Debugging embedded systems: using a serial condition monitor to overcome limited diagnostic access

STUART R BALL

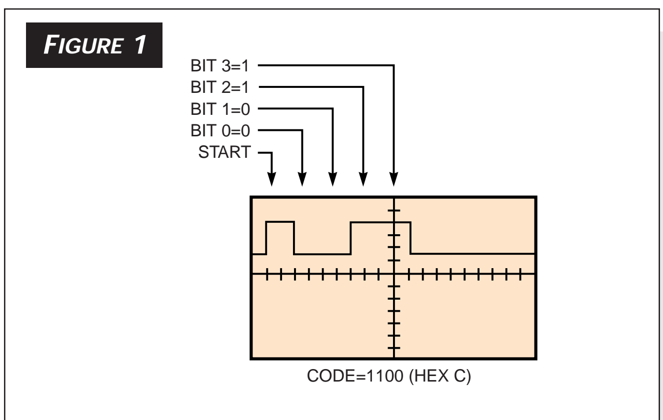
If your μC-based design is short on pins, you can perform diagnostics via only one pin by implementing a serial condition monitor.

If your μC-based design has only one pin free for diagnostic use, you can implement a serial condition monitor to provide at least some diagnostic information over that pin. Your implementation can monitor conditions in your software and report on those conditions via serial bit transmissions, which go to an attached oscilloscope. Your software sends a start bit to trigger the scope at an appropriate time, and then sends data one bit at a time. By setting the data bits either on or off, your software provides displayable information about its own status (**Figure 1**).

A serial condition monitor provides limited information, because it can send only a few bits—typically less than a byte. If you display as many as eight bits on a scope screen, it's difficult to determine from the display which bits are ones and which bits are zeros. Also, a serial condition monitor can't generate a display as often as you might want, because the scope would have difficulty triggering. Typically, you send serial condition data each time your code goes around its background loop or every time it services a regular timer interrupt.

Still, the information from a serial condition monitor can be helpful. For example, it might include a bit to indicate when a motor is active, when one system is waiting for another, or when a receive or transmit FIFO contains enough data. However, a serial condition monitor technique isn't well suited to displaying events that come and go very quickly, because the events are too hard to see on a scope. **Listings 1** and **2** show routines for sending serial status information for an 8031 and a PIC17C42, respectively. Other routines set and reset the bits that indicate status.

Ideally, a software monitor program needs a serial port to operate, but you might not always have a serial port available. The 8031, for example, has only one serial port, and your design will probably need the port for its own use, thus excluding it from diagnostic use. Also, if your design uses a μP that doesn't have an on-chip serial port, you might not want to add a hardware UART just for debugging. However, if you have a spare pin on the μC you're using, or if you have a spare register output bit on a multichip



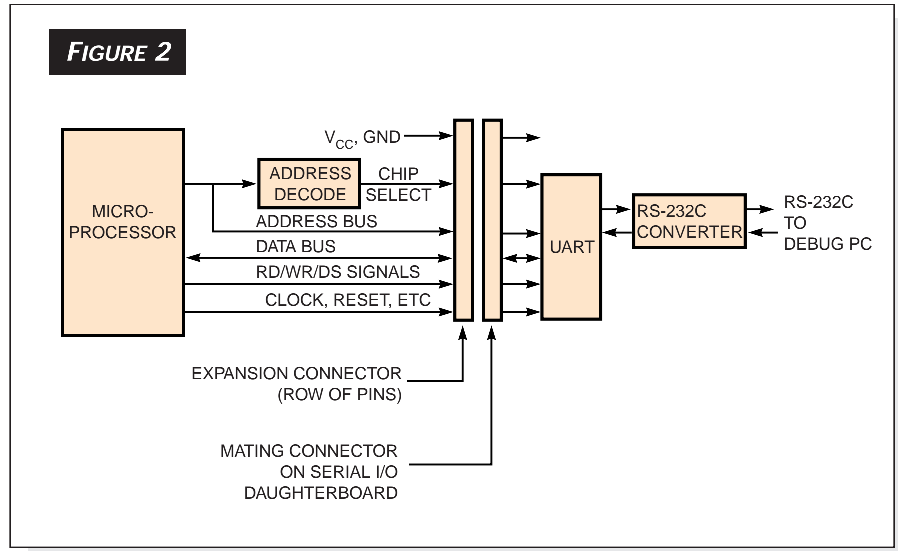
A serial condition monitor sends only a few bits of information to an oscilloscope, but each bit can represent a status condition.

This article, one of an occasional series on basic debugging techniques, is an adaptation from the book, Debugging Embedded Microprocessor Systems by Stuart R Ball. Material reproduced courtesy of Newnes, an imprint of Butterworth-Heinemann, 225 Wildwood Ave, Woburn, MA 01801-2041. For more information, check www.bh. com. To order,

processor design, you can implement a serial port in software.

An example of a software serial port, using a technique sometimes called bit banging, appears in **Listing 3**. With this code, written for the 8031, or with similar code for another processor, you can implement a post-mortem dump of memory after a failure. Your software can activate the routine after a hardware reset (manually initiated when an error occurs) or via an unused interrupt.

Listing 3's code transmits 128 bytes of memory to a PC, sending each byte of RS-232C serial data as a low-level start bit, followed by eight data bits (LSB first), and then a high-level stop bit (assuming no parity). The code converts each byte in memory to two hex digits, separating each pair of digits with a space. A carriage return and line feed go after every 16 digits. The logic levels invert before going to the RS-232C connector.



By adding an expansion connector to a design, you can connect a daughterboard containing a serial port when you need to do debugging.

The code in **Listing 3** uses one of the 8031's internal timers, but you could instead use software loops for timing. Implementing software timing loops is fairly straightforward on the 8031, but less so on more complex processors, where independent bus interfaces and execution cores make timing less predictable. On more complex processors, especially, a hardware timer is easier to use.

It's also possible to implement a serial receive routine in software, although it might not be particularly useful. You can construct a complete monitor program around send and receive serial routines, but it would be somewhat limited. The processor can't do anything else while receiving or transmitting, and the interface is half duplex, so you can't receive and transmit at the same time. On a very fast processor, you can implement a full-duplex UART in

LISTING 1—PIC17CXX ASSEMBLY CODE FOR SERIAL STATUS OUTPUT

```
; STATUS OUTPUT TO PORT D BIT 0.
; PORT D BIT 0 MUST BE CONFIGURED AS OUTPUT.
; CODE SENDS LEAST SIGNIFICANT FOUR BITS OF
; BYTE DIAG TO PORT.
; OUTPUT SEQUENCE IS:
; SYNC BIT (1)
; DIAG BIT 0
; DIAG BIT 1
; DIAG BIT 2
; DIAG BIT 3
; ZERO
; EACH BIT TAKES ABOUT 1.6 MICROSECONDS ON A PIC17C43
; WITH A 12 MHZ CRYSTAL.
        MOVLB 1
        BSF PORTD,0         ; OUTPUT SYNC BIT (1)
        NOP
        NOP
        BTFSC DIAG,0        ; DIAG BIT 0 SET?
        GOTO MSET0          ; YES, OUTPUT A 1.
        BCF PORTD,0         ; NO, OUTPUT A 0.
        GOTO M1
MSET0:
        BSF PORTD,0
        NOP
M1:
        BTFSC DIAG,1        ; DIAG BIT 1 SET?
        GOTO MSET1          ; YES, OUTPUT A 1.
        BCF PORTD,0         ; NO, OUTPUT A 0.
        GOTO M2
MSET1:
        BSF PORTD,0
        NOP
M2:
        BTFSC DIAG,2        ; DIAG BIT 2 SET?
        GOTO MSET2          ; YES, OUTPUT A 1.
        BCF PORTD,0         ; NO, OUTPUT A 0.
        GOTO M3
MSET2:
        BSF PORTD,0
        NOP
M3:
        BTFSC DIAG,3        ; DIAG BIT 3 SET?
        GOTO MSET3          ; YES, OUTPUT A 1.
        BCF PORTD,0         ; NO, OUTPUT A 0.
        GOTO MSDON
MSET3:
        BSF PORTD,0
MSDON:
        NOP
        NOP
        BCF PORTD,0         ; TERMINATE WITH A 0.
        MOVLB 0             ; END OF DIAGNOSTIC OUTPUT.
```

LISTING 2—8051/8052 ASSEMBLY CODE FOR SERIAL STATUS OUTPUT

```
; THIS CODE FRAGMENT OUTPUTS A FOUR-BIT STATUS
; VALUE, DIAGNOSTIC, TO PORT 0 BIT 0 OF AN
; 8051/8052 PROCESSOR. OUTPUT SEQUENCE IS:
; START BIT (1)
; DIAGNOSTIC BIT 0
; DIAGNOSTIC BIT 1
; DIAGNOSTIC BIT 2
; DIAGNOSTIC BIT 3
; ZERO.
; ON AN 8051 WITH AN 8 MHZ CRYSTAL,
; EACH BIT WILL BE ABOUT 4.4 MICROSECONDS LONG.
      MOV ACC,DIAGNOSTIC
      SETB P0.0          ; OUTPUT START BIT (1)
      RRC A
      MOV P0.0,C         ; OUTPUT DIAG BIT 0
      RRC A
      MOV P0.0,C         ; OUTPUT DIAG BIT 1
      RRC A
      MOV P0.0,C         ; OUTPUT DIAG BIT 2
      RRC A
      MOV P0.0,C         ; OUTPUT DIAG BIT 3
      NOP
      NOP
      CLR P0.0           ; ALL DONE
```

LISTING 3—POST-MORTEM MEMORY DUMP ON 8031 USING SOFTWARE UART

```
; 8031 post-mortem dump, sending data at
; 9600 baud to port P1.0 using bit-banging.
; The input crystal is 11.0592 MHz, and we
; use timer T0 to generate the baud clock.
;
; All locations from 0 to 7F are sent.
; Output format is two hex digits, one space,
; cr/lf after every 16 bytes.

init:                    ; Initialize timer 0.
     mov th0,#159        ; 255-159 = 104 us = 9600 baud.
     mov tl0,#0
     mov tmod,#2         ; set timer 0 for mode 2
                         ; (8-bit reloading, free running)
     setb tr0            ; enable timer0
; Read contents of memory from 00-7F, convert to ASCII,
; and output via bit-banging serial routine.
; Uses R0, acc, DPTR.
dumploop:
     mov a,@r0
     rr a
     rr a
     rr a
     rr a
     anl a,#0fh
     mov dptr,#asctable ; convert hi nybble to ascii
     movc a,@a+dptr
     call aout           ; xmit lo nybble
     mov a,@r0
     anl a,#0fh
     mov dptr,#asctable ; convert lo nybble to ascii
     movc a,@a+dptr
     call aout           ; xmit low nybble of byte
     mov a,#' '          ; space after digit pair
     call aout
     inc r0              ; Incr memory pointer
     mov a,r0
     xrl a,#080h         ; all done?
     jz done             ; yes, exit.
     mov a,r0            ; no, check for EOL
     anl a,#0fh          ; EOL (every 16 bytes)?
     jnz dumploop        ; no, keep looping.
     mov a,#0dh          ; yes, send cr, lf every 16 bytes
     call aout
     mov a,#0ah
     call aout
     jmp dumploop
done: jmp done
asctable:                ; hex-to-ascii table
          db '0123456789ABCDEF'
     aout:                          ; sends contents of accum to
                                    ; port 1.0 by bit-banging.
                                    ; uses no other registers.
     startbit:
          jnb tf0,startbit          ; loop until overflow
          clr tf0
          clr p1.0
     bit0:
          jnb tf0,bit0
          clr tf0
          rrc a
          mov p1.0,c
     bit1:
          jnb tf0,bit1
          clr tf0
          rrc a
          mov p1.0,c
     bit2:
          jnb tf0,bit2
          clr tf0
          rrc a
          mov p1.0,c
     bit3:
          jnb tf0,bit3
          clr tf0
          rrc a
          mov p1.0,c
     bit4:
          jnb tf0,bit4
          clr tf0
          rrc a
          mov p1.0,c
     bit5:
          jnb tf0,bit5
          clr tf0
          rrc a
          mov p1.0,c
     bit6:
          jnb tf0,bit6
          clr tf0
          rrc a
          mov p1.0,c
     bit7:
          jnb tf0,bit7
          clr tf0
          rrc a
          mov p1.0,c
     stopbit:
          jnb tf0,stopbit
          clr tf0
          nop
          setb p1.0
          ret
```

LISTING 4—SERIAL RECEIVE ROUTINE FOR 8031

```
; swuartin is the receive process. It is activated
; by an interrupt on INT0, (P3.2) which is the
; leading edge of the start bit for the incoming
; character.
; The code sets up timer 0 as an 8-bit reloading
; timer, sampling approximately in the middle of the
; bit. This code expects rx data to be 8N1.
; Code assumes a crystal frequency of 11.0592 MHz.
; Not shown is a routine, ERROR, which must handle
; bad start bits and any other errors that may occur
; in the serial data stream. Could be left out
; for a debugger application, if the start bit
; isn't verified at the first timer rollover.
swuartin:
                         ; Initialize timer 0.
     clr tr0             ; Stop Timer0 in case it's running
     mov th0,#160        ; 255-160 = 104 us = 9600 baud.
     mov tl0,#210        ; Starting at 210 will put the
                         ; first sample about in the
                         ; middle of the start bit.
                         ; Subsequent samples will be
                         ; spaced 1 bit time apart.
     mov tmod,#2         ; Set timer 0 for mode 2
                         ; (8-bit reloading, free running)
     clr tf0             ; Clear any pending rollover indication
     setb tr0            ; Enable timer0
rxstart:
     jnb tf0,rxstart     ; Wait for middle of bit.
     jb p3.2,error       ; If P3.2 is set, then we had
                         ; an invalid start bit.
     clr tf0
     mov a,#0            ; Clear byte accumulator
rxd0:
     jnb tf0,rxd0        ; Wait for middle of bit.
     clr tf0
     mov c,p3.2          ; Get the incoming bit,
     rrc a               ; rotate into the byte accumulator
rxd1:
     jnb tf0,rxd1        ; Wait for middle of bit.
     clr tf0
     mov c,p3.2          ; Get the incoming bit,
     rrc a               ; rotate into the byte accumulator
rxd2:
     jnb tf0,rxd2        ; Wait for middle of bit.
     clr tf0
     mov c,p3.2          ; Get the incoming bit,
     rrc a               ; rotate into the byte accumulator
rxd3:
     jnb tf0,rxd3        ; Wait for middle of bit.
     clr tf0
     mov c,p3.2          ; Get the incoming bit,
     rrc a               ; rotate into the byte accumulator
rxd4:
     jnb tf0,rxd4        ; Wait for middle of bit.
     clr tf0
     mov c,p3.2          ; Get the incoming bit,
     rrc a               ; rotate into the byte accumulator
rxd5:
     jnb tf0,rxd5        ; Wait for middle of bit.
     clr tf0
     mov c,p3.2          ; Get the incoming bit,
     rrc a               ; rotate into the byte accumulator
rxd6:
     jnb tf0,rxd6        ; Wait for middle of bit.
     clr tf0
     mov c,p3.2          ; Get the incoming bit,
     rrc a               ; rotate into the byte accumulator
rxd7:
     jnb tf0,rxd7        ; Wait for middle of bit.
     clr tf0
     mov c,p3.2          ; Get the incoming bit,
     rrc a               ; rotate into the byte accumulator
rxstop:
     jnb tf0,rxstop      ; Wait for middle of bit.
     jnb p3.2, error     ; If P3.2 not set, invalid stop bit.
     clr tf0
                         ; At this point, the received character is in
                         ; the accumulator.
```

software with a regular timer interrupt that operates at 4×, 8×, or some other multiple of the baud rate. The fast interrupts really tie the processor up, though.

Even so, a software serial routine is sometimes useful. **Listing 4**'s routine, written for an 8031, functions as an interrupt service routine (ISR), with serial data connected to the 8031 INT0 interrupt pin. With INT0 programmed to be edge-sensitive, the ISR reads serial data when the leading edge of the start bit occurs. Note that if other, higher priority interrupts occur, the interrupt latency for the serial-receive code might be too long for reliable operation.

If you want a full serial interface for debugging, but you can't justify adding hardware to every production board just in case it's needed, consider connecting processor data-bus and control lines to a pin header. You can then build a single serial interface daughterboard containing a UART and an RS-232C interface (**Figure 2**). The serial interface board connects to the pin header, and you plug it in only when you need to debug a problem.

If you have a spare interrupt available for your serial interface, you can leave a ROM-based debugger resident in your program (but beware of license fees!) and activate it when a character arrives via the serial interface. If you pull the interrupt to the inactive state on the processor board, the software won't respond to an interrupt on that line unless you have the serial board installed. Another means of testing for the serial board's presence is with a bit on the processor board, also connected to the header connector, that you can test with your software. You pull up the bit on the processor board and ground it on the serial board. Regardless of how you test for the serial board's presence, you don't want the software to execute your debugger code unless the board is installed.

e

The software listings in this article are available on EDN's Web site: www.ednmag.com. At the registered-user area, provide the required information, then go into the Software Center to download 12desp3.

References

1. Ball, Stuart R, "Debugging embedded systems: using a trace buffer to see what went wrong," *EDN*, April 9, 1998, pg 161.

2. Ball, Stuart R, "Debugging embedded systems: using hardware tricks to trace program flow," *EDN*, April 23, 1998, pg 163.