

code which does a simple echo to exercise the functions. The flowchart of the serial interrupt is in Figure 20.13. Figure 20.18 shows a header file that can be included to define the registers with the addressing of Figure 20.15.

Handshake Flags

In UART hardware, usually two or three flags are provided to help in interfacing.

1. The *transmit buffer empty* flag alerts you that an additional character can be loaded to go out.

2. The *transmitter empty* flag lets you know that the last character is all shifted out; you can know the message has left your computer, and you can begin looking for a reply if it is expected.
3. The *receive buffer full* flag lets you know that there is a completed character to pick up.
4. In some cases a fourth flag, *overrun error*, lets you know that you have let the incoming character buffer get too full and a character has been lost.

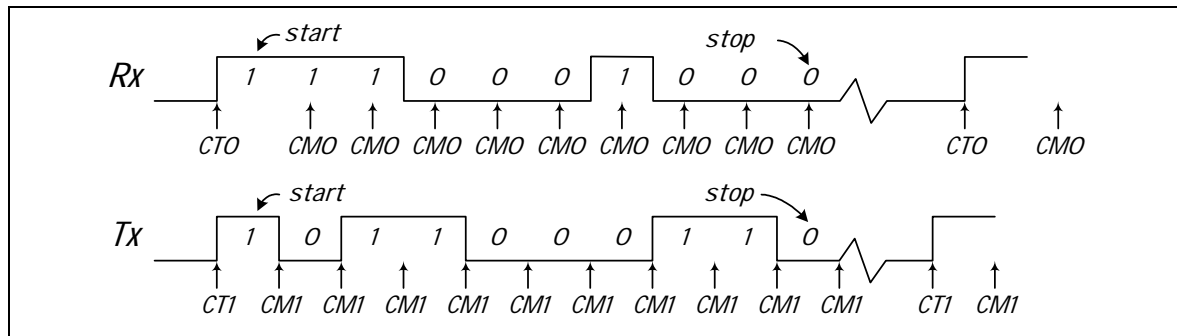


Figure 20.19: PCA Capture & Compare Serial Waveform Processing

Software-only “UARTs”

Communicating *without* using a UART saves hardware, but it can be demanding of processor time. Avoiding serial interface hardware makes sense only for low-cost applications that are not making heavy demands on the processor; otherwise, the processor will be tied up in fairly rapid, time-critical activities. Use this approach only when you must minimize hardware cost and still have a serial interface.

If you are communicating only between nearby devices, consider generating a separately clocked serial protocol like SPI or I²C. Both protocols are compatible with standard 5V ports. Since microcontroller port pins put out only logic levels, for RS-232 you would need a driver chip, although you could use the protocol with TTL levels between two agreeing devices.

Next come two examples of software-only solutions. One uses a PCA with one capture and two compare registers. The other uses only a single timer and one external interrupt.

Capture & Compare Software UART

Incoming Characters For *incoming* characters (Rx in Figure 20.19) use the trigger pin of a capture

register (or, if not available, an external interrupt). If you review Figure 20.2 on page 214, you will notice that a *start* bit, on the TTL-level side of the driver, begins with a positive-going edge. With a PCA Capture Registers (CT0 in Figure 20.19), select a positive-edge-triggered mode (standard interrupts trigger only on a *negative* edge). The interrupt function triggered by the beginning of the start bit can set a compare register value (CM0 in Figure 20.19 or in the non-PCA case set a timer delay value) for the first bit sample time. Make this delay 1½ bit-times from the *start* of the start bit so the first data sample will fall in the *middle* of the first bit. For 9600 Baud the bit duration is 104µS, so you need a 156µS delay. Then set each subsequent compare value (or timer delay) for one bit time so successive interrupts (and incoming data bit samples) fall at the middle of each bit. The same pin which triggers the capture (or external interrupt) can also be the data-in pin; in addition to being a trigger pin it can be used as a data-in pin. You get double use of that pin!

Outgoing Characters It is easiest to transmit *outgoing* bits by a second timed interrupt. Using a second compare register, CM1 in Figure 20.19 (or, in the

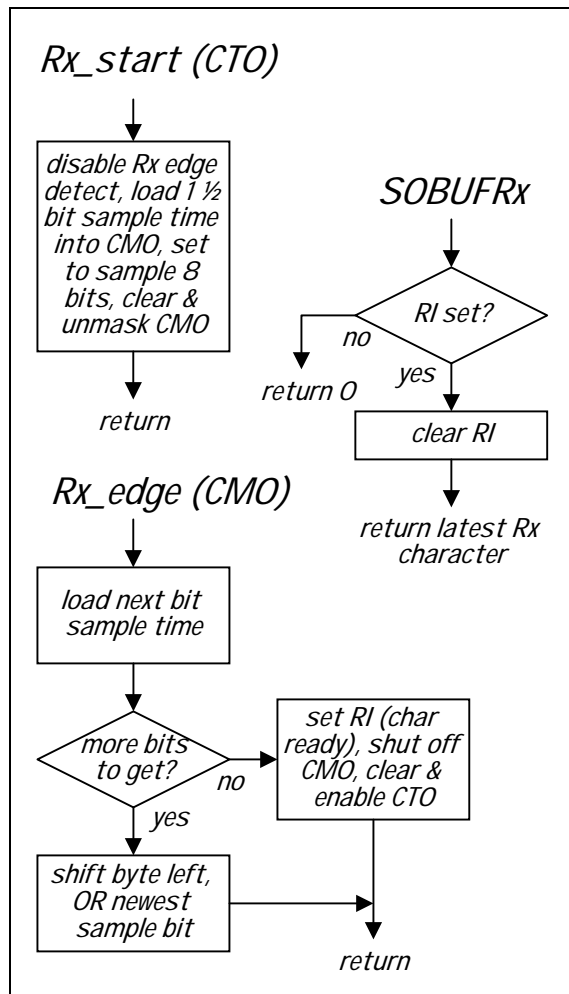


Figure 20.20: Rx Function Flowcharts

non-PCA case, a second timer) allows you to simultaneously send and receive bits keyed to separate time intervals and interrupts.

For the program shown in Figure 20.21 and Figure 20.22, two compare registers and one capture register can manage all the serial transmission.

For capturing incoming characters (which can begin at any time), consider the flowchart of Figure 20.20. As a start bit appears, the positive edge causes the capture of the timer value (*CT0*; *Rx_start*) and causes an interrupt. In that interrupt routine you add $1\frac{1}{2}$ bit times to the captured time to get the load value for a second compare register. The subsequent match of the timer value with the compare register causes an interrupt (*CM0*; *Rx_int*) that samples the first incoming data bit. At that time compute the next sample time by adding one bit time to the current count.

```

#include <reg552.h>
//Baud divide for 9600Baud
#define BDdiv 95
typedef unsigned char uchar;
typedef unsigned uint;
//public functions & flags
sbit Rxport=0x90;//P1.0 also CT0 pin
sbit Txport=0x92;//P1.2 also unused CT2
//buffers--declared here
uchar rbuf,tbuf;
bit tempty,txempty,rbempty;
//private functions, variables & flags
//union just to avoid 2-byte processing for loads
typedef union{uint in;struct{uchar hi,lo;}byt;}split;
static split reload0;//next compare value of tmr2
static split reload1;
static split reload2;
static uchar rxcnt,txcnt;//bytes left to send
static uchar Rxbyte,Txbyte;//data going in or out
void Rx_start(void)interrupt 6 using 2{//CT0
//capture0 interrupt senses Rx start bit edge
//programmed to capture on rising edge
    ECT0=0;//mask CT0 interrupt(until end of character)
    reload0.in=(CTL0|CTH0*256)+BDdiv+BDdiv/2;
    CMH0=reload0.byt.hi;CML0=reload0.byt.lo;
    rxcnt=9;//initialize to sense 8 bytes of data
    CMIO=0;//clear previous CMO interrupt flags
    ECMO=1;//unmask CMO to time first sample
}
void Rx_edge(void) interrupt 11 using 2{//CM0
//compare0 interrupt samples incoming character
    reload0.in+=BDdiv;
    CMH0=reload0.byt.hi;CML0=reload0.byt.lo;
    if(--rxcnt>0){//bits still coming
        Rxbyte>>=1;
        if(Rxport==1)Rxbyte|=0x80;
    }
    else{//all bits in
        if(rbempty==0){//full--discard
            else{
                rbuf=Rbyte;
                rbempty=0;//no longer empty
            }
        }
        ECMO=0;//mask compare interrupts
        CTIO=0;//clear old capture interrupts
        ECT0=1;//allow capture (Rxstart) interrupt
    }
    CMIO=0;//clear CMO interrupt flag
}
}

```

Figure 20.21: Driver Software (no UART; 8xC552): 1

When the software chooses, it begins transmission (*xmitstart* in Figure 20.22). It sends the first bit, captures the current timer value (*CTI*), adds the bit