
;=======================================================================
======

HexIn:      CLR     A                   ; Clear out some variables.
            MOV     State,A
            MOV     Flags,A
            MOV     HighAddr,A
```

```
            MOV      LowAddr,A
            MOV      HASave,A
            MOV      LASave,A
            MOV      ChkSum,A
            MOV      FilChkHi,A
            MOV      FilChkLo,A
            MOV      EFlags,A
            SETB     ErrFlag4           ; Start with a 'no data'
condition.

StateLoop:  ACALL    GetChar            ; Get a character for processing.
            ACALL    AscHex             ; Convert ASCII-hex character to
hex.
            MOV      Ch,A               ; Save result for later.
            ACALL    GoState            ; Go find the next state based on
                                        ;   this char.
            JNB      DoneFlag,StateLoop ; Repeat until done or
terminated.

            ACALL    PutChar            ; Send the file checksum back as
            MOV      A,#'('             ;   confirmation. " (abcd) "
            ACALL    PutChar
            MOV      A,FilChkHi
            ACALL    PrByte
            MOV      A,FilChkLo
            ACALL    PrByte
            MOV      A,#')'
            ACALL    PutChar
            ACALL    CRLF
            RET                         ; Exit to main program.


; Find and execute the state routine pointed to by "State".

GoState:    MOV      A,State            ; Get current state.
            ANL      A,#0Fh             ; Insure branch is within table
range.
            RL       A                  ; Adjust offset for 2 byte insts.
            MOV      DPTR,#StateTable
            JMP      @A+DPTR            ; Go to appropriate state.

StateTable: AJMP     StWait             ;  0 - Wait for start.
            AJMP     StLeft             ;  1 - First nibble of count.
            AJMP     StGetCnt           ;  2 - Get count.
            AJMP     StLeft             ;  3 - First nibble of address
byte 1.
            AJMP     StGetAd1           ;  4 - Get address byte 1.
            AJMP     StLeft             ;  5 - First nibble of address
byte 2.
            AJMP     StGetAd2           ;  6 - Get address byte 2.
            AJMP     StLeft             ;  7 - First nibble of record
type.
            AJMP     StGetRec           ;  8 - Get record type.
            AJMP     StLeft             ;  9 - First nibble of data byte.
            AJMP     StGetDat           ; 10 - Get data byte.
            AJMP     StLeft             ; 11 - First nibble of checksum.
            AJMP     StGetChk           ; 12 - Get checksum.
            AJMP     StSkip             ; 13 - Skip data after error
condition.
            AJMP     BadState           ; 14 - Should never get here.
```

```
            AJMP     BadState          ; 15 -   "      "    "    "


; This state is used to wait for a line start character. Any other
characters
;   received prior to the line start are simply ignored.

StWait:     MOV      A,Ch              ; Retrieve input character.
            CJNE     A,#StartChar,SWEX ; Check for line start.
            INC      State             ; Received line start.
SWEX:       RET


; Process the first nibble of any hex byte.

StLeft:     MOV      A,Ch              ; Retrieve input character.
            JNB      HexFlag,SLERR     ; Check for hex character.
            ANL      A,#0Fh            ; Isolate one nibble.
            SWAP     A                 ; Move nibble too upper location.
            MOV      DataByte,A        ; Save left/upper nibble.
            INC      State             ; Go to next state.
            RET                        ; Return to state loop.

SLERR:      SETB     ErrFlag1          ; Error - non-hex character
found.
            SETB     DoneFlag          ; File considered corrupt. Tell
main.
            RET


; Process the second nibble of any hex byte.

StRight:    MOV      A,Ch              ; Retrieve input character.
            JNB      HexFlag,SRERR     ; Check for hex character.
            ANL      A,#0Fh            ; Isolate one nibble.
            ORL      A,DataByte        ; Complete one byte.
            MOV      DataByte,A        ; Save data byte.
            ADD      A,ChkSum          ; Update line checksum,
            MOV      ChkSum,A          ;   and save.
            RET                        ; Return to state loop.

SRERR:      SETB     ErrFlag1          ; Error - non-hex character
found.
            SETB     DoneFlag          ; File considered corrupt. Tell
main.
            RET


; Get data byte count for line.

StGetCnt:   ACALL    StRight           ; Complete the data count byte.
            MOV      A,DataByte
            MOV      ByteCount,A
            INC      State             ; Go to next state.
            RET                        ; Return to state loop.


; Get upper address byte for line.

StGetAd1:   ACALL    StRight           ; Complete the upper address
```

```
byte.
            MOV      A,DataByte
            MOV      HighAddr,A         ; Save new high address.
            INC      State              ; Go to next state.
            RET                         ; Return to state loop.


; Get lower address byte for line.

StGetAd2:   ACALL    StRight            ; Complete the lower address
byte.
            MOV      A,DataByte
            MOV      LowAddr,A          ; Save new low address.
            INC      State              ; Go to next state.
            RET                         ; Return to state loop.


; Get record type for line.

StGetRec:   ACALL    StRight            ; Complete the record type byte.
            MOV      A,DataByte
            MOV      RecType,A          ; Get record type.
            JZ       SGRDat             ; This is a data record.
            CJNE     A,#1,SGRErr        ; Check for end record.
            SETB     EndFlag            ; This is an end record.
            SETB     DatSkipFlag        ; Ignore data embedded in end
record.
            MOV      State,#11          ; Go to checksum for end record.
            SJMP     SGREX

SGRDat:     INC      State              ; Go to next state.
SGREX:      RET                         ; Return to state loop.

SGRErr:     SETB     ErrFlag2           ; Error, bad record type.
            SETB     DoneFlag           ; File considered corrupt. Tell
main.
            RET


; Get a data byte.

StGetDat:   ACALL    StRight            ; Complete the data byte.
            JB       DatSkipFlag,SGD1   ; Don't process the data if the
skip
                                        ;   flag is on.
            ACALL    Store              ; Store data byte in memory.

            MOV      A,DataByte         ; Update the file checksum,
            ADD      A,FilChkLo         ;   which is a two-byte summation
of
            MOV      FilChkLo,A         ;   all data bytes.
            CLR      A
            ADDC     A,FilChkHi
            MOV      FilChkHi,A
            MOV      A,DataByte
SGD1:       DJNZ     ByteCount,SGDEX    ; Last data byte?
            INC      State              ; Done with data, go to next
state.
            SJMP     SGDEX2
```

```
SGDEX:       DEC      State              ; Set up state for next data
byte.
SGDEX2:      RET                         ; Return to state loop.


; Get checksum.

StGetChk:    ACALL    StRight            ; Complete the checksum byte.
             JNB      EndFlag,SGC1       ; Check for an end record.
             SETB     DoneFlag           ; If this was an end record,
             SJMP     SGCEX              ;  we are done.

SGC1:        MOV      A,ChkSum           ; Get calculated checksum.
             JNZ      SGCErr             ; Result should be zero.
             MOV      ChkSum,#0          ; Preset checksum for next line.
             MOV      State,#0           ; Line done, go back to wait
state.
             MOV      LASave,LowAddr     ; Save address byte from this
line for
             MOV      HASave,HighAddr    ;   later check.
SGCEX:       RET                         ; Return to state loop.

SGCErr:      SETB     ErrFlag3           ; Line checksum error.
             SETB     DoneFlag           ; File considered corrupt. Tell
main.
             RET


; This state used to skip through any additional data sent, ignoring it.

StSkip:      RET                         ; Return to state loop.


; A place to go if an illegal state comes up somehow.

BadState:    MOV      State,#Skip        ; If we get here, something very
bad
             RET                         ;   happened, so return to state
loop.


; Store - Save data byte in external RAM at specified address.

Store:       MOV      DPH,HighAddr       ; Set up external RAM address in
DPTR.
             MOV      DPL,LowAddr
             MOV      A,DataByte
             MOVX     @DPTR,A            ; Store the data.

             MOVX     A,@DPTR            ; Read back data for integrity
check.
             CJNE     A,DataByte,StoreErr ; Is read back OK?

             CLR      ErrFlag4           ; Show that we've found some
data.
             INC      DPTR               ; Advance to the next addr in
sequence.
             MOV      HighAddr,DPH       ; Save the new address
             MOV      LowAddr,DPL
             CLR      A
```

```
            CJNE     A,HighAddr,StoreEx ; Check for address overflow
            CJNE     A,LowAddr,StoreEx ;   (both bytes are 0).
            SETB     ErrFlag5          ; Set warning for address
overflow.
StoreEx:    RET

StoreErr:   SETB     ErrFlag6          ; Data storage verify error.
            SETB     DoneFlag          ; File considered corrupt. Tell
main.
            RET


;========================================================================
======
;                                   Subroutines
;========================================================================
======


; Subroutine summary:

; SerStart - Serial port setup and start.
; GetChar  - Get a character from the serial port for processing.
; GetByte  - Get a hex byte from the serial port for processing.
; PutChar  - Output a character to the serial port.
; AscHex   - See if char in ACC is ASCII-hex and if so convert to hex
nibble.
; HexAsc   - Convert a hexadecimal nibble to its ASCII character
equivalent.
; ErrPrt   - Return any error codes to our host.
; CRLF     - output a carriage return / line feed pair to the serial
port.
; PrByte   - Send a byte out the serial port in ASCII hexadecimal
format.


; SerStart - Serial port setup and start.

SerStart:   MOV      A,PCON            ; Make sure SMOD is off.
            CLR      ACC.7
            MOV      PCON,A
            MOV      TH1,#0FDh         ; Set up timer 1.
            MOV      TL0,#0FDh
            MOV      TMOD,#20h
            MOV      TCON,#40h
            MOV      SCON,#52h         ; Set up serial port.
            RET


; GetByte - Get a hex byte from the serial port for processing.

GetByte:    ACALL    GetChar           ; Get first character of byte.
            ACALL    AscHex            ; Convert to hex.
            MOV      Ch,A              ; Save result for later.
            ACALL    StLeft            ; Process as top nibble of a hex
byte.
            ACALL    GetChar           ; Get second character of byte.
            ACALL    AscHex            ; Convert to hex.
            MOV      Ch,A              ; Save result for later.
            ACALL    StRight           ; Process as bottom nibble of hex
```

```
byte.
             RET


; GetChar - Get a character from the serial port for processing.

GetChar:     JNB      RI,$                 ; Wait for receiver flag.
             CLR      RI                   ; Clear receiver flag.
             MOV      A,SBUF               ; Read character.
             CJNE     A,#ESC,GCEX          ; Re-start immediately if Escape
char.
             LJMP     Start
GCEX:        RET


; PutChar - Output a character to the serial port.

PutChar:     JNB      TI,$                 ; Wait for transmitter flag.
             CLR      TI                   ; Clear transmitter flag.
             MOV      SBUF,A               ; Send character.
             RET


; AscHex - See if char in ACC is ASCII-hex and if so convert to a hex
nibble.
;   Returns nibble in A, HexFlag tells if char was really hex. The ACC
is not
;   altered if the character is not ASCII hex. Upper and lower case
letters
;   are recognized.

AscHex:      CJNE     A,#'0',AH1           ; Test for ASCII numbers.
AH1:         JC       AHBad                ; Is character is less than a
'0'?
             CJNE     A,#'9'+1,AH2         ; Test value range.
AH2:         JC       AHVal09              ; Is character is between '0' and
'9'?

             CJNE     A,#'A',AH3           ; Test for upper case hex
letters.
AH3:         JC       AHBad                ; Is character is less than an
'A'?
             CJNE     A,#'F'+1,AH4         ; Test value range.
AH4:         JC       AHValAF              ; Is character is between 'A' and
'F'?

             CJNE     A,#'a',AH5           ; Test for lower case hex
letters.
AH5:         JC       AHBad                ; Is character is less than an
'a'?
             CJNE     A,#'f'+1,AH6         ; Test value range.
AH6:         JNC      AHBad                ; Is character is between 'a' and
'f'?
             CLR      C
             SUBB     A,#27h               ; Pre-adjust character to get a
value.
             SJMP     AHVal09              ; Now treat as a number.

AHBad:       CLR      HexFlag              ; Flag char as non-hex, don't
alter.
```

```
            SJMP     AHEX                 ; Exit
AHValAF:    CLR      C
            SUBB     A,#7                 ; Pre-adjust character to get a
value.
AHVal09:    CLR      C
            SUBB     A,#'0'               ; Adjust character to get a
value.
            SETB     HexFlag              ; Flag character as 'good' hex.
AHEX:       RET


; HexAsc - Convert a hexadecimal nibble to its ASCII character
equivalent.

HexAsc:     ANL      A,#0Fh               ; Make sure we're working with
only
                                          ;   one nibble.
            CJNE      A,#0Ah,HA1          ; Test value range.
HA1:        JC        HAVal09             ; Value is 0 to 9.
            ADD       A,#7                ; Value is A to F, extra
adjustment.
HAVal09:    ADD       A,#'0'              ; Adjust value to ASCII hex.
            RET


; ErrPrt - Return an error code to our host.

ErrPrt:     MOV       A,#':'              ; First, send a prompt that we
are
            CALL      PutChar             ;   still here.
            MOV       A,EFlags            ; Next, print the error flag
value if
            JZ        ErrPrtEx            ;   it is not 0.
            CALL      PrByte
ErrPrtEx:   RET


; CRLF - output a carriage return / line feed pair to the serial port.

CRLF:       MOV       A,#CR
            CALL      PutChar
            MOV       A,#LF
            CALL      PutChar
            RET


; PrByte - Send a byte out the serial port in ASCII hexadecimal format.

PrByte:     PUSH      ACC                 ; Print ACC contents as ASCII
hex.
            SWAP      A
            CALL      HexAsc              ; Print upper nibble.
            CALL      PutChar
            POP       ACC
            CALL      HexAsc              ; Print lower nibble.
            CALL      PutChar
            RET


;=======================================================================
```

======

END

```
*
* "Bit-bang" serial I/O functions for the 8051.
*
* These routines transmit and receive serial data using two general
* I/O pins, in 8 bit, No parity, 1 stop bit format. They are useful
* for performing serial I/O on 8051 derivatives not having an
* internal UART, or for implementing a second serial channel.
*
* Dave Dunfield - May 17, 1994
*
* NOTE that R0 and R1 are used by the functions. You may wish to
* add PUSH/POP instructions to save/restore these registers.
*
TXD   EQU   P1.0        Transmit on this pin
RXD   EQU   P1.1        Receive on this pin
* The serial baud rate is determined by the processor crystal, and
* this constant which is calculated as: (((crystal/baud)/12) - 5) / 2
BITTIM      EQU   45          (((11059200/9600)/12) - 5) / 2
*
* Transmit character in A via TXD line
*
putc  CLR   TXD         Drop line for start bit
      MOV   R0,#BITTIM  Wait full bit-time
      DJNZ  R0,*        For START bit
      MOV   R1,#8       Send 8 bits
putc1 RRC   A           Move next bit into carry
      MOV   TXD,C       Write next bit
      MOV   R0,#BITTIM  Wait full bit-time
      DJNZ  R0,*        For DATA bit
      DJNZ  R1,putc1    write 8 bits
      SETB  TXD         Set line high
      RRC   A           Restore ACC contents
      MOV   R0,#BITTIM  Wait full bit-time
      DJNZ  R0,*        For STOP bit
      RET
*
* Receive a character from the RXD line and return in A
*
getc  JB    RXD,*       Wait for start bit
      MOV   R0,#BITTIM/2      Wait 1/2 bit-time
      DJNZ  R0,*        To sample in middle
      JB    RXD,getc    Insure valid
      MOV   R1,#8       Read 8 bits
getc1 MOV   R0,#BITTIM  Wait full bit-time
      DJNZ  R0,*        For DATA bit
      MOV   C,RXD       Read bit
      RRC   A           Shift it into ACC
      DJNZ  R1,getc1    read 8 bits
      RET               go home
```

Application Note

MCS® 51 On-Chip UART

A Simplified Users Guide

Overview

The MCS®-51 family contains a flexible set of microcontrollers. These 8-bit embedded controllers have different features such as on-chip program memory, data RAM and some even have integrated A/D converters. One feature that all of the microcontrollers in the MCS®-51 family have in common is an integrated UART (Universal Asynchronous Receiver Transmitter).

This guide has been designed so that any programmer with basic microcontroller experience can learn how to use the general features of the on-chip UART in a MCS®-

51 microcontroller. This document has been created and designed in response to repeated inquires on the usage of the serial port. Working examples have been included and explained to ease the learning process.

The serial port can operate in 4 modes:
Mode 0: TXD outputs the shift clock. In this mode, 8 bits are transmitted ***and*** received by the same pin, RXD. The data is transmitted starting with the least significant bit first, and ending with the most significant bit. The baud rate is fixed at 1/12 the oscillator frequency.

Mode 1: Serial data enters through the RXD pin and exits through the TXD pin. In this mode, a start bit of logic level 0 is transmitted then 8 bits are transmitted with the least significant bits first up to the most significant bit; following the most significant bit is the stop bit which is a logic 1. When receiving data in this mode, the stop bit is placed into RB8 in the SFR (Special Function Register) SCON. The baud rate is variable and is controlled by either timer 1 or timer 2 reload values.

Mode 2: Serial data enters through the RXD pin and exits through the TXD pin. In this mode, a total of 11 bits are transmitted or received starting with a start bit of logic level 0, 8 bits of data with the least significant bit first, a user programmable ninth data bit, and a stop bit of logic level 1. The ninth data bit is the value of the TB8 bit inside the SCON register. This programmable bit is often used for parity information. The baud rate is programmable to either 1/32 or 1/64 of the oscillator frequency.

Mode 3: Mode three is identical to mode 2 except that the baud rate is variable and is controlled by either timer 1 or timer 2 reload values.

For more detailed information on each serial port mode, refer to the "Hardware Description of the 8051, 8052, and 80c51." in the 1993 Embedded Microcontrollers and Processors (270645).

Baud Rate Generation Using Timer Two

$$\text{Baud Rate} = \frac{F_{osc}}{(32(65536-(RCAP2H,RCAP2L)))}$$

$$(\text{RCAP2H,RCAP2L}) = 65536 - \frac{F_{osc}}{32*(BaudRate)}$$

RCAP2L and RCAP2H are 8-bit registers combined as a 16-bit entity that timer 2 uses as a reload value. Each time timer 2 overflows (goes one past FFFFH), this 16-bit reload value is placed back into the timer, and the timer begins to count up from there until it overflows again. Each time the timer overflows, it signals the processor to send a data bit out the serial port. The larger the reload value (RCAP2H, RCAP2L), the more frequently the data bits are transmitted out the serial port. This frequency of data bits transmitted or received is known as the baud rate.

Table One

Baud Rate	Freq (Mhz)	RCAP2H	RCAP2L	Baud Rate	Freq (Mhz)	RCAP2H	RCAP2L
38,400	16	FF	F3	56,800	11.059	FF	FA
19,200	16	FF	E6	38,400	11.059	FF	F7
9,600	16	FF	CC	19,200	11.059	FF	EE
4,800	16	FF	98	9,600	11.059	FF	DC
2,400	16	FF	30	4,800	11.059	FF	B8
1,200	16	FE	5F	2,400	11.059	FF	70
600	16	FC	BF	1,200	11.059	FE	E0
300	16	F9	7D	600	11.059	FD	C0
110	16	EE	3F	300	11.059	FB	80
375,000	12	FF	FF	4,800	6	FF	D9
9,600	12	FF	D9	2,400	6	FF	B2
4,800	12	FF	B2	1,200	6	FF	64

2,400	12	FF	64	600	6	FE	C8
1,200	12	FE	C8	300	6	FD	8F
600	12	FD	8F	110	6	F9	57
300	12	FB	1E				

Baud Rate Generation Using Timer One

$$\text{Baud Rate} = \frac{2^{SMOD} F_{OSC}}{(384(256 - TH1))}$$

$$TH1 = 256 - \frac{2^{SMOD} F_{OSC}}{BaudRate * 384}$$

Similar to timer 2, TH1 is an 8-bit register that timer 1 uses as it's reload value. The larger the number placed in TH1, the faster the baud rate. SMOD1 is bit position 7 in the PCON register. This bit is called the "Double Baud Rate Bit". When the serial port is in mode 1, 2 or 3 and timer 1 is being used as the baud rate generator, the baud rate can be doubled by setting SMOD1. For example; TH1 equals DDH and the oscillator frequency equals 16Mhz, then the baud rate equals 2400 baud if SMOD1 is set. If SMOD1 is cleared, for the same example, then the baud rate would be 1200.

Table Two

Baud Rate	Freq (Mhz)	SMOD1	TH1	Baud Rate	Freq (Mhz)	SMOD1	TH1
4,800	16	1	EF	56,800	11.059	1	FF
2,400	16	1	DD	19,200	11.059	1	FD
1,200	16	1	BB	9,600	11.059	1	FA
600	16	1	75	4,800	11.059	1	F4
2,400	16	0	EF	2,400	11.059	1	E8
1,200	16	0	DD	1,200	11.059	1	D0
600	16	0	BB	600	11.059	1	A0
300	16	0	75	300	11.059	1	40
4,800	12	1	F3	9,600	11.059	0	FD
2,400	12	1	E6	4,800	11.059	0	FA
1,200	12	1	CC	2,400	11.059	0	F4
600	12	1	98	1,200	11.059	0	E8
300	12	1	30	600	11.059	0	D0
2,400	12	0	F3	300	11.059	0	A0
1,200	12	0	E6	1,200	6	0	F3
600	12	0	CC	600	6	0	E6
300	12	0	98	300	6	0	CC
				110	6	0	72

Baud Rates Missing

Why are some baud rates missing from the table?

If you look at the table carefully, you will notice that some common baud rates are missing in certain scenarios. The reason is, certain microcontroller operating frequencies will only support specific baud rates. Just because a baud rate reload value can be calculated by the previous equations, doesn't mean that the microcontroller can accurately generate that specific baud rate. If you would like to calculate a baud rate that is not in the previous tables, or if you want to find out if a specific baud rate can be accurately generated at a specific operating frequency, follow these steps:

1. Use the appropriate equation to calculate the reload value.
2. Round off the calculated reload value to the nearest whole number.
3. Recalculate the baud rate using the rounded off reload value.
4. Calculate the percent error between the two baud rates by using the following formula:

$$error = \frac{abs(desired - calculated)}{desired} \times 100$$

5. If the percent error is less that 2%, then the rounded reload value is adequate to generate the specified baud rate. If the error is greater than 2%, this means the baud rate generated by the microcontroller would be different from the baud rate that you expect to be transmitting and there may be a loss of data in the process.

Common Questions
The intention of this section is to provide quick answers to common problems and questions when trying to set up the serial port in the MCS®-51 family. This has been compiled by Intel employees who technically support the MCS®-51 family of microcontrollers.

1. *What is the purpose of using interrupts and/or polling in serial applications?* In serial applications, it is necessary to know when data has completed transmission or has completed reception. Whenever data has completed transmission or completed reception, there is a specific bit (flag) that is set when the process has been completed. These two specific bits are located in the SCON register and determine when an interrupt will occur or when the polling sequence should be complete. The bits are RI and TI.

- RI is the receive interrupt flag. When operating in mode 0 of the UART, this bit is set by hardware when the 8th bit is received. In all other UART operating modes, the RI bit is set by hardware upon reception halfway through the stop bit. RI bit must be cleared by software at the end of the interrupt service routine or at the end of the polling sequence.
- TI is the transmit interrupt flag. This bit operates in the same manner as RI except it is valid for transmission of data, not reception. By using either interrupts or polling, it is necessary to check to see if either of the two bits are set.
- For the case of transmitting data, it is necessary to "watch" to see if the TI bit is set. A set bit has a logic level of 1 and a cleared bit has a logic level of 0. If you try to transmit more data and your previous data has not yet fully been transmitted, you will overwrite on top of it and have data corruption. Therefore, you must only transmit the next piece of data after the transmission of the current data has been completed.
- For the case of receiving data, it is necessary to watch and see if the RI bit is set. This bit serves a similar purpose as the TI bit. Upon reception of data, it is necessary to know when data has been completely received so it can be read before more data comes and overwrites the existing data in the register.

2. *How does the serial interrupt and polling work?*
A serial interrupt will occur whenever the RI or the TI bit has been set and the serial interrupts have been enabled in the IE and SCON register. When TI or RI is set, the processor will vector to location 23H. A common serial interrupt routine would be the following:

```
...
org 23h
JMP label
...
...
label: subroutine code
...
RETI
```

After the processor vectors to 23H, it will then vector off to location *label* which has a physical location defined by the assembler. *Label* is the start of your serial interrupt subroutine which should do the following:

- Find out which bit caused the interrupt RI or TI.
- Move data into or out of the SBUF register if necessary.
- Clear the corresponding bit that caused the interrupt.

The last line of your serial interrupt subroutine should be RETI. This makes the processor vector back to the next line of code to be executed before the processor was interrupted.

Polling is easier to implement than interrupt driven routines. The technique of polling is simply to continuously check a specified bit without doing anything else. When that bit changes state, the loop should end. For the case of serial transmission, a section of sample code would be the following:

```
...
JNB TI, $ ;this code will jump onto itself until TI is set
CLR TI ;clear the TI bit
...
```

For receive polling, just replace the TI in the previous code with RI. In either case, make sure that after polling has completed, clear the bit that you were polling.

3. ***When should I choose polling or interrupts?***
Polling is the simplest to use but it has a drawback; high CPU overhead. This means that while the processor is polling, it is not doing anything else, this is a waste of the CPU's time and tends to make programs slow.

Interrupts are a little more complex to use but allows the processor to do other functions. Thus, serial communication functions are executed only when needed. This makes programs run faster than programs that use polling.

Common Problems

I am viewing data on an oscilloscope and I am not seeing the data I transmitted; I see other data instead. Why?
You are not waiting for the data to be completely transmitted before you send more data out. The new data is being written on top of the old data before it exits to the serial port. See "What is the purpose of using interrupts and/or polling in serial applications" on page 6.

I am moving data into SBUF, all my registers are configured for serial communications, and nothing is being transmitted. Why?
Chances are that the timer you chose for your baud rate generator was never started or "turned on."

All of the registers are set up correctly, but when I receive data, the microcontroller never vectors to the interrupt routine. Why?
The global interrupt enable bit has not been set or the serial interrupt bit has not been set. The address of the first line of the serial interrupt routine was not at location 23H.

I am trying to transmit data and all I see on the oscilloscope is a square wave coming out of the Txd pin. Why?
The microcontroller serial port is in mode 0. In mode 0, the Txd pin outputs the shift clock (a square wave). Data is actually transmitted and received through the Rxd pin.

I am receiving data and I move it to another register and read it. The value that I am reading is not the data that I received. Why?
The data that was received was not moved out of the buffer (SBUF) fast enough before the new data arrived. Therefore, part of the old data got overwritten before you transferred it to another register. To avoid this, see "What is the purpose of using interrupts and/or polling in serial applications?" on page 6.

Sample Programs
The following programs have been designed to aid in the understanding of the general setup and transmission of serial applications.

```
;FILE: M0.ASM
;
;THIS PROGRAM TRANSMITS THE HEX VALUE AA REPETITIVELY ACROSS THE SERIAL PORT
;OF A MCS®-51 MICROCONTROLLER IN MODE 0
;
;DETAILS:
;
;MODE 0: SERIAL DATA EXITS AND ENTERS THROUGH THE RXD PIN.  THE
;TXD PIN OUTPUTS THE SHIFT CLOCK.  IN MODE 0, 8 BITS ARE TRANSMITTED/RECEIVED
;STARTING WITH THE LEAST SIGNIFICANT BIT.  THE BAUD RATE IS FIXED TO 1/12 THE
;OSCILLATOR FREQUENCY.
;
;
      ORG 00H
      JMP MAIN
MAIN: MOV SCON,#00H            ;SET UP FOR MODE 0
      CLR TI                   ;READY TO TRANSMIT
LOOP: MOV SBUF,#0AAH           ;TRANSMIT AAH
      JNB TI,$                 ;WAIT FOR END OF TRANSMISSION
      CLR TI                   ;CLEAR TRANSMIT FLAG
      JMP LOOP                 ;DO IT ALL AGAIN
      END
```

```
;FILE: M1T1.ASM
;
;THIS PROGRAM TRANSMITS THE HEX VALUE AA REPETITIVELY ACROSS THE SERIAL PORT
;OF A MCS®-51 IN MODE 1 USING TIMER 1 AT A RATE OF 1200 BAUD
;
;DETAILS:
;
;MODE 1: 10 BITS ARE TRANSMITTED THROUGH TXD OR RECEIVED THROUGH RXD WITH THE
;START BIT FIRST (0), 8 DATA BITS WITH THE LEAST SIGNIFICANT BIT FIRST, AND A
;STOP BIT (1).  ON RECEIVE, THE STOP BIT GOES INTO RB8 IN SPECIAL FUNCTION
;REGISTER SCON.  THE BAUD RATE IS VARIABLE.
;
;
      ORG 00H
      JMP MAIN
MAIN: MOV SCON,#40H            ;SET SERIAL PORT FOR MODE 1 OPERATION
      MOV TMOD,#20H            ;SET TIMER 1 TO AUTO RELOAD
      MOV TH1,#0DDH            ;LOAD RELOAD VALUE FOR 1200 BAUD AT 16MHZ
      MOV TCON,#40H            ;START TIMER 1
      CLR TI
LOOP: MOV SBUF,#0AAH           ;TRANSMIT AA HEX OUT THE TXD LINE
      JNB TI,$                 ;WAIT UNTIL TRANSMISSION COMPLETED
      CLR TI                   ;READY TO TRANSMIT ANOTHER
      JMP LOOP                 ;DO IT ALL OVER AGAIN
      END
```

```
;FILE: M2.ASM
;
;THIS PROGRAM TRANSMITS THE HEX VALUE AA REPETITEVELY ACROSS THE SERIAL PORT
;OF A MCS®-51 IN MODE 2 AT A RATE OF 1/32 THE OSCILLATOR FREQUENCY
;
;DETAILS:
;
;MODE 2:  11 BITS ARE TRANSMITTED THROUGH TXD OR RECEIVED THROUGH RXD.
;STARTING WITH A START BIT (0), 8 DATA BITS WITH THE LEAST SIGNIFICANT BIT
;FIRST, A PROGRAMMABLE 9th DATA BIT, AND A STOP BIT (1).  ON TRANSMIT, THE 9th
;DATA BIT, TB8 IN SCON, CAN BE ASSIGNED A VALUE OF 0 OR 1.  FOR EXAMPLE THE
;PARITY BIT, P FROM PSW, COULD BE MOVED INTO TB8.  ON RECEIVE, THE NINTH DATA
;BIT GOES INTO RB8 IN SCON WHILE THE STOP BIT IS IGNORED.  (THE VALIDITY OF
;THE STOP BIT CAN BE CHECKED WITH FRAMING ERROR DETECTION.  THE BAUD RATE IS
;PROGRAMMABLE TO EITHER 1/32 OR 1/64 THE OSCILLATOR FREQUENCY.  IF SMOD1 BIT
;IN THE PCON REGISTER IS 0, THEN THE BAUD RATE IS 1/64 THE OSCILLATOR
;FREQUENCY, IF SMOD1 IS 1, THE THE BAUD RATE IS 1/32 THE OSCILLATOR FREQUENCY.
;
;
;
        PCON EQU 87H

        ORG 00H
        JMP MAIN
MAIN:   MOV SCON,#80H                   ;SET UP FOR MODE 2
        MOV PCON,#80H                   ;BAUD RATE EQUALS 1/32 OSC. FREQ
        CLR TI                          ;READY TO TRANSMIT
LOOP:   MOV SBUF,#0AAH                  ;TRANSMIT AAH
        JNB TI,$                        ;WAIT FOR END OF TRANSMISSION
        CLR TI                          ;READY TO TRANSMIT
        JMP LOOP                        ;DO IT ALL AGAIN
        END

;FILE: M3T2.ASM
;
;THIS PROGRAM TRANSMITS THE HEX VALUE AA REPETITIVELY ACROSS THE SERIAL PORT
;OF A MCS®-51 IN MODE 3 USING TIMER 2 AS A BAUD RATE GENERATOR TO GENERATE A
;BAUD RATE OF 2400 BAUD AT 16MHZ WITH A PARITY BIT
;
;DETAILS:
;
;MODE 3:  11 BITS ARE TRANSMITTED THROUGH TXD OR RECEIVED THROUGH RXD
;TRANSMISSION STARTS WITH A START BIT (0), EIGHT DATA BITS WITH THE LEAST
;SIGNIFICANT BIT FIRST, A PROGRAMMABLE 9TH DATA BIT, AND A STOP BIT (1). MODE
;3 IS THE SAME AS MODE 2 EXCEPT THAT MODE 3 HAS A VARIABLE BAUD RATE
;
;
        RCAP2H EQU 0CBH
        RCAP2L EQU 0CAH
        T2CON  EQU 0C8H

        ORG 00H
        JMP MAIN
MAIN:   MOV SCON,#0C0H      ;SET UP FOR SERIAL MODE 3
        MOV RCAP2H,#0FFH    ;LOAD HIGH BYTE TO GENERATE 2400 BAUD AT 16MHZ
        MOV RCAP2L,#30H     ;LOAD LOW BYTE TO GENERATE 2400 BAUD AT 16MHZ
        MOV T2CON,#14       ;TIMER 2 BAUD RATE GENERATOR AND START TIMER
        MOV A,#0AAH         ;PUT THE VALUE TO BE TRANSMITTED IN THE ACC
        MOV C,P             ;PARITY INFORMATION TO CARRY FLAG
        MOV TB8,C           ;PARITY INFO FROM CARRY TO PROGRAMMABLE BIT *
;                            *NOTE: THE CONTENTS OF THE CARRY FLAG IN THE
;                                   PSW MAY BE ALTERED
        CLR TI              ;READY TO TRANSMIT
LOOP:   MOV SBUF,A          ;TRANSMIT AAH
        JNB TI,$            ;WAIT UNTIL DONE TRANSMITTING
        CLR TI              ;READY TO TRANSMIT
        JMP LOOP            ;DO IT ALL OVER AGAIN
        END
```

```
;
;THIS PROGRAM RECEIVES A VALUE ENTERING INTO THE SERIAL PORT PIN RXD AND PUTS
;THE DATA OUT TO PORT 1.
;
;DETAILS:
;
;THE PROGRAM IS DESIGNED TO BE IN A CONTINUOUS NEVER ENDING LOOP UNTIL A BYTE
;OF DATA HAS BEEN COMPLETELY RECEIVED.  THE LOOP IS EXITED BECAUSE OF THE
;OCCURANCE OF A SERIAL INTERRUPT.  AFTER THE INTERRUPT HAS BEEN SERVICED, THE
;PROGRAM GOES BACK INTO IT'S ENDLESS LOOP UNTIL ANOTHER INTERRUPT OCCURS
;
;
      PCON EQU 87H              ;DEFINE REGISTER LOCATION
;
      ORG 00H
      JMP MAIN

      ORG 023H                 ;STARTING ADDRESS OF SERIAL INTERRUPT
      JMP SERIAL_INT
;
MAIN: MOV SCON, #50H           ;SET UP SERIAL PORT FOR MODE 1 WITH RECEIVE
                               ;ENABLED
      MOV TMOD, #20H           ;SET UP TIMER 1 AS AUTO-RELOAD 8-BIT TIMER
      MOV TH1, #0DDH           ;BAUD RATE EQUALS 2400 BAUD AT 16Mhz
      MOV PCON, #80H           ;SET THE DOUBLE BAUD RATE BIT
      MOV IE, #90H             ;ENABLE THE SERIAL PORT & GLOBAL INTERRUPT BITS
      MOV TCON, #40H           ;START TIMER 1
      CLR RI                   ;ENSURE THAT THE RECEIVE INTERRUPT FLAG IS
                               ;CLEAR
LOOP:  JMP LOOP                ;ENDLESS LOOP (UNLESS INTERRUPT OCCURS)
;
SERIAL_INT:                    ;SERIAL INTERRUPT ROUTINE
      CLR RI                   ;CLEAR THE RI BIT (SINCE WE KNOW THAT WAS THE
                               ;BIT THAT CAUSED THE INTERRUPT)
      MOV P1, SBUF             ;MOVE THE RECEIVED DATA OUT TO PORT 1
      RETI                     ;EXIT THE SERIAL INTERRUPT ROUTINE
      END
```