

EDITED BY BILL TRAVIS &amp; ANNE WATSON SWAGER

# 25-kV generator tests insulation

ŁUKASZ ŚLIWCZYŃSKI AND PRZEMYSŁAW KREHLIK, UNIVERSITY OF MINING AND METALLURGY, KRAKÓW, POLAND

To generate high voltages with proper insulation between the “hot node” and the rest of the circuitry, a car ignition coil can function in place of a high-voltage transformer. These coils have voltage ratings of approximately 20 kV, so you can use them to produce voltages around this value. Because you know the turns ratio of the coil, you can make a stable high-voltage source using a well-controlled voltage at the primary side (Figure 1).

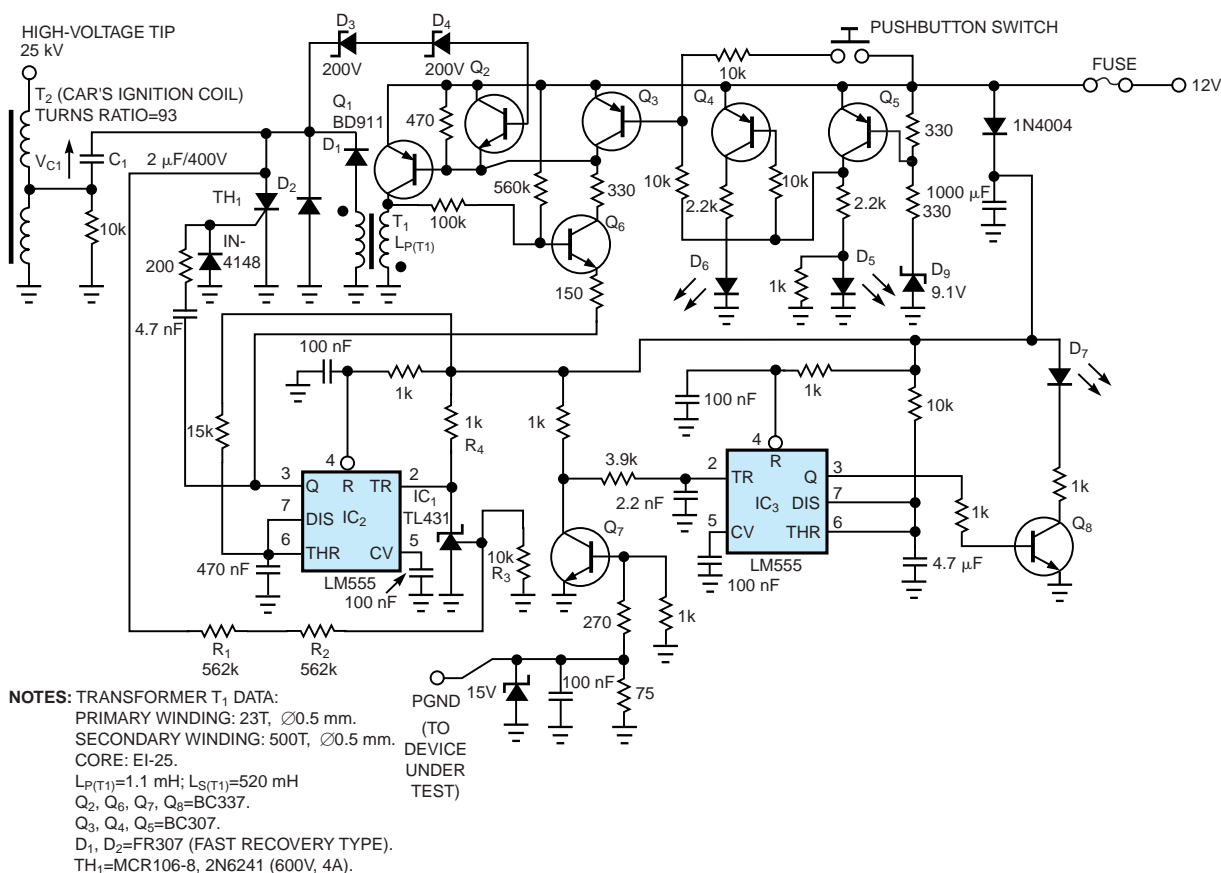
A high-voltage source is a useful device for many applications, including when it is necessary to evaluate the continuity of the dielectric coating deposited on a metal surface. If you also want to estimate the breakdown strength of the coating, this voltage source must be stable. You can easily generate the high voltage with the use of a step-up transformer, but the serious problem of proper insulation emerges. For voltages greater than a few kilovolts, specially construct-

ed transformers with the old insulation are often useful, but these devices are rather expensive and bulky.

The main part of the generator in Figure 1 consists of a free-running converter comprising  $Q_1$ ,  $Q_6$ , and the transformer,  $T_1$ . During the first part of the conversion cycle,  $Q_1$  is saturated, and energy stores in the magnetic field of  $T_1$ .  $D_1$  is reverse-biased during this time. In the second part of the cycle,  $Q_1$  is in cutoff, and the current from the secondary winding of  $T_1$  forces  $D_1$  into conduction. During this time, energy pumps into  $C_1$  through part of the ignition coil,  $T_2$ . This process allows the voltage,  $V_{C1}$ , on  $C_1$  to build gradually in a quantized manner. The value of the individual “quantum,”  $\Delta V_{C1}$ , is not constant and depends on the initial voltage,  $V_{CO}$ , which comes from the previous cycle, as follows: where

is the energy stored in the magnetic field of  $T_1$  in the first cycle

FIGURE 1



Using a car's ignition coil produces a test voltage as high as 25 kV for insulation testing.

and  $I_{C_{MAX}(Q1)}$  is the collector current of  $Q_1$  at the end of the first cycle. For the component values in **Figure 1**,  $\Delta E_C \approx 0.5$

$$\Delta V_{C1} = V_{C0} \left( \sqrt{1 + \frac{2\Delta E_C}{V_{C0}^2 \cdot C_1}} - 1 \right) \approx \frac{\Delta E_C}{V_{C0} \cdot C_1},$$

$$\Delta E_C = \frac{L_{P(T1)} \cdot I_{C_{MAX}(Q1)}^2}{2}$$

mJ, and  $I_{C_{MAX}(Q1)} \approx 1A$ .

$R_1$ ,  $R_2$ , and  $R_3$  divide down  $V_{C1}$ . When this reduced voltage reaches 2.5V, the TL431's 2.5V reference starts to sink the current through  $R_4$ , so the voltage at the trigger input of the one-shot,  $IC_2$ , rapidly decreases. An output pulse from  $IC_2$  stops the converter for about 8 msec; the emitter node of  $Q_6$  goes high, driving it into cutoff. The rising edge of  $IC_2$ 's output pulse also triggers thyristor  $TH_1$ . The thyristor connects  $C_1$ , which is charged to the appropriate voltage, directly to the primary winding of the ignition coil, and the high-voltage pulse appears at the "hot" end of the coil. A damped oscillation also starts because the ignition coil and  $C_1$  form a resonant circuit.

When a path between the "hot" end and ground exists, part of the energy from the capacitor disperses in the electric arc, and the rest returns to the capacitor through  $D_2$ . When there is no path from this end of the current to flow, almost all of the energy pumps back into  $C_1$ . This scheme provides the circuit with relatively high efficiency.

You can calculate the voltage at the "hot" side using the following formula:

where  $N_{SEC(T2)}/N_{PRI(T2)}$  is the turns ratio of the ignition coil, which equals 93 in this case. Changing the value of  $R_3$  conveniently regulates  $V_{HIGH}$ . The accuracy of this voltage is in

$$V_{HIGH} = 2.5 \left[ 1 + \frac{R_1 + R_2}{R_3} \right] \left[ 1 + \frac{N_{SEC(T2)}}{N_{PRI(T2)}} \right],$$

the range of one "quantum"  $\Delta V_{C1}$  multiplied by  $T_2$ 's turns

ratio. Thus,  $\Delta V_{C1}$  should be small to achieve good stabilization. On the other hand, a smaller value increases the time between subsequent high-voltage pulses. In this case, the accuracy estimate of the high-voltage pulse is better than 0.5% at 25 kV.

The free-running frequency of the converter depends on the time it takes to lead  $Q_1$  out of saturation (first part of the cycle) and the time when the current from the secondary winding of  $T_1$  drops to a value near zero (second part of the cycle). This circuit doesn't tightly control this frequency, which isn't a critical design parameter; the values in **Figure 1** set the frequency to approximately 6 kHz.

$Q_2$ ,  $D_3$ , and  $D_4$  prevent  $V_{C1}$  from exceeding about 400V, which protects the generator from producing excessively high voltages.  $Q_3$ ,  $Q_4$ ,  $Q_5$ , and associated circuitry allow for blocking the converter when the power supply to the circuit is too low. A too-low power-supply level may lead to an output-pulse amplitude from  $IC_2$  that is too low to trigger the thyristor, so  $V_{C1}$  may reach a very high value, limited only by the breakdown voltage of the thyristor. This breakdown voltage is the second level of protection, but you can never take too much care in circuits like these.

Two LEDs indicate the status of the power supply:  $D_5$  indicates that the level is OK, and  $D_6$ , that the power supply is too low. One-shot  $IC_3$ ,  $Q_4$ , and associated components form the source of an alarm, indicated by a flashing  $D_7$ , when the isolation breaks down or a discontinuity occurs. A simple push-button switch turns on the generator.

For the component values in **Figure 1**, the circuit generates 25-kV pulses with a repetition rate of approximately 0.2 sec. This repetition rate depends on the occurrence or lack of occurrence of the electric arc. Because the amount of energy stored in  $C_1$  is relatively low, the energy of the high-voltage pulse is also low, which is good for safety purposes. Note that it is very important and absolutely necessary to connect the part you're testing to the PGND point, because the risk of electric shock exists. (DI #2199) e

**To Vote For This Design, Circle No. 414**

## Scheme speeds access to $\mu P$ 's real-time clock

**JERZY CHRZASZCZ, WARSAW UNIVERSITY OF TECHNOLOGY, WARSAW, POLAND**

The DS5000T (Dallas Semiconductors, [www.dalsemi.com](http://www.dalsemi.com)) is an 8051-compatible processor that integrates nonvolatile memory and a real-time clock. This module has an impressive set of functional extensions and security features, which makes it particularly useful for all-in-one embedded systems.

Unfortunately, access to the real-time clock is complicated and thus inefficient. You access the on-chip real-time-clock registers serially in secondary address space by selecting the ECE2 bit in the MCON register. Instead of just moving the data, you must execute MOVX instructions with appropriate address patterns. First, a 64-bit key is necessary to open the clock, followed by a read or write of the next 64 bits of

date/time data. The access routines available from the manufacturer (example file DEMODS5T.SRC) are painfully slow: a byte read takes 106 processor cycles, a byte write takes 112 cycles, and clock opening takes 1929 cycles. Therefore, access to all real-time-clock registers (open/read or open/write sequences) lasts more than 2800 cycles.

**Listing 1** uses a different control scheme; the protocol logic resets only during system start-up, which consumes 436 cycles. Also, the **listing** linearizes the short loops used in the original procedures to open the clock and read/write the data byte. The result is that a byte read takes 51 cycles, a byte write takes 57 cycles, a whole real-time-clock read takes 926 cycles,

and a real-time-clock write takes 972 cycles. The potential drawback of this option, with respect to the original approach, is that interrupt-service routines executed during real-time-clock access must not address external data memory, because any MOVX would interfere with the clock-access protocol.

You can download **Listing 1** and other related listings from EDN's Web site, [www.ednmag.com](http://www.ednmag.com). At the registered-user area, go into the Software Center to download the file from DI-SIG, #2187. (DI #2187)

e

**To Vote For This Design, Circle No. 415**

## LISTING 1—REAL-TIME-CLOCK-ACCESS ROUTINE

```

;*****
; time-optimized RTC access routines for DS5000T
; based on DEMODSST.SRC of Dallas Semiconductor
; JCh - January 1998
;*****
SMOD5000

;*****
; Close the RTC in case it was open before
;*****
CloseRTC:
    push    MCON          ; save MCON register
    orl     MCON,#4       ; switch to CE2
    mov     DPTR,#0004    ; set up for data input
    movx    A,@DPTR       ; reset pattern detector
    call    RBYTE_        ; read 1st byte
    call    RBYTE_        ; read 2nd byte
    call    RBYTE_        ; read 3rd byte
    call    RBYTE_        ; read 4th byte
    call    RBYTE_        ; read 5th byte
    call    RBYTE_        ; read 6th byte
    call    RBYTE_        ; read 7th byte
    call    RBYTE_        ; read 8th byte
    pop     MCON          ; restore MCON register
    ret

;*****
; Read all RTC registers into predefined memory locations
;*****
ReadRTC:
    push    MCON          ; save MCON register
    orl     MCON,#4       ; switch to CE2
    mov     DPTR,#0004    ; set up for data input
    movx    A,@DPTR       ; reset pattern detector
    mov     A,#0C5H       ; load 1st byte of pattern
    call    WBYTE_        ; generate 2nd pattern byte
    cpl     A              ; generate 3rd pattern byte
    swap    A              ; generate 4th pattern byte
    call    WBYTE_        ; generate 5th pattern byte
    cpl     A              ; generate 6th pattern byte
    swap    A              ; generate 7th pattern byte
    call    WBYTE_        ; generate 8th pattern byte
    cpl     A              ; generate 9th pattern byte
    mov     DPL,#4        ; set up for data input
    call    RBYTE_        ; read a byte
    mov     RTCChr,A       ; read a byte
    call    RBYTE_        ; read a byte
    mov     RTCSec,A       ; read a byte
    call    RBYTE_        ; read a byte
    mov     RTCmin,A       ; read a byte
    call    RBYTE_        ; read a byte
    mov     RTCmon,A       ; read a byte
    call    RBYTE_        ; read a byte
    mov     RTCyrs,A       ; read a byte
    pop     MCON          ; restore MCON register
    ret

;*****
; load all RTC registers from predefined memory locations
;*****
teRTC:
    push    MCON          ; save MCON register
    orl     MCON,#4       ; switch to CE2
    mov     DPTR,#0004    ; set up for data input
    movx    A,@DPTR       ; reset pattern detector
    mov     A,#0C5H       ; load 1st byte of pattern
    call    WBYTE_        ; generate 2nd pattern byte
    cpl     A              ; generate 3rd pattern byte
    swap    A              ; generate 4th pattern byte
    call    WBYTE_        ; generate 5th pattern byte
    cpl     A              ; generate 6th pattern byte
    swap    A              ; generate 7th pattern byte
    call    WBYTE_        ; generate 8th pattern byte
    cpl     A              ; generate 9th pattern byte
    mov     A,RTCChr       ; write a byte
    call    WBYTE_        ; write a byte
    mov     A,RTCSec       ; write a byte
    call    WBYTE_        ; write a byte
    mov     A,RTCmin       ; write a byte
    call    WBYTE_        ; write a byte
    mov     A,RTCChr       ; write a byte
    call    WBYTE_        ; write a byte
    mov     A,RTCdow       ; write a byte
    call    WBYTE_        ; write a byte
    mov     A,RTCday       ; write a byte
    call    WBYTE_        ; write a byte
    mov     A,RTCyrs       ; write a byte
    call    WBYTE_        ; write a byte
    pop     MCON          ; restore MCON register
    ret

call    WBYTE_        ; write a byte
mov     A,RTCmon       ; write a byte
call    WBYTE_        ; write a byte
mov     A,RTCyrs       ; write a byte
call    WBYTE_        ; write a byte
pop     MCON          ; restore MCON register
ret

;*****
; Read one byte from the RTC and return it in ACC
;*****
RBYTE_:
    movx    A,@DPTR       ; input 1st bit
    mov     C,ACC.7        ; move it to carry
    mov     B.0,C          ; save the data bit
    movx    A,@DPTR       ; input 2nd bit
    mov     C,ACC.7        ; input 3rd bit
    mov     B.1,C          ; input 4th bit
    movx    A,@DPTR       ; input 4th bit
    mov     C,ACC.7        ; input 5th bit
    mov     B.2,C          ; input 6th bit
    movx    A,@DPTR       ; input 6th bit
    mov     C,ACC.7        ; input 7th bit
    mov     B.3,C          ; input 8th bit
    movx    A,@DPTR       ; input 8th bit
    mov     C,ACC.7        ; copy result
    mov     A,B            ; return
    ret

;*****
; Write one byte from the accumulator to the RTC
;*****
WBYTE_:
    mov     B,A            ; save the accumulator
    anl     A,#1           ; set up bit for output
    mov     DPL,A          ; set address to write bit
    movx    A,@DPTR       ; output 1st bit
    clr     A              ; output 2nd bit
    mov     C,B.1          ; output 3rd bit
    rlc     A              ; output 4th bit
    mov     DPL,A          ; output 5th bit
    movx    A,@DPTR       ; output 5th bit
    clr     A              ; output 6th bit
    mov     C,B.2          ; output 7th bit
    rlc     A              ; output 8th bit
    mov     DPL,A          ; output 9th bit
    movx    A,@DPTR       ; output 9th bit
    clr     A              ; restore the accumulator
    mov     B,A            ; return
    ret

DSEG
ORG     30H

RTCChr: DS 1             ; 00-99 0.01 seconds
RTCSec: DS 1             ; 00-59 seconds
RTCmin: DS 1             ; 00-59 minutes
RTCChr: DS 1             ; 00-23 or 01-12 hours (b7 12/24 b5
AM/PM)
RTCdow: DS 1             ; 01-07 weekday (b5 -GSC)
RTCday: DS 1             ; 01-31 day
RTCmon: DS 1             ; 01-12 month
RTCyrs: DS 1             ; 00-99 year

END

```

## Limiting amplifier is digitally programmable

**V MANOHARAN, NAVAL PHYSICAL AND OCEANOGRAPHIC LABORATORY, KOCHI, INDIA**

Amplitude limiters are necessary in many systems, such as radar and FM receivers, for which the system cannot allow the amplitude of the signal to exceed the given positive, negative, or both limits. In the circuit in **Figure 1**, amplifier IC<sub>4B</sub>'s maximum output is digitally programmable over  $\pm 2$  to  $\pm 10V$  in  $2^n$  steps, where n is the number of bits of the DAC. IC<sub>1</sub>, a precision 10V reference, provides a full-scale reference current,  $I_{REF}=V_{REF}/R_1=2$  mA, to IC<sub>2</sub>, a multiplying DAC.

IC<sub>3</sub>'s output voltage,  $V_L$ , is the sum of the product of the digital word and unipolar reference voltage and IC<sub>1</sub>'s dc offset as follows:

$$V_L = \left( \frac{V_{REF}}{R_1} \cdot \frac{N}{2^n} \cdot R_2 \right) + \frac{R_4}{R_3 + R_4} \cdot V_{REF},$$

where  $N$  can assume values of 0 to  $2^n-1$ .

When all digital inputs are set to a logic low,  $N=0$ ,

$$V_{L(MIN)} = \frac{R_4}{R_3 + R_4} \bullet V_{REF}.$$

For the values of  $R_3$ ,  $R_4$ , and  $V_{REF}$  in this example,  $V_{L(MIN)}=1V$ . When all the digital inputs are set to logic high ( $n=8$ , and  $N=255$ ).

$$\begin{aligned} V_{L(\text{MAX})} &= \frac{V_{\text{REF}}}{R_1} \cdot \frac{2^n - 1}{2^n} \cdot R_2 + \frac{R_4}{R_3 + R_4} \cdot V_{\text{REF}} \\ &= \frac{V_{\text{REF}}}{R_1} \cdot \frac{255}{256} \cdot R_2 + 1V. \end{aligned}$$

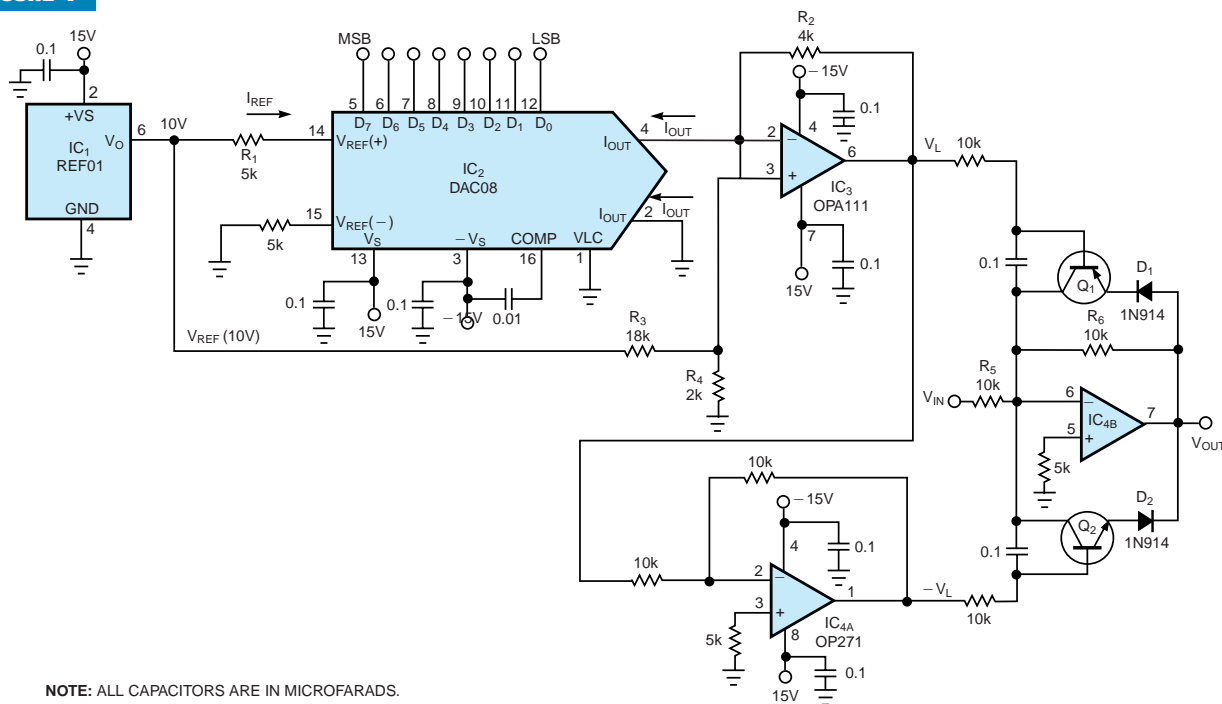
For the values of  $R_1$  and  $R_2$ ,  $V_{L(MAX)} \approx 9V$ .

Within the limiting levels, the amplifier does not modify its input signal but provides a gain of  $A_v = -R_6/R_5$ . As  $V_{OUT}$  rises above  $V_L + 1V$ —adding 1V overcomes the potential drops of the base-emitter junctions of  $Q_1$  and  $D_1$ —the base-emitter junction of  $Q_1$  becomes forward-biased, allowing the collector current to flow to the summing node, thus limiting  $V_{OUT}$ . A similar action occurs with  $Q_2$  and  $D_2$  as  $V_{OUT}$  goes below  $-V_L - 1V$ .

You can thus program the limiting levels or the maximum output voltage of the amplifier symmetrically over  $V_{L(MIN)}+1V$  to  $V_{L(MAX)}+1V$  with a resolution of  $[(V_{L(MAX)}+1)-(V_{L(MIN)}+1V)]/2^nV$  in accordance with the 8-bit digital-input binary word. The circuit becomes a programmable positive/negative limiting amplifier if you remove the appropriate diode-transistor pairs from the feedback. (DI #2201) e

**To Vote For This Design, Circle No. 416**

## FIGURE 1



**NOTE: ALL CAPACITORS ARE IN MICROFARADS.**

The maximum output of this amplitude limiter is digitally programmable over  $\pm 2$  to  $\pm 10V$  in  $2^n$  steps, where n is the number of bits of the DAC.

# Get more than three colors from a dot-matrix LED

W KURDTHONGMEE, NAKORN SI THAMMARAT, THAILAND

Dot-matrix LEDs find wide use in advertising displays. Products now on the market range from an inexpensive 5×8 (row-by-column) single-color LED to an expensive 8×8 RGB device. The method provided here allows you to obtain more than three main colors from an 8×8 tricolor LED. In fact, tricolor dot-matrix LEDs have only two LED dies—red and green. When you apply current to one, you obtain a red or a green color. When you apply current to both, orange results. The circuit in **Figure 1**, used in conjunction with the MCS-51 code in **Listing 1**, works efficiently in controlling the LED to generate various shades of the three colors.

To add tones or shades of the main colors to the tricolor LED, you do not need to modify the circuit in **Figure 1**; you need only consider the software. Software modifications consist of adding more color planes or pages of display buffer, adding memory locations (mapped onto the LED dots), and increasing the number of refresh times, in which the controller updates all LED dots to cover all added color planes.

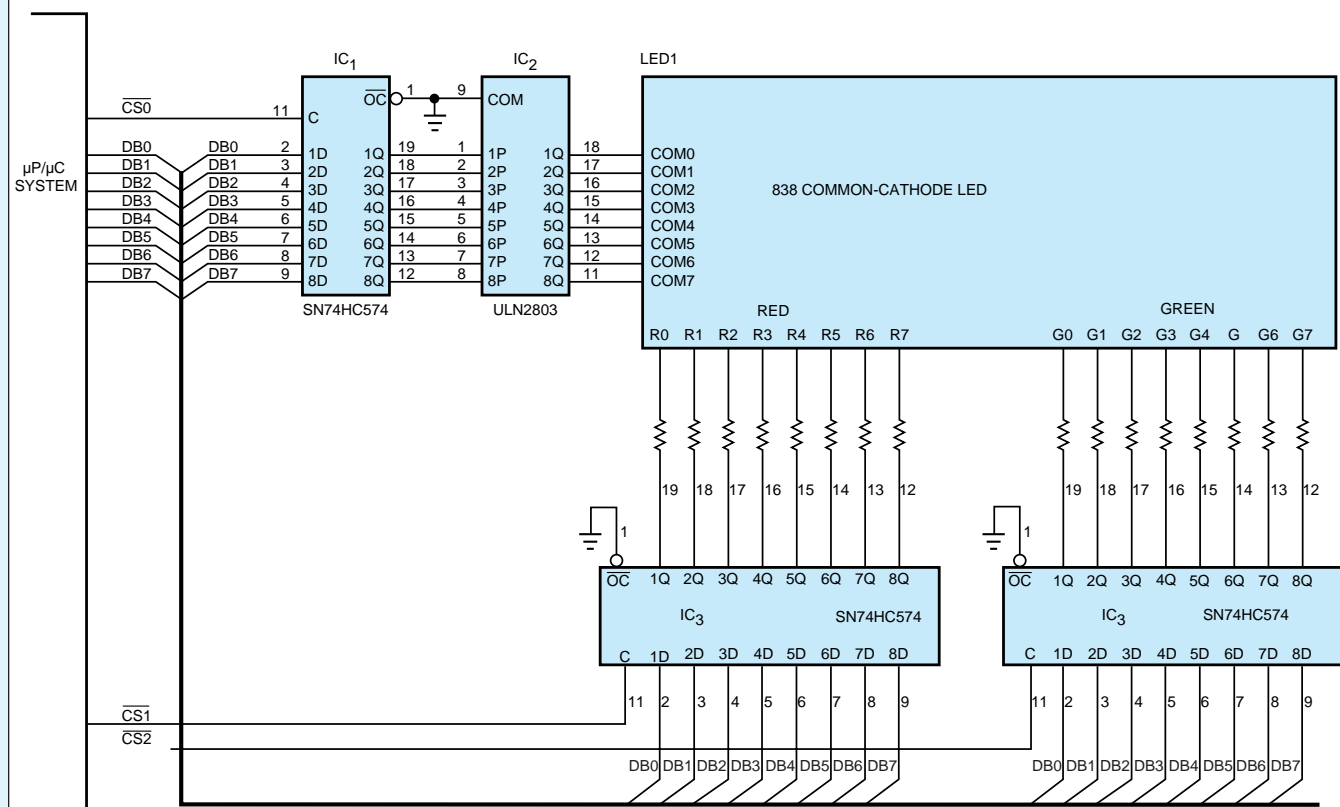
**TABLE 1—VALUE IN COLOR PLANE MAPPED TO DOT 1**

Red 1	Red 2	Green 1	Green 2	Color
0(1)	1(0)	0	0	Red 50%
1	1	0	0	Red 100%
0	0	0(1)	1(0)	Green 50%
0	0	1	1	Green 100%
0(1)	1(0)	0(1)	1(0)	Orange 50%
1	1	1	1	Orange 100%
0	0	0	0	Blank

For example, if you decide to use four color planes, divided into two red and two green planes, for dot 1 of the dot-matrix LED, you'll obtain the shades listed in **Table 1**.

In addition, by allocating eight color planes (four red and

**FIGURE 1**



A few TTL circuits and some MCS-51 code allow you to obtain more than three colors from a tricolor dot-matrix LED.

four green), you can obtain the color shades listed in **Table 2**. Note that only the number of ones in the color planes controls color appearance. Therefore, the permutations do not change the color, as long as the numbers of ones in **Table 2** remain constant. For example, the values 0110, 1001, 1100, and 0011 for R1 through R4 all produce the same color: orange 50%. You can download **Listing 1**—as well as the MCS-51 code that produces 13 colors from an 8×8 tricolor LED—from *EDN's* Web site, [www.ednmag.com](http://www.ednmag.com). At the registered-user area, go to the “Software Center” to download the file from DI-SIG #2195.

Note that, in practice, bitwise output ports control the LED. To assign a color to a dot, the routine must extract a bit from a byte and then assign the bit value of the selected color plane by plane. (DI #2195)

**TABLE 2—VALUE IN COLOR PLANE MAPPED TO DOT 1**

Red 1	Red 2	Red 3	Red 4	Green 1	Green 2	Green 3	Green 4	Color
0	0	0	1	0	0	0	0	Red 25%
0	0	1	1	0	0	0	0	Red 50%
0	1	1	1	0	0	0	0	Red 75%
1	1	1	1	0	0	0	0	Red 100%
0	0	0	0	0	0	0	1	Green 25%
0	0	0	0	0	0	1	1	Green 50%
0	0	0	0	0	1	1	1	Green 75%
0	0	0	0	1	1	1	1	Green 100%
0	0	0	1	0	0	0	1	Orange 25%
0	0	1	1	0	0	1	1	Orange 50%
0	1	1	1	0	1	1	1	Orange 75%
1	1	1	1	1	1	1	1	Orange 100%
0	0	0	0	0	0	0	0	Blank

**To Vote For This Design, Circle No. 417**

## LISTING 1—MCS-51 CODE FOR SIX COLORS FROM A TRICOLOR LED

```

;*****
; Program Name : TriCol.asm
; Generate three colours from triple colours dot matrix LED.
;*****
; Note:
; Only a sample implementation on 8x8 triple colours LED
;*****
; Author      : Wattanapong Kurdthongmee,
;              : School of Physics, Walailak University, Thaiburi,
;              : Tha-Sa-La, Nakorn si Thammarat, 80160 Thailand.
;*****
;*****
; Global variable declarations:
;*****
temp      equ    10h
dBuf      equ    11h

;*****
; Interrupt vectors
;*****
org       0000h
jmp       main
org       000bh
jmp       Timer0SR
;*****
; Timer 0 interrupt service routine
; Update row-by-row the triple-colour 8x8 dot matrix LED.
; To make more tones of colour, this routine reads 4 set of display buffer
; and scan 4 times.
;*****
Timer0SR:
    setb   rs0           ; Select set of registers in bank 1
    push   dph
    push   dpl
    push   psw
    push   acc
    mov    th0,#0ffh     ; Re-initialize timer registers
    mov    tl0,#00h
    ;*****
    mov    dptr,#0a000h  ; Clear enable controlled port
    mov    a,#00h        ; before updating
    movx   @dptr,a
    ;*****
    mov    a,#dBuf
    add    a,r0
    push   acc
    mov    r1,a
    mov    a,@r1
    ;*****
    ; Read from current address at a current
    ; plane and update column controlled port
    ; colour by colour
    mov    dptr,#8000h   ; Red colour port
    movx   @dptr,a
    pop     acc
    add    a,#08h        ; Difference display buffer between RED and
    ; green colour plane
    mov    r1,a
    mov    a,@r1
    mov    dptr,#9000h   ; Green colour port
    movx   @dptr,a
    ;*****

mov    dptr,#0a000h     ; Enable current row (controlled by ULN2803)
mov    a,r3
movx   @dptr,a
    mov    r3,a
    ;*****
    ; Next row (there are 8-row for 8x8 LED)
    inc    r0
    cjne   r0,#08h,T0SR_r
    mov    r0,#00h
    mov    r3,#01h
T0SR_r:
    pop     acc          ; Restore registers
    pop     psw
    pop     dpl
    pop     dph
    clr     r0
    reti

;*****
; Main program starting here
;*****
main:    mov    sp,#60h

    setb   rs0           ; Initialise register in bank 1 to be used
    mov    r0,#00h       ; the timer interrupt service routine.
    mov    r3,#01h
    clr    r0
    mov    tmod,#21h     ; timer 0: mode 1, timer 1: mode 2
    mov    tcon,#0ddh
    mov    th0,#0fdh     ; Initial value for timer
    mov    tl0,#00h
    setb   ea            ; Enable all interrupts
    setb   tr0           ; Start timer
    setb   et0           ; Enable timer interrupt 0

    ; Sample patterns to show different shades
    ; colours on the LED.
    mov    r0,dBuf
    mov    r1,#00h

main_0:  mov    a,#01010011b
    mov    @r0,a
    inc    r0
    inc    r1
    cjne   r1,#08h,main_0
    mov    r0,dBuf+8
    mov    r1,#00h
    mov    a,#10101111b
main_1:  mov    @r0,a
    inc    r0
    inc    r1
    cjne   r1,#08h,main_1
    sjmp   $
    end

```



# Implement a nine-data-bit UART on a PC

AUBREY KAGAN, WEIDMULLER LTD, MARKHAM, ON, CANADA

Many  $\mu$ Cs, such as the 8051 and the 68HC11, can support a ninth data bit on the asynchronous serial port. This bit is useful in multidrop applications in which you can use it to denote an address on the serial bus, as opposed to data destined for a particular address. The UART used in IBM PCs (and clones) does not directly support this operating mode. However, through some software manipulation, you can add the PC to a serial bus and integrate it into a ninth-bit system, albeit with some limitations.

The method differs for data reception and transmission. As a result, the PC can work only in half-duplex mode. Because half-duplex communication is common practice on PC networks, this limitation is not a significant drawback. The technique also requires that the CPU check each incoming byte for the ninth bit. (You can usually configure a  $\mu$ C to generate an interrupt when the ninth bit is set.) For the PC to receive the nine bits, it is necessary to treat the ninth bit as a parity bit. Although it's impossible to read the parity bit in the PC's UART directly, it is possible to analyze the received data byte and determine what the parity should be.

If analysis reveals a parity error, then the value of the ninth bit is opposite to the calculated parity. If no error exists, then the value of the ninth bit is equal to the calculated parity. In the 16550 UART, the FIFO includes the three error bits with each data byte, so the parity error (or lack thereof) is always

associated with the current data byte. It is possible, however, to disable the FIFO feature. The technique for transmission is slightly different. The 8250/16450/16550 UART has a forced-parity format (also known as a "stick" parity), in which you can set the parity to a one or to a zero. You do this by setting bit 5 (stick parity) and bit 3 (parity enable) in the UART's line-control register (LCR). The transmitted parity bit is then the logical inverse of bit 4 of the LCR.

In the sample code in **Listing 1**, address 0xff (with bit 9 set) is reserved and used to indicate the last byte of the transmission. The first byte of the transmission is an address, and it transmits with bit 9 set. The RS-232C port connects to an RS-232C/RS-485 converter, where the RTS line controls the direction. The code given here is not interrupt-driven, but you could implement it as an interrupt-driven routine. The code comprises three modules: background (back.cpp), serial procedures (serial.cpp), and memory declaration (mem.cpp). Note that mem.cpp declares one include file (mem.h) for the public memory. You can download the files from EDN's Web site, [www.ednmag.com](http://www.ednmag.com). At the registered-user area, go to the Software Center to download the files from DI-SIG #2198. (DI #2198)

e

**To Vote For This Design, Circle No. 418**

## LISTING 1—BACKGROUND CODE FOR NINTH-BIT TRANSMISSION

```
//background program
//developed with Turbo C++
//operating under DOS

#include "mem.h"
#include <conio.h>

#define RTS 0x2

//prototypes
void setupUART (void);
void deAssert (int ControlPin);
void Assert (int ControlPin);
unsigned char SerialIn(void);
void UARTTx(void);
void UARTTx(void);
unsigned char UART_TX_clear( void);
unsigned char SerialOut (void);
unsigned int checksum (unsigned char NumberOfBytes);

void main (void)
{
    unsigned int j;

    comport=1;
    //setting to COM1

    module_address=0xa;
    //PC address=10 decimal

    //other transmission constants
    setupUART();
    //initialise the UART

    capture_enabled=0;
    rx_pnt=0;
    //initialise variables

    Assert(RTS);
    //turn the RS485 buffer to receive

    while (1)
    {
        /*the actions are divided into several states as indicated by
        the variable "phase".
        Phase=0- waiting for a complete serial message
        Phase=1- preparing a response
        Phase=2- wait for end of transmission
        Phase=3- wait for message to completely clear the UART (buffers
        empty) & then
        re-enable reception (turn RS485 buffer around)*/

        switch (phase)
        {
            case 0:
                if (SerialIn())
                {
                    //checking for complete message received
                    {
                        //now to process the input
                        phase++;
                        UARTTx();
                        //prepare UART to send
                    }
                }
                break;
            case 1:
                //prepare to transmit
                rx_buff[0]=0x0; //destination address
                rx_buff[1]=0x13; //response
                rx_buff[2]=0xff; //set last byte.
                number_of_characters=3;
                //variable for transmit routine
                tx_pnt=0;
                //initialise the fetch pointer
                phase++;
                break;
            case 2:
                if (SerialOut())
                {
                    //at the end of the message
                    //bump on to wait for complete transmission
                    phase++;
                }
                break;
            case 3:
                //wait for message to clear
                if (UART_TX_clear())
                {
                    UARTTx();
                    phase=0;
                }
                break;
            default:
                break;
        }

        if (kbhit())
        {
            //terminate execution if any key pressed.
            break;
        }
    }
}
```

## Design Idea Entry Blank

Entry blank must accompany all entries. \$100 Cash Award for all published Design Ideas. An additional \$100 Cash Award for the winning design of each issue, determined by vote of readers. Additional \$1500 Cash Award for annual Grand Prize Design, selected among biweekly winners by vote of editors.

To: Design Ideas Editor, EDN Magazine  
275 Washington St, Newton, MA 02158

I hereby submit my Design Ideas entry.

Name \_\_\_\_\_

Title \_\_\_\_\_

Phone \_\_\_\_\_ Fax \_\_\_\_\_

E-mail \_\_\_\_\_

My e-mail address may be published Yes \_\_\_\_\_ No \_\_\_\_\_

Company \_\_\_\_\_

Address \_\_\_\_\_

Country \_\_\_\_\_ ZIP \_\_\_\_\_

Design Idea Title \_\_\_\_\_

Social Security Number \_\_\_\_\_  
(US authors only)

Entry blank must accompany all entries. (A separate entry blank for each author must accompany every entry.) Design entered must be submitted exclusively to EDN, must not be patented, and must have no patent pending. Design must be original with author(s), must not have been previously published (limited-distribution house organs excepted), and must have been constructed and tested. Fully annotate all circuit diagrams. Please submit software listings and all other computer-readable documentation on a IBM PC disk in plain ASCII.

Exclusive publishing rights remain with Cahners Publishing Co unless entry is returned to author, or editor gives written permission for publication elsewhere.

In submitting my entry, I agree to abide by the rules of the Design Ideas Program.

Signed \_\_\_\_\_

Date \_\_\_\_\_

Your vote determines this issue's winner. Vote now, by circling the appropriate number on the reader inquiry card.