

Debugging embedded systems: using hardware tricks to trace program flow

STUART R BALL

A trace buffer is a valuable debugging aid. It provides a history of your program's behavior by recording explanatory "action codes" whenever your program reaches certain key points in its execution (**Reference 1**).

In some small μ C-based designs, however, you might not have enough memory to implement a trace buffer. Likewise, your system may not have a serial port that could aid debugging by transferring data back and forth between the system and a PC. You can still track your program's execution flow in such a minimal system, however. One way is by writing data to an unused memory or I/O location and capturing it on a logic analyzer in state mode. The analyzer's memory, in effect, serves as a trace buffer.

An illustration of tracing with 80188-based hardware appears in **Figure 1**. A 74ACT138 3-to-8-line decoder decodes the low order address from a 74ACT373 latch. PCS2 and WR (from the 80188) enable the 74ACT138. On the 80188, the PCSx lines are active for 128 contiguous addresses, and they may map into either memory or I/O space.

For purposes of illustration, assume that PCS2 is active (low) for all I/O addresses in the range 100 to 17F hex, or 512 to 639 decimal. As shown, the circuit uses Y0, Y1, and Y2, leaving the remaining lines free. Y7 provides a signal called -TRACE WR that doesn't connect to any system hardware, but can serve as a strobe to a logic analyzer's clock input.

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, call 1-800-366-2665.

If your embedded system is too small to provide even a trace buffer or a serial port as debugging aids, you can use a variety of hardware tricks to observe your program's activity.

Because the 74ACT138 decodes lines A3 through A5, Y7 is active at addresses 138 to 13F hex. If the software writes action codes to address 138, and if Y7 (-TRACE WR) is connected to a logic analyzer's clock input (**Figure 2**), the logic

analyzer can pick up the codes and store them.

Time tags and other data

Of course, you might want to store more than action codes in some cases, just as you might if you had a trace buffer in your embedded system's memory. Time tags, for example, can provide valuable information for debugging.

TABLE 1—ADDRESS DECODING IN FIGURE 1

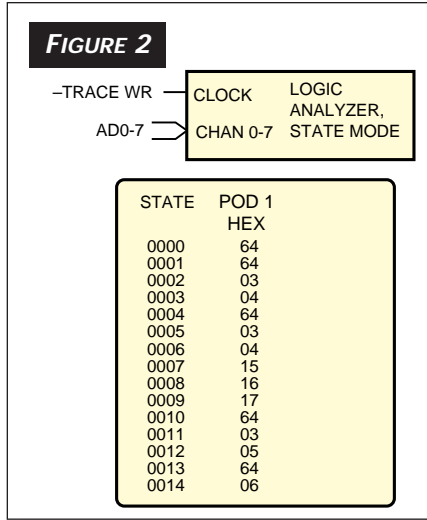
74ACT138 output	Active at addresses:
Y0 (pin 15)	100-107, 140-147, 180-187, and 1C0-1C7
Y1 (pin 14)	108-10F, 148-14F, 188-18F, and 1C8-1CF
Y2 (pin 13)	110-117, 150-157, 190-197, and 1D0-1D7
Y3 (pin 12)	118-11F, 158-15F, 198-19F, and 1D8-1DF
Y4 (pin 11)	120-127, 160-167, 1A0-1A7, and 1E0-1E7
Y5 (pin 10)	128-12F, 168-16F, 1A8-1AF, and 1E8-1EF
Y6 (pin 9)	130-137, 170-177, 1B0-1B7, and 1F0-1F7
Y7 (pin 7)	138-13F, 178-17F, 1B8-1BF, and 1F8-1FF

DEBUGGING EMBEDDED SYSTEMS

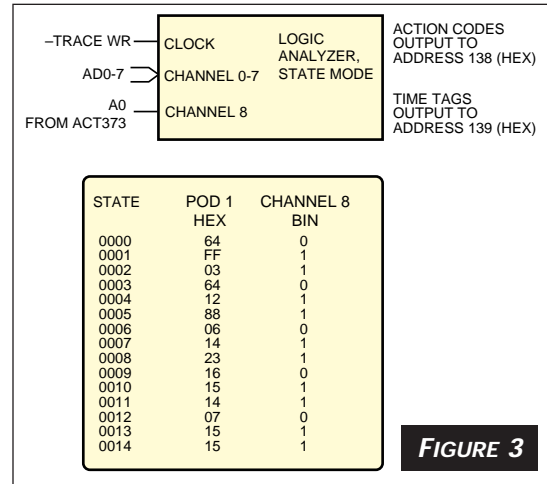
Fortunately, most modern logic analyzers can time stamp captured states, so your software doesn't have to. And you, in turn, don't have to keep track of trace-buffer memory pointers, which you would need to do if you were mixing action codes that stand alone and action codes that have time tags associated with them. Of course, you still need a way to dis-

tinguish action codes and data if the data contains anything other than time information.

A slight change in the logic-analyzer connection can make it easy to separate action codes and data. To see how, note that in **Figure 1**, the ACT138 decodes only A3, A4, and A5 (a partial decode) of the address bus, so each line is active for more than one address. For example, Y0 (pin 15) is active for addresses 100 to 107, and again from 140 to 147, 180 to 187, and 1C0 to 1C7. **Table 1** shows the address ranges for which the ACT138 decoder's outputs become active. Remember that, on the 80188, PCSx lines are active for 128 contiguous addresses.



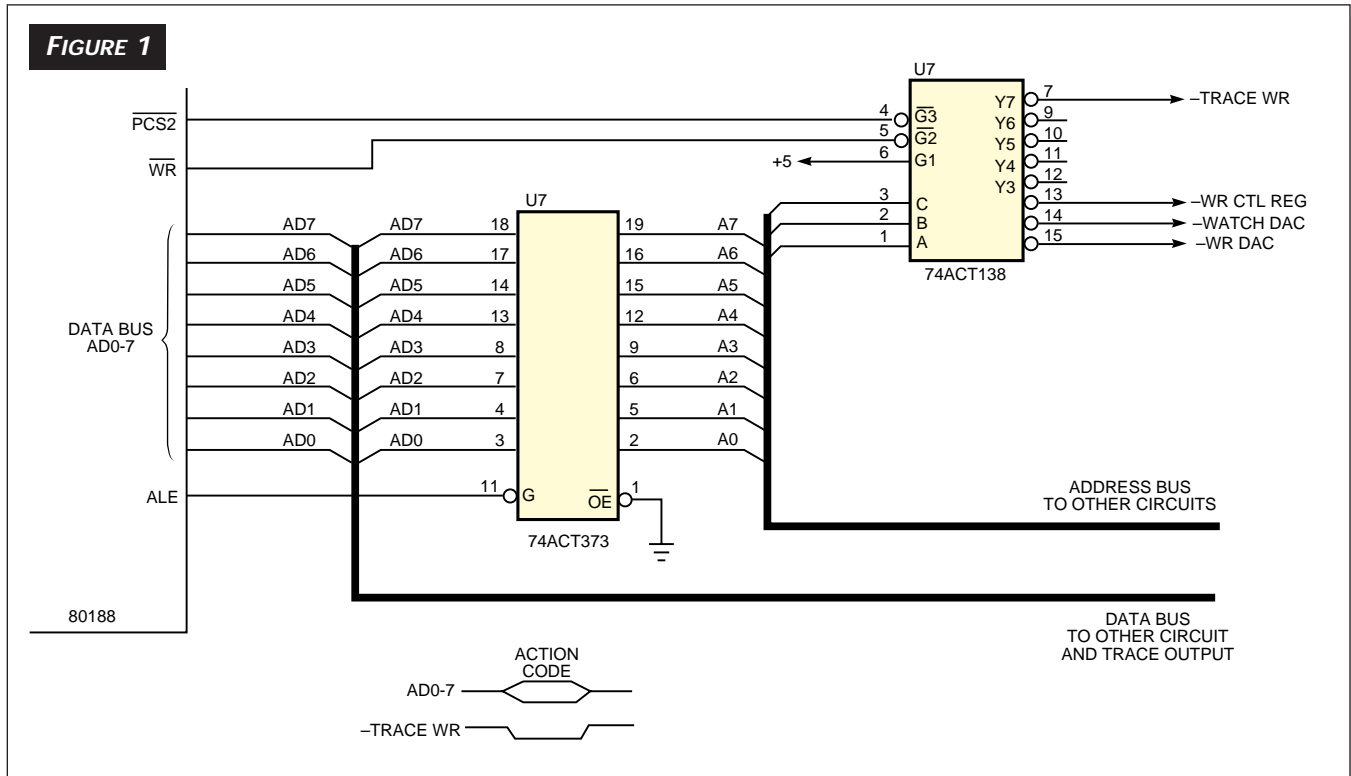
A logic analyzer can store "action codes" corresponding to key points in your program, in effect creating a trace



With appropriate addresses decoding, you can make a logic analyzer store not only action codes that trace program flow, but also data corresponding to the action codes.

Figure 3 shows how you can connect a logic analyzer to take advantage of this situation. As in **Figure 2**, Y7

With appropriate addresses decoding, you can make a logic analyzer store not only action codes that trace program flow, but also data corresponding to the action codes.



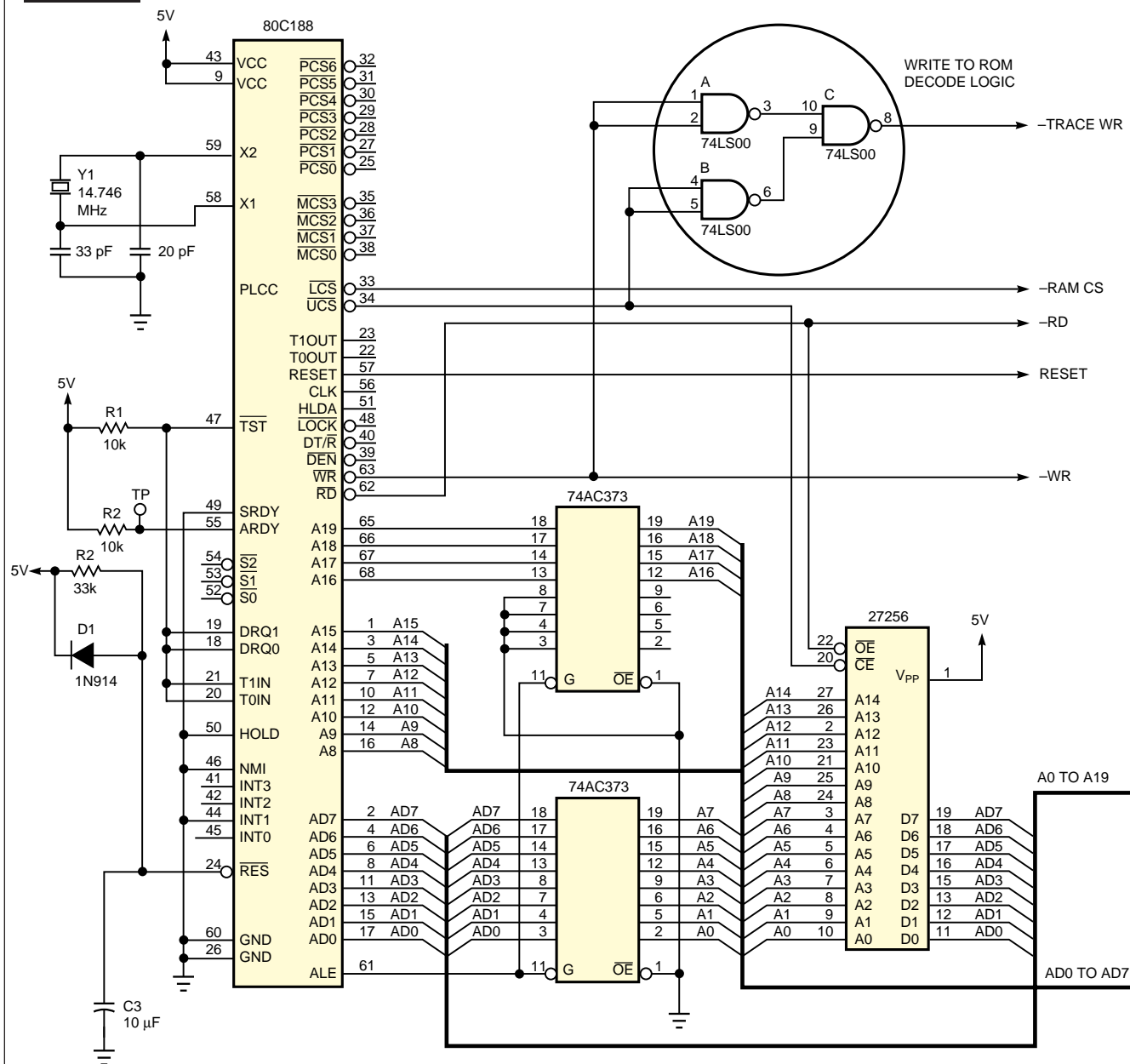
By writing to unused addresses, your program can make a logic-analyzer store codes that trace the program's execution flow. Here, a 74ACT138 3-to-8-line decoder generates a signal that clocks the analyzer whenever the program writes to a particular address.

serves as a diagnostic strobe, connected to the logic analyzer's clock input. Data capture still occurs on the analyzer's pod 1. However, **Figure 3** adds A0 from the 80188 address latch to the logic analyzer, connecting it to channel 8. Now, when you want to write an action code, you write it to address 138 (hex) as before. But, if you want to write additional diagnostic data, you write it to address 139 (hex). **Figure 3** includes an example display.

With the connections of **Figure 3**, writing to address 138

toggles the -TRACE WR signal, clocking data into the analyzer as with the connections of **Figure 2**. Writing to address 139 also toggles the -TRACE WR signal and clocks data into the analyzer, but with the A0 line high. As a result, you can tell the difference between an action code and an action code's accompanying data. You can even attach more than two bytes of data with selected action codes, because you can always find the action code corresponding to a stream of data bytes. Just look backward in the analyzer buffer for the

FIGURE 4



Decoding write operations to ROM space is an alternative way of strobing a logic analyzer to store trace information.

DEBUGGING EMBEDDED SYSTEMS

an ACK signal to terminate the cycle, make sure the PROM decoding logic generates the ACK on both reads and writes.

On some microcontrollers, such as the 8031, you can't write to ROM space. However, if you're not using all the ROM space, you can accomplish your objective by reading from ROM. In **Figure 5**, the 27256 EPROM requires 32k of the 64k address space, so you can decode any read from the upper 32k ($A_{15}=1$) as a trace output. The circled gates in **Figure 5** provide logic for generating a logic-analyzer write strobe from a read-from-ROM operation. **Listing 1** shows the necessary program code. Note that the logic analyzer takes trace data from the low order *address* lines.

Whatever scheme you use to generate trace-output signals, you can simplify hookup by adding a connector for these signals to your board's design. A 10-pin inline header works well with 8-bit processors; a dual-row header is good for 16-bit designs. Eight (or 16) pins connect to the data lines, one connects the write strobe, and one serves as ground. You can also add an edge connector on one side of the board. If board space is at a premium, you can add a row of pads and access the test points with a fixture containing spring-loaded "pogo pins" of the type used in bed-of-nails testers.

Another debugging aid is the use of LEDs as status indicators (**Figure 6**). The LEDs typically connect to a register out-

put (in the figure, pin 19 of a 74AC374 flip-flop) or a port pin. When connecting an LED to some μ P's, you need to add a drive transistor, because the μ P's output sink current is insufficient for adequate LED brightness. In the figure, an LED connects to pin 8 of an 8031 through a MOSFET. If you use a bipolar LED, as shown connected to pins 2 and 5 of the 74AC374 in the figure, the LED can glow either red or green to indicate different status conditions:

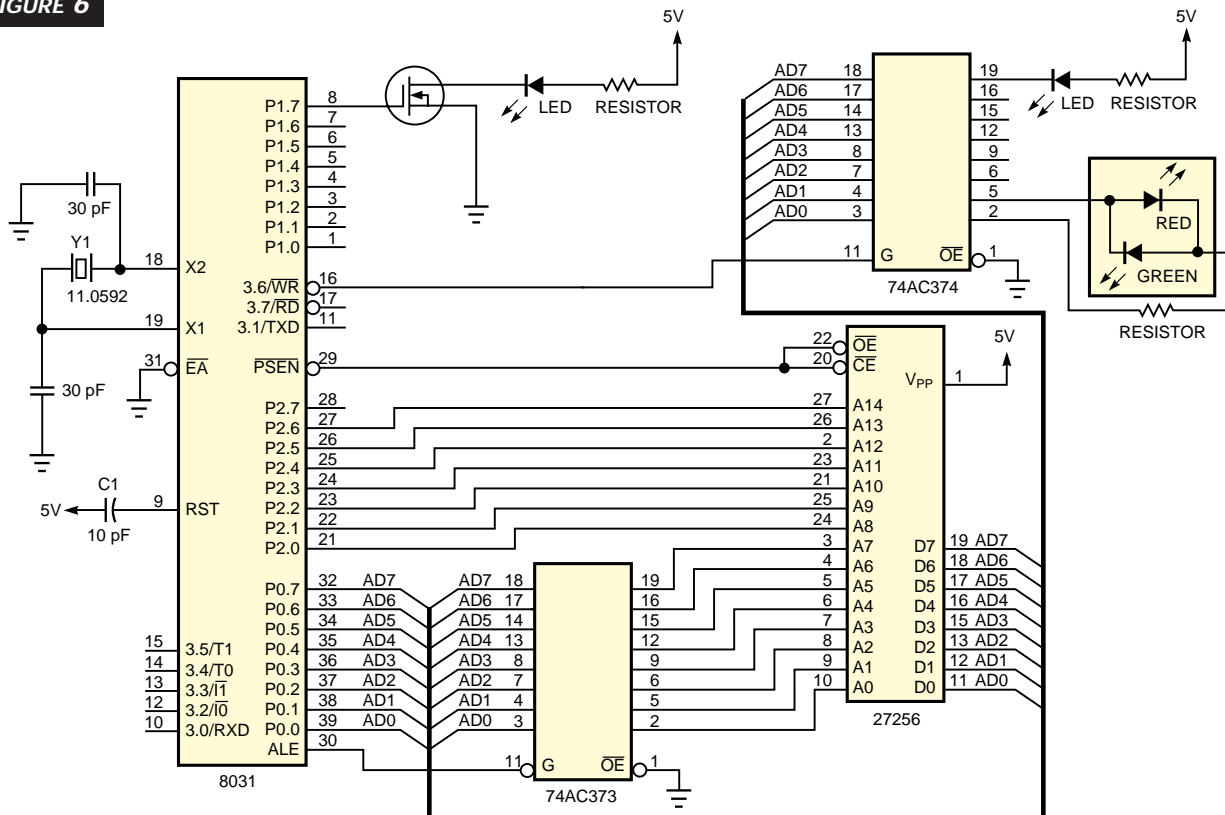
Pin 5	Pin 2	LED color
0	0	off
0	1	green
1	0	red
1	1	off

If the pins toggle quickly between the 10 and 01 states, the LED can also display an amber color.

A single software-controlled LED on a board can serve as a good/bad indicator—useful for indicating which board in a malfunctioning system needs replacing. Often, though, you'll want a more detailed idea of what went wrong. One way to supply this additional information is to flash an error code on an LED. **Listing 2** shows how to do this on an 8031, although the technique will work for any processor.

Listing 2's code turns an LED on for about one second as a start indicator, then flashes the LED quickly to indicate the

FIGURE 6



LEDs added to your embedded system can serve as status indicators.

DEBUGGING EMBEDDED SYSTEMS

most significant error-code digit and then more slowly to indicate the least significant digit. To read a two-digit error code, look for the start indication, count the slow flashes, and then count the fast flashes. (For ease in counting and remembering the flashes, implement error codes in which neither digit has a value greater than five.) **Listing 2's** 8031 code is suitable as a display-and-hang error handler. If you want the processor to keep running, servicing interrupts, or communicating with a display, you can implement the LED flashes as part of a regular tick interrupt service routine. **EDN**

Author's biography

Stuart Ball, PE, is an electrical engineer who has spent the last 16 years designing digital, analog, and embedded- μ P systems. He is the author of two books, [Embedded Microprocessor Systems](#), [Real World Design and Debugging Embedded Microprocessor Systems](#), both published by Newnes (Woburn, MA).

He is currently employed at Organon-Teknika (Oklahoma City, OK), a manufacturer of medical instruments.

Reference

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

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 [09msdesp](#)

VOTE

Please use the Information Retrieval Service card to rate this article (circle one):

High Interest	Medium Interest	Low Interest
586	587	588

LISTING 2—8031 CODE FOR FLASHING A TWO-DIGIT ERROR CODE

```

; 8031 error-code flasher.
; Code to flash is in R0.
; Flashes one long (1 sec) ON time,
; then MSB at about .4 sec/digit, then
; LSB at about .8 sec/digit.
; So a code of 53 results in a
; 1 sec ON time, then 5 quick flashes
; then 3 slow flashes.
; A '1' at P1.0 turns the LED ON, a '0'
; turns it off.
; The input crystal is 11.0592 MHz, and we
; use timer T0 to generate the flash rate.
; R1 and R2 are used as a temp storage locations.
;
flash:
; first, break the two-digit code down into
; a msb count (in R2) and an LSB count (in R1).
mov a,r0
anl a,#0fh
mov r1,a
mov a,r0
rr a ; get msb to lsb
rr a
rr a
rr a
anl a,#0fh ; strip off high bits.
mov r2,a
; now r2 = upper 4 bits of orig value,
; r1 contains lower 4 bits.
; to output a 1 sec ON time, we need 14 iterations
; of 65536 counts.
clr p1.0
mov a,#14
startoff:
call delay
dec a
jnz startoff
setb p1.0 ; turn on LED.
mov a,#14 ; 14 cycles
startset:
call delay ; delay one count of FFFF
dec a
jnz startset ; loop until a=0
; now send the MS count. This is done by
; turning the LED on and off for 6 intervals,
; decrementing R0, and repeating until R0
; goes to 0.
; Since the start bit was on, do off first.
msoff:
clr p1.0
mov a,#3 ; Flash on time = 3
msofflp:
call delay
dec a
jnz msofflp
; off time done, turn on LED and do ON time.
setb p1.0
mov a,#3 ; flash off time = 3
msonlp: call delay
dec a
jnz msonlp ; loop until done.
; ON time done, now decr MS bit count,
; repeat until done.
mov a,r2
dec a
mov r2,a
jnz msloff ; loop until r0 = 0.
; now turn the LED off and delay
; to make a break between the
; fast and slow flashes.
clr p1.0 ; LED off
mov a,#3
call delay
; ms nybble done, now output ls nybble at
; slow rate (12 delays, 6 on, 6 off).
lsloff:
clr p1.0
mov a,#6 ; on time = 6.
lslofflp:
call delay
dec a
jnz lslofflp
; off time done, turn on LED and do ON time.
setb p1.0
mov a,#6 ; off time = 6.
lsonlp:
call delay
dec a
jnz lsonlp ; loop until done.
; ON time done, now decr LS bit count,
; repeat until done.
mov a,r1
dec a
mov r1,a
jnz lsloff ; loop until r1 = 0.
jmp flash ; keep outputting the code forever.
; Delay is a support routine that delays for a
; count of 65536 (71 ms). Uses T0, no regs.
delay:
clr tf0
mov th0,#0
mov tl0,#0
mov tmod,#1
setb tr0
delaywait:
jnb tf0, delaywait ; loop until timer done
clr tr0
ret

```