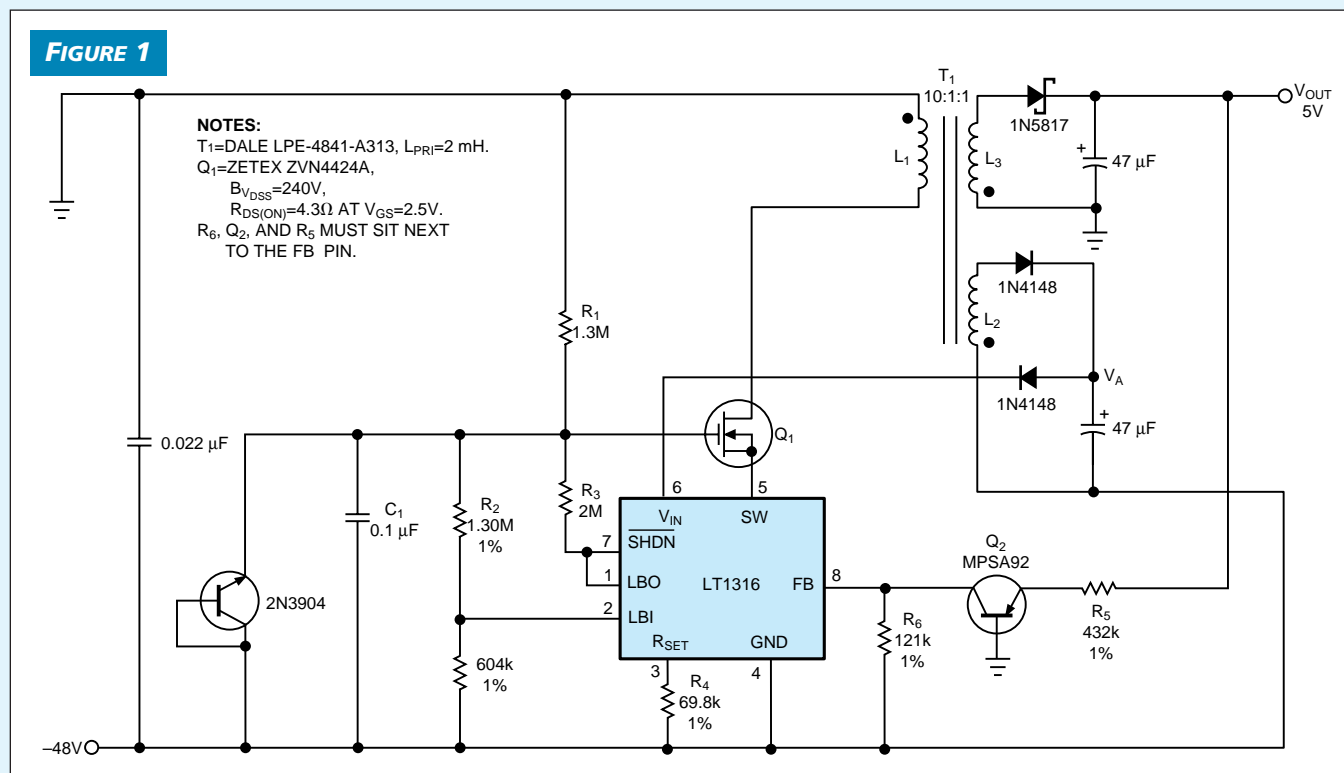


DC/DC converter operates from phone line

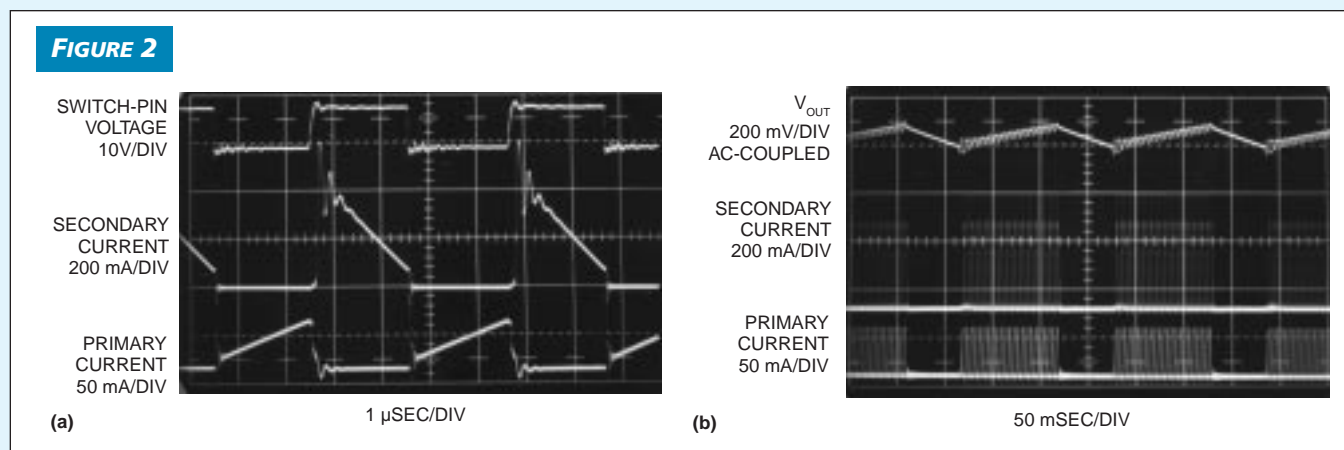
GARY SHOCKEY, LINEAR TECHNOLOGY CORP, MILPITAS, CA

DC/DC converters for use inside the telephone handset require operation from the high-source-impedance phone line. Additionally, the CCITT specifications call for maximum on-hook power consumption of 25 mA. The dc/dc

converter in **Figure 1** is 70%-efficient at an input power of 25 mA, providing 5V at 3.4 mA. Controlled, low-peak switch current ensures that the -48V input line experiences no excessive voltage drops during switching.



A -48-to-+5V flyback converter provides 3.4 mA of output current.



Switch voltage and current waveforms (a) show how the primary current ramps up during the switching cycle. The output ripple voltage is approximately 100 mV p-p (b).

The circuit operates as a flyback regulator with an auxiliary winding to provide power for the LT1316 switching-regulator IC. When you first apply power, the LBI pin is low, causing the SHDN pin to connect to ground through LBO. Grounding the SHDN pin places the part in shutdown mode, and only the low-battery comparator remains active. During this state, V_{IN} rises at a rate determined by R_1 and C_1 . The IC draws only 6 μ A in shutdown mode. R_1 needs to supply only this shutdown current, the current through R_2 and R_3 , and C_1 's charging current.

When LBI reaches 1.17V (which corresponds to a V_{IN} of approximately 3.7V), the LBO pin lets go of SHDN and the IC enters the active mode; switching action begins, and the output voltage begins to increase. As the device switches, the V_{IN} pin draws current out of C_1 . V_{IN} then decreases sufficiently to trip the low-battery detector, stopping the switching. Start-up proceeds in this irregular fashion until, eventually, the voltage at V_A increases to 5V. (V_A is the same as V_{OUT} because L_2 and L_3 have the same number of turns.) After start-up, current flows to the IC from V_A rather than from the -48V rail, increasing efficiency. The circuit will not

start if V_{OUT} is loaded before it reaches 5V.

During each switch cycle, current in the transformer primary ramps up until reaching the current limit (Figure 2a). The value of R_3 sets the peak switch current. The circuit uses a 69.8-k Ω resistor to provide a peak switch current of 50 mA; increasing R_3 decreases the current limit. Secondary peak current is approximately equal to the primary peak current multiplied by the transformer's turns ratio (Figure 2b). The FB pin has a sense voltage of 1.23V, and you can set V_{OUT} by the following formula:

$$V_{OUT} = 1.23 \left(\frac{R_5}{R_6} \right) + 0.6V.$$

For the load currents of 4 to 80 mA, the circuit achieves a minimum of 70% efficiency. Less than 80 μ A quiescent current flows when the converter supplies 0.5 mA over 36 to 72V. (DI #2148) **EDN**

To Vote For This Design, Circle No. 356

Transfer data frames over asynchronous RS-232C lines

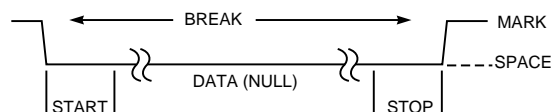
SK SHENOY, NAVAL PHYSICAL AND OCEANOGRAPHIC LAB, KOCHI, INDIA

The asynchronous RS-232C interface is a simple, low-cost option for interconnecting processor-based systems. In many applications, you need to transfer variable-size messages. However, the character-oriented RS-232C protocol offers no direct mechanism for transferring messages as self-contained packets. The method described here uses an obscure feature found in most UART devices to indicate packet boundaries. The feature is the capability to transmit and recognize the "Break" character. This character is nothing but a "space," or low, in the transmit line of a duration equal to or greater than an entire asynchronous character-transmission time, including the start and stop bits (Figure 1). In this framing method, the message data bytes sandwich between two Break characters to form a data frame (Figure 2).

A Turbo C program demonstrates the transfer of variable-size messages between two PCs with 8250-compatible UARTs (Listing 1). You can download the program from EDN's Web site, www.ednmag.com. At the registered-user area, go into the Software Center to download the files from DI-SIG, #2140.

A null-modem cable interconnects the PCs' COM ports. The same routine works with most other UARTs. The method allows data-packet reception in interrupt mode and wastes no CPU overhead looking at each character to detect packet boundaries. Instead, the UART does the detecting. Because the Break is not a legitimate data character, it is data-

FIGURE 1



Most UARTs can transmit and recognize the Break character, a state of logic 0 between the start and stop bits.

FIGURE 2



The Turbo C routine in Listing 1 sandwiches data bytes between two Break characters to form a data frame.

transparent, and you can use it for binary-data exchange. You can use this “in-band” scheme with repeaters and modems, as long as they permit transmission of the Break condition. The packet-boundary detection is relatively immune to a missed Break character and to data errors. You can render the detection more robust by introducing data-length and check-sum fields in the frame to allow detection of errors and flow control using an RTS/CTS (request-to-send/clear-to-send) handshake.

To transmit a Break, set bit 6 (Set Break) of the line-control register to 1. The UART then sets its Tx line low, until bit 6 encounters a 0. Transmission of a Null character (00 hex) makes the duration of the Break equal to one character-transmission delay. Bit 6 of the line-status register (Tx Machine Status) indicates when this delay is over; then, the Break bit resets. To enable detection of the Break, bit 2 of the interrupt-enable register (interrupt-on-Rx-error condition)

sets during UART initialization. Bit 0, set to 1, enables receive-data interrupts. In the interrupt-service routine (ISR), bits 1 and 2 of the interrupt-identification register indicate the interrupt type.

A global variable, Receive_Count, initialized to zero, handles frame reception. Upon detection of a Break, the UART raises an interrupt. If Receive_Count is zero, the interrupt is a start-of-frame break and the UART ignores it. (You can use the interrupt to set a Packet_Receive_On flag.) On each subsequent receive interrupt, the ISR stores the data in the Receive buffer with Receive_Count as the index. If Receive_Count is nonzero when the Break interrupt is raised, the interrupt is an end-of-frame break. Then the routine calls the frame-processing function and resets Receive_Count to zero. (DI #2140)

EDN

To Vote For This Design, Circle No. 357

LISTING 1—DATA-FRAME-TRANSFER PROGRAM

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>

/* COM PORT DEFINITIONS AND GLOBAL VARIABLES */
#define com_reg 0x3f8 /* Default is com1; 2f8 for com2 */
#define DATA_PORT com_reg + 0
#define LINE_CNTRL com_reg + 3
#define MODEM_CNTRL com_reg + 4
#define INT_ENABL com_reg + 1
#define INT_IDENT com_reg + 2
#define LINE_STS com_reg + 5
#define MODEM_STS com_reg + 6
#define BAUD_LOW com_reg + 0
#define BAUD_HIGH com_reg + 1
#define DLAB_SET 0x80
#define BAUDMSE 0
#define BAUDLSE 0xc /* 9600 BPS */
#define CNTRL_CMD 7 /* 8 BIT, 2 STOP BIT, NO PARITY */
#define WAIT_TX_RDY() while (((inportb(LINE_STS)) & 0x60) != 0x60)
/* Check for Tx buf empty & Tx shift reg empty */

unsigned char sdatatabuf[256], rdatatabuf[256]; /* Send & Recv buffers */
int Receive_Count = 0; /* Counter for data stored in rdatatabuf[] */
void interrupt(*OldComHandler)(void);

/* FUNCTION CALLED TO DISPLAY RECEIVED DATA PACKET */
void processdata(void)
{
    int i;
    printf("\n\rRX > "); clrreol(); /* Received data cursor */
    for (i = 0; i < Receive_Count; i++) /* Display received data */
        putchar(rdatatabuf[i]);
    printf("\n\rTX > "); clrreol(); /* Transmitted data cursor */
}

/* INTERRUPT SERVICE ROUTINE TO TAKE CARE OF PACKET RECEPTION */
void interrupt service_sio(void)
{
    unsigned char iir;
    iir = (inportb(INT_IDENT) >> 1) & 3; /* Get interrupt type */
    switch(iir)
    {
        case 0: /* Modem status int DSR,CTS,RI,RLSD */
            inportb(MODEM_STS); /* Ignore; reading IIR resets int */
            break; /* reading IIR resets int */
        case 1: /* Tx int */
            break; /* reading IIR resets int */
        case 2: /* Rx int */
            rdatatabuf[Receive_Count++] = inportb(DATA_PORT); /* Store packet break;
            break;
        case 3: /* Rx error ( Break detect etc.) */
            inportb(DATA_PORT); /* NULL char */
            if(((inportb(LINE_STS)) & 0x10) == 0x10)
                /* Break detected; Reading LSR Resets int */
                if (Receive_Count) processdata(); /* EndOfFrame Break Process
                /* Else Start of Frame Break. Do nothing */
                /* Else Receive error; Drop packet */
                Receive_Count = 0; /* Re-initialise for next packet */
    }
    outportb(0x20, 0x20); /* EOF */
    return;
}

/* FUNCTION TO INITIALISE SERIAL PORT */
void init_serial_io(void)
{
    outp(LINE_CNTRL, DLAB_SET); /* DLAB SET */
    outp(BAUD_LOW, BAUDLSE); outp(BAUD_HIGH, BAUDMSE); /* 9600 BAUD */
    outp(LINE_CNTRL, CNTRL_CMD); /* 8 BIT, 2 STOP BIT, NO PARITY */
    outp(MODEM_CNTRL, 8); /* DTR, RTS & OUT2 SET */
    OldComHandler = getvect(0xc); /* 0xb for com2 */
    disable();
    setvect(0xc, (service_sio)); /* 0xb for com2 */
    outportb(0x21, ((inportb(0x21)) & (10x10))); /* PIC mask word 0x8 for
    outportb(INT_ENABL, 0x5); /* IER enable Rx Machine error & RX Data
    enable();
}

/* FUNCTION TO TRANSMIT A BREAK OF ONE CHARACTER DURATION */
void SendBreak(void)
{
    outportb(LINE_CNTRL, inportb(LINE_CNTRL) | 0x40); /* LCR; set break */
    outportb(DATA_PORT, 0); /* Send NULL data */
    WAIT_TX_RDY(); /* Wait on TxShift Reg Empty; Null char is shifted out
    outportb(LINE_CNTRL, inportb(LINE_CNTRL) & 0xbf); /* LCR; remove break
}

/* FUNCTION TO TRANSMIT A DATA PACKET */
void SendBuffer(unsigned char packet[], int DatLen)
{
    int i;
    SendBreak(); /* Send START OF PACKET break */
    for (i=0; i<DatLen; i++) /* For each message byte*/
    {
        WAIT_TX_RDY(); /* Wait for Tx Ready */

        outportb(DATA_PORT, packet[i]); /* send one data char */
    }
    WAIT_TX_RDY(); /* Wait on TxShift Reg Empty; last char is shifted out
    SendBreak(); /* Send END OF PACKET break char */
}

/* BARE-BONES APPLICATION; TAKES STRING INPUT (TERMINATED BY ENTER) FROM
KEYBOARD AND TRANSMITS AS A PACKET. ALSO DISPLAYS RECEIVED PACKETS */
void main(void)
{
    int c, count = 0;

    init_serial_io(); /* Initialise serial port */
    printf("\n\rTX > "); /* Transmit Prompt */
    while(1) /* Forever Loop */
    {
        if((c=getche()) == 27) break; /* Exit if Escape key pressed */
        sdatatabuf[count++] = c;
        if(c == '\r') /* If Enter Key pressed */
        {
            putchar('\n'); clrreol(); /* Go to newline */
            printf("TX > "); /* Transmit Prompt */
            SendBuffer(sdatatabuf, count); /* Transmit Data Packet */
            count = 0; /* Reset Tx data count */
        }
    }
    setvect(0xc, OldComHandler); /* Restore int vector; 0xb for com2 */
    outportb(0x21, ((inportb(0x21)) | (0x10))); /* PIC mask word 0x8 for com2
}

```

Controller provides multiple alarm-driver formats

WILLIAM GRILL, RIVERHEAD SYSTEMS, LITTLETON, CO

Using a piezoelectric element for alarm applications offers low cost, low power, and flexibility. By coupling this element with a 12C508 programmable controller (Microchip Technology, Chandler, AZ), you can implement an eight-pin alarm generator. This approach provides multiple driver formats with a minimum of additional cost and footprint.

The controller in **Figure 1** drives the piezoelectric element directly from pins 2 and 7 with complementary square waves. A siren, chirp, warble, or constant-alarm output is available by setting the corresponding mode on pins 5 and 6 (**Table 1**). The design codes each mode as separate processes, which you can consider as variations of frequency, frequency step, and dwell.

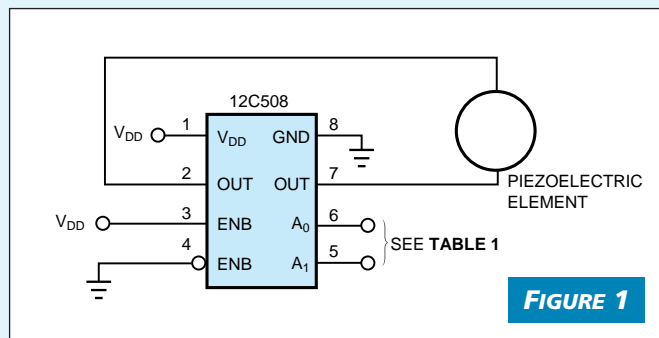
The design also codes the positive and negative true alarm enables, pins 3 and 4, into the device. The controller retests the mode and these alarm enables at periodic intervals in the currently selected mode's cycle. This retesting permits dynamic selection of the output formats using the mode pins without a power reset. Applications can then use any or

all of the output formats to indicate application alarm or status conditions.

The 12C508's internal RC oscillator provides the timing control used in each of the modes. Using an average of 3 mA, the controller operates from 2 to approximately 5.25V. The frequency-stepped formats are in constant "timebased" increments with constant frequency dwell times. Based on a 2.2-kHz piezo-element resonant peak, each of the mode's characteristics uses code-settable, dedicated registers to establish the output format.

The coded sequences use 127 bytes of code space. You can port the sequences into one of several code-compatible Microchip controllers or use a stand-alone peripheral controller, as in **Figure 1**, for any number of alarm applications. You can download applicable code from *EDN's* Web site, www.ednmag.com. At the registered-user area, go into the Software Center to download the file from DI-SIG, #2147. (DI #2147) EDN

To Vote For This Design, Circle No. 358



A programmable controller and a piezoelectric element combine to produce an alarm generator with multiple output formats: continuous frequency, two chirps, a warble, or a siren.

TABLE 1—STRAPPED ALARM CONFIGURATIONS

A ₁	A ₀	Alarm output
0	0	Continuous 2.2 kHz
0	1	Two single 2.2-kHz chirps: 0.2 sec each with 0.2 sec between
1	0	Warble: eight frequency steps swept continuously from below to above to below 2.2 kHz
1	1	Siren: repeating six frequency steps beginning at 2 kHz and stepping up to approximately 4.1 kHz

Algorithm extracts cube root

JOHN T HANNON JR, PHILIPS CONSUMER ELECTRONICS CORP, KNOXVILLE, TN

The C routine in **Listing 1** generates the cube root of either a positive or a negative number. The number can range from a small fraction to greater than 1 billion. Note that this idea is irrelevant for a PC, which includes a math library with the compiler and produces a more accurate result with less effort. However, this idea is useful if you use

a processor for which you don't have a math library.

The main routine inputs a number from the keyboard and calls the *cube root* function. After calculating the cube root, the routine prints the result on the screen. The routine then recalculates the cube of this cube-root answer and prints this number on the screen so you can compare the accuracy of

the result with the original number. Five decimal places are set up for the print command to allow an accurate comparison.

This algorithm is similar to the square-root algorithm that uses the divide-and-average technique to reach a minimum error value. Before the cube-root operation begins, the routine determines if the number is positive or negative. If the value is negative, the program sets a flag so the returned root value can be set to a negative value. To start the process, the routine sets up an error value and assigns an initial value for the integer *root*. The error value for this function is 0.00001, and the initial value of *root* is 2.0. The routine squares this value and divides it into the given number. The routine then adds the result to the original value of *root* and divides this sum by 2 to generate the new value of *root*. This process continues until the absolute value of the difference between the cube of the value of *root* and the given number is less than the error value—in this case, 0.00001.

For some values, this algorithm approaches but never reaches the set error. Thus, the routine increments a counter after each iteration. If the counter reaches the maximum set count before the routine finds the minimum error, the function ends and returns the value of *root*. If the original number was negative, the routine changes the cube root to a negative value.

With the values in **Listing 1**, the accuracy for numbers from 0.1 to greater than 1 billion is better than 0.001%. For numbers less than 0.2, the accuracy is slightly lower. However, this accuracy difference is a result of the error value that you use in the calculation. By setting the error value (float variable *error*) to a smaller value, such as 0.000001, the accuracy for very small numbers can also approach this value.

Two other factors affect the accuracy: the number of bits the processor uses for floating-point numbers, and the size of the original number. For numbers with small absolute values, the variable *error* controls the accuracy. The accuracy, which is

$$\frac{(\text{calculated root} - \text{actual root})}{\text{actual root}},$$

is approximately equal to

$$\frac{\text{error variable}}{3 \times \text{original number}}.$$

So, with *error* set to 0.00001, the accuracy is about 0.003% for the cube root of 0.1, 0.03% for the cube root of 0.01, 0.3% for the cube root of 0.001, and so forth. In the limit—the cube root of 0—you can't divide by 0 to find percent accuracy, but the actual root that the program calculates is

LISTING 1—CUBE-ROOT EXTRACTOR

```

/* Function to calculate the cube root of a number. */
#include <stdio.h>

float absval(float x)          /* GENERATE THE ABSOLUTE VALUE */
{
    if (x < 0)
        x = -x;
    return (x);
}

float cuberoot (float num)     /* CUBE ROOT FUNCTION */
{
    float error = 0.00001;    /* SET UP MAXIMUM ERROR */
    float root = 2.0;        /* SET STARTING VALUE */
    int negflag = 0, count = 0;

    if (num < 0)              /* IF NUMBER IS NEGATIVE */
    {
        num = -num;          /* CHANGE TO POSITIVE & */
        negflag = 1;        /* SET NEG. NUMBER FLAG */
    }

    while (absval(root * root * root - num) >= error)
    {
        root = (num/(root * root) + root)/2.0;
        ++count;
        if (count > 25)      /* IF NO MINIMUM ERROR AFTER 25 */
            break;          /* ITERATIONS, EXIT THE FUNCTION */
    }                        /* THIS COULD BE LARGER FOR VERY LARGE NUMBERS */

    if (negflag == 1)        /* IF ORIGINAL NUMBER WAS NEGATIVE */
        root = -root;      /* SET ROOT TO NEGATIVE NUMBER */
    return (root);
}

main()
{
    float number, root, newnum;

    clrscr();
    printf ("\n\n");
    printf ("\tEnter the number for cubed root: ");
    scanf ("%f", &number);  /* INPUT A VALUE FOR CUBEROOT */

    root = cuberoot (number); /* CALL CUBE-ROOT FUNCTION */
    printf ("\n\n\tThe cube root of %.3f is %.5f.", number, root);

    newnum = root * root * root; /* CUBE ANSWER FOR COMPARISON */
    printf ("\n\n\n\t%.5f cubed is %.3f.", root, newnum);
    return(0);
}

```

0.0156250.

For numbers with large absolute values, the number of iterations determines the accuracy. For instance, the algorithm calculates the cube root of 10^{12} as 10083.87109 after 25 iterations and as 10000.00000 after 45 iterations. (DI #2144) EDN

To Vote For This Design, Circle No. 359

Shunt regulator provides overvoltage protection

ROBERT N BUONO, MAHWAH, NJ

The circuit in **Figure 1** uses a typical technique for varying the output voltage of a power supply via a programmable control voltage. Although the topology and schematic details of the power supply are not critical, the protection technique is novel.

The control IC is a UC3843AN PWM controller. This IC applies 2.5V to the noninverting input of an internal error amplifier but does not bring this input out to a pin (node B). The inverting input of the error amplifier is available at an external pin (node A). To regulate V_{OUT} , the control IC must maintain the voltage at Node A equal to the 2.5V at Node B. The component values in the **figure** allow the dc output voltage of the power supply to vary between a minimum of 5V and a maximum of 75V, as a function of $V_{CONTROL}$, which can vary from 0 to 3V (corresponding to a V_{OUT} of 5V and 75V, respectively).

In the absence of Q_1 , V_{OUT} and the voltage division of R_3 and R_4 determine the voltage at Node A. The circuit compares this voltage with the 2.5V at Node B. The power supply's output-voltage control loop keeps the voltage at Node A equal to Node B by appropriately controlling V_{OUT} .

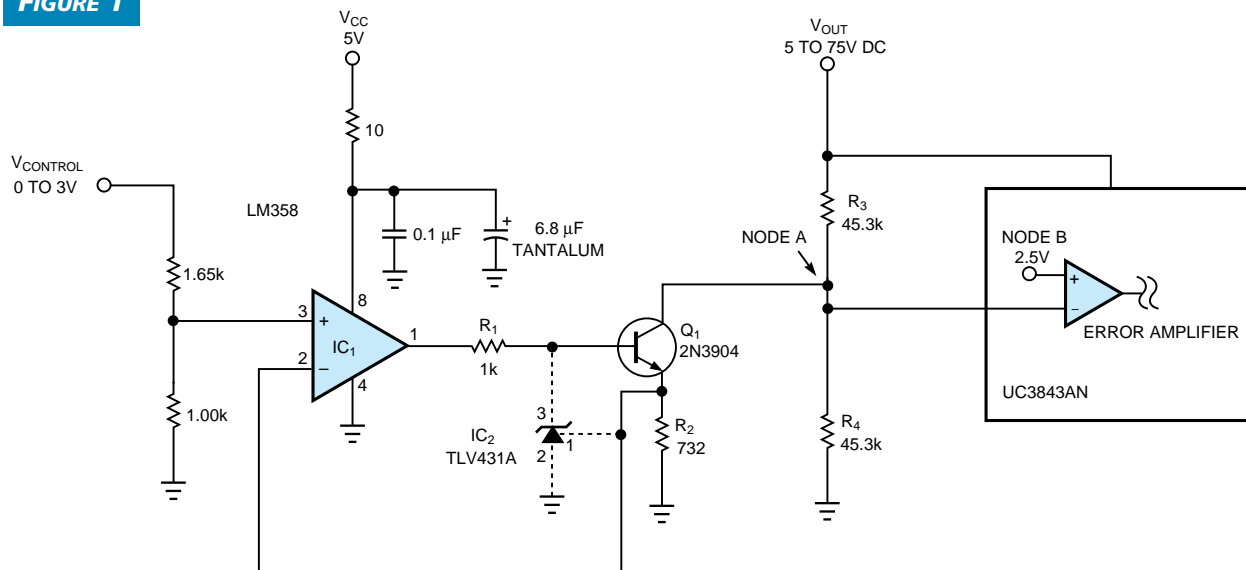
The circuit provides programmability of V_{OUT} by sinking current from Node A. V_{OUT} must source any current that flows from this node. Also, the current must flow through R_3 , which causes the voltage drop across R_3 to increase. V_{OUT}

is then always equal to the voltage drop across R_3 plus a current set by $V_{CONTROL}$. IC_1 's op amp forces the voltage across R_2 to equal the voltage at Pin 3 of IC_1 .

The addition of just one component, IC_2 , adds precise overvoltage protection to the variable-output power supply. IC_2 is a low-voltage shunt regulator that incorporates an internal 1.24V precision reference. This low reference voltage allows you to use this protection circuit with conventional power-supply control ICs for which 2.5V is a common internal reference voltage. Under normal operating conditions (for output voltages between 5 and 75V), IC_2 does nothing. The voltage at IC_2 's reference (Pin 1) is less than its internal 1.24V reference, so its cathode (Pin 3) draws no current. In this case, IC_1 solely controls the voltage at the base of Q_1 . For example, if $V_{CONTROL}$ is 3V, then the voltage across R_2 is 1.13V and V_{OUT} equals 75V. Note that, to simplify this example, the beta of Q_1 is assumed to be infinite.

However, in the event of any kind of fault that might cause the voltage across R_2 to rise above 1.24V (which corresponds to a maximum V_{OUT} of 81.7V), the shunt regulator begins to function. As the voltage at Pin 1 of IC_2 begins to exceed the internal 1.24 reference voltage, the cathode of IC_2 begins to conduct. The cathode of IC_2 then pulls down on the base of Q_1 to maintain 1.24V at Pin 1 of IC_2 . As this happens, IC_2 through Q_1 controls the voltage across R_2 . This con-

FIGURE 1



Adding one shunt regulator, IC_2 , to an otherwise typical programmable power supply provides precise overvoltage protection.

control causes the output of IC₁ to saturate at a positive voltage of approximately 3.7V because IC₁ can no longer keep the voltage across R₂ equal to the voltage at its input (Pin 3). The benefit to circuit operation is that IC₂ now operates with a constant-cathode current determined by the voltage across R₁, which equals 3.7–1.8V/1kΩ=1.9 mA. This level of cathode current ensures that IC₂ regulates properly.

This protection circuit is immune to any potential failure

mode of IC₁'s op amp or the programming voltage source, V_{CONTROL}. IC₁ operates only from 5V. If its output (Pin 1) shorts to ground, the minimum V_{OUT} results. If its output shorts to V_{CC}, IC₂ sinks 5–1.8V/1 kΩ=3.2 mA, and V_{OUT} clamps at the maximum of 81.7V. (DI #2146)

EDN

To Vote For This Design, Circle No. 360

LED flasher and triac pulser work off ac line

DENNIS EICHENBERG, PARMA HEIGHTS, OH

A flashing LED is an excellent visual alarm. Unfortunately, the LED is a dc device and requires additional circuitry to operate from an ac source. Several circuits can perform the necessary function, but the circuit in **Figure 1a** is the most efficient. This circuit is also reliable, compact, and inexpensive.

The F336HD red-flashing LED (part no. 276-036 at Radio Shack) operates directly from 5V and produces a consistent pulse of light at approximately 1 Hz without a time-constant capacitor. This LED starts immediately when you apply power and is insensitive to temperature variations. The W04G full-wave bridge rectifier produces a full-wave dc waveform from the 120V-ac line. The 0.5-μF capacitor provides current limiting for operating the LED from the rectified 120V-ac line. The 100Ω resistor protects the circuit from

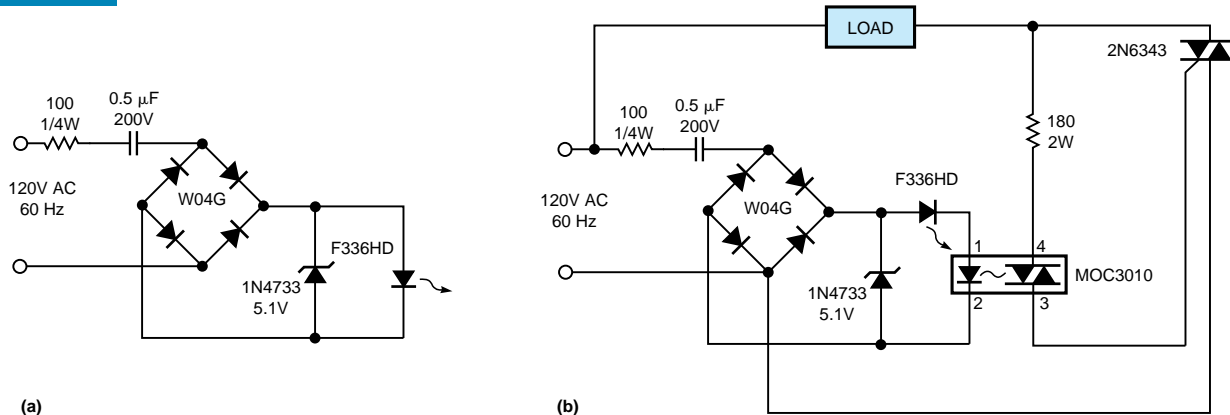
surges when you first apply power. The 1N4733 5.1V zener diode protects the LED from high-voltage excursions.

Some applications require a more intense alarm. A simple triac pulser can pulse a 120V-ac lamp or other resistive load of as much as 8A (**Figure 1b**). This circuit is also reliable, compact, efficient, and inexpensive. The circuit is similar to the one in **Figure 1a**, but, in this case, the F336HD LED drives an MOC3010 opto-coupled triac driver. The 180Ω resistor provides current limiting for the 2N6343 triac gate. This configuration can pulse a load as high as 960W. You can increase the power rating by choosing a different triac. (DI #2143)

EDN

To Vote For This Design, Circle No. 361

FIGURE 1



A full-wave bridge rectifier provides a dc signal for the red-flashing LED (a). Adding a triac allows the circuit to pulse a 120V-ac lamp or other resistive load of as much as 8A (b).

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.

The winning Design Idea for the August 15, 1997, issue is entitled "Gates provide low-cost sine-wave generator," submitted by Adolfo Mondragon of Philips Components (Juarez, MX).