Efficient LED power supply has battery backup

Zihong Yu, Juno Lighting Group, Des Plaines, IL

LEDs find wide use in emergency lighting because of their high efficiency and control simplicity. The circuit in Figure 1 provides a highly efficient and reliable design for emergency LED lighting at 3 to 6W. The circuit’s input is 12V ac, which the full-wave bridge rectifies and one or two capacitors filter into dc. The battery (not shown) is a 12V lead-acid type. IC₁ compares the battery voltage to the supply voltage. When the rectified voltage drops below the battery voltage, the battery takes over to provide LED power.

The circuit has some small switching losses, which should be acceptable as long as IC₂, a 12V PB137 battery-charging circuit from STMicroelectronics (www.st.com), keeps the battery from draining. If this switch-over is unacceptable, add a 470-μF electrolytic capacitor to filter the input voltage to maintain a certain level above the battery voltage. Note that adding this capacitor lowers the power factor.

To get 12V ac, you can use an electronic transformer. These transformers

---

**Figure 1** This circuit converts ac voltage to dc voltage, charges a battery, and drives LEDs from the ac source or the battery.
provide 12V at a higher frequency, so a 10-μF capacitor can hold the voltage high as well as provide a high power factor.

IC1, a Linear Technology (www.linear.com) LTC4412, controls two external PFETs that create a near-ideal diode function for switching between ac and battery output. The PFETs’ voltage drop is only about 20 mV compared with a normal 0.7V diode-voltage drop. Pin 5 is low when ac power is off, so you can use this pin to turn on a warning LED through another PFET. IC1 has an internal current limit of 1.5A. Resistor R1 limits IC1’s input; when the current reaches a certain level, Q4 turns off the charging circuit. This IC does not require reverse-diode protection.

IC2, an LT3517 LED driver from Linear Technology, acts as an inverting buck-boost converter because the input can range from 8 to 17V for rectified ac. R10 sets up the LEDs’ current. Because the voltage drop from each of the three LEDs varies from 3 to 4V, the IC’s output voltage can be higher or lower than its input voltage if all 300-mA LEDs connect in series.

By connecting a resistor divider, including a photocell, to the analog-dimming pin, Pin 8, you can achieve some dimming, which results in some power savings at higher ambient light. You can use IC1’s Pin 5 to turn on a transistor or an optoisolator to pull IC1’s control-pin voltage lower if you need to dim the LED when ac power is out. Resistor R5 programs IC1 to operate at 1 MHz. The circuit’s efficiency is 82% when you power it directly from the ac power supply and about 70% from an electronic transformer.

With a few minor changes to the circuit, you can add LEDs. For example, you can use Linear Technology’s LT3518, which is a pin-to-pin-compatible version of the LT3517 but with a higher switching-current limit. You may need to adjust the feedback-resistor pair R5 and R15 for higher output voltage. You may also need more input-filtering capacitance to hold up the voltage.

Tests show that the circuit can power as many as six LED in series. Figure 2 shows the circuit in operation.

Figure 2 LEDs provide enough light for emergency lighting.

Single IC forms precision triangular-wave generator

Akshay Bhat, Maxim Integrated Products Inc, Sunnyvale, CA

The linearity of triangular waveforms makes the triangular-wave generator useful in sweep circuits and test equipment. For example, switched-mode power supplies and induction motor-control circuits often include a triangular-wave oscillator as part of their PWM (pulse-width-modulation) circuit.

The basic triangular-wave generator includes an integrator for generating the triangular-wave output and a comparator with external hysteresis, such as a Schmitt trigger, for setting the output amplitude (Figure 1). You can implement these components with a Maxim (www.maxim-ic.com) MAX9000 IC, which includes a high-speed operational amplifier, a 185-nsec comparator, and a precision 1.23V bandgap reference.

The integration of a constant current, which you obtain by applying constant voltage across a resistor, produces a linear ramp at the op amp’s output. This output feeds a Schmitt trigger whose output feeds back to the integrator resistor. Abrupt state changes in the Schmitt trigger’s output determine the peak voltages for the triangular-wave output. These changes in turn depend on the input threshold voltages you set for the Schmitt trigger.

Unfortunately for this circuit, the triangular-wave peaks must be symmetrical about the reference voltage you apply to the comparator’s inverting input. To generate a triangular wave from 0.5 to 4.5V, for example, you must provide a reference voltage of (0.5V+4.5V)/2=2.5V.

It would be preferable to set this voltage range independently of the standard bandgap-reference voltage.
available, 1.23V. You can achieve this flexibility by adding resistor R_C to the hysteresis network in a single-IC version of the circuit (Figure 2). R_1 lets you set the triangular-wave peaks independently of the reference voltage.

To build the Schmitt-trigger comparator, you first select R_C. The comparator’s input-bias current at C_COM is less than 80 nA. To minimize the error this current causes, the current through R_C, 
\[ (V_{REF} - V_{OUT})/R_C \] , should be at least 8 \( \mu \)A. R_C requires two equations, corresponding to the two possible comparator-output states: V_OUT = V_REF/R_C and V_OUT = (V_DD - V_REF)/R_C.

Use the smaller of the two resulting resistor values. For example, if the supply voltage is 5V, the reference voltage is 1.23V, and the reference current is 8 \( \mu \)A, the two R_C values are 471.25 and 153.75 k\( \Omega \), so this circuit uses the standard value of 154 k\( \Omega \).

Next, select R_1 and R_2. During a rising ramp, the comparator output is logic low (V_IL). Similarly, the comparator output is at logic high (V_IH) during a falling ramp. Thus, the comparator must change state according to the required peak and valley points of the triangular wave.

Two simultaneous equations result when you apply nodal analysis at the noninverting input of the comparator and solve for these two thresholds:

\[
\frac{V_{IH}}{R_1} + \frac{V_{SS}}{R_2} = V_{REF}\left(\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}\right)
\]

and

\[
\frac{V_{IL}}{R_1} + \frac{V_{DD}}{R_2} = V_{REF}\left(\frac{1}{R_1} + \frac{1}{R_2} + \frac{1}{R_3}\right)
\]

In this example, the voltage range of the triangular wave is 0.5 to 4.5V. You therefore substitute a value for V_{REF} of 4.5V, V_{SS} of 0.5V, V_{DD} of 5V, and V_{IH} of 1.23V into the above equations to obtain a value of 124 k\( \Omega \) for R_1 and 66.5 k\( \Omega \) for R_2.

You can now design the integrator. Considering the comparator’s two possible output states, the magnitude of current flowing through R_4 is:

\[ I_{R_4} = (V_{DD} - V_{REF})/R_4 \]

The op amp’s maximum input-bias current is 2 nA. To minimize error, therefore, the current through R_4 must always be greater than 0.2 \( \mu \)A. This constraint implies that R_4’s value is less than 612 M\( \Omega \).

The triangular-waveform frequency is:

\[
f = \frac{1}{2\pi} \left(\frac{V_{OUT-P}-V_{OUT-P}}{(V_{CC} - V_{REF})} (R_4C) + \frac{V_{OUT-P}}{V_{REF}} (R_4C)\right)
\]

For this example, the frequency is 25 kHz, the output voltage is 4V p-p, or 0.5 to 4.5V for a triangular wave, and the reference voltage is 1.23V. Solving for the resulting time constant, R_4 = 9.27 \( \mu \)sec. Select a capacitance of 220 pF and a value of 42.2 k\( \Omega \) for R_4.

The resulting output should match the desired frequency, provided that the op amp is not slew-limited. Because the feedback capacitor charges or discharges with a constant current, the output signal’s maximum rate of
Use a low-cost PWM ramp generator in switch-mode power supplies

Dwayne Reid, Edmonton, AB, Canada

The circuit in Figure 1 shows a PWM (pulse-width-modulated) ramp generator that you can use in low-cost switch-mode dc/dc power supplies. Its supply voltage can range from 5 to 35V dc, and you can set the output-ramp amplitude of 0.3 to 1V. You can also set a minimum off time that lets you set a maximum 50% duty cycle for magnetic components that need duty-cycle limiting.

The ramp generator (Figure 1a) uses one-half of an LM393 dual comparator. The other half of the comparator is available to generate the PWM portion of the converter. The ramp amplitude and frequency depend on the reference. An ordinary red LED can act as a low-cost reference. Its forward voltage of approximately 1.7V is reasonably constant over indoor temperature ranges. The ratio of $R_1$ to $R_2$ sets the ramp amplitude relative to the reference, and $R_3$, $R_4$, and $C_1$ set the minimum off time. $R_4$ and $C_2$ establish a time constant, which sets the period. Note that the $R_3$, $R_4$, and $C_1$ network also affects the period. Table 1 shows examples of various configurations.

Figure 1b, a 70V-dc upconverter, employs the ramp generator. You can easily configure it at any output ranging from the highest input voltage to whatever the FET can handle. This
example uses a 330-μH inductor, but you can easily change that value by choosing the appropriate PWM frequency.

Note that the output FET does not turn on quickly, and it doesn’t need to, but it does turn off quickly. You can enhance the turn-off speed by adding a 2N4403 PNP transistor between the output of the comparator and the pullup resistor. Connect the base to the comparator, the emitter to the FET gate, and

### Listing 1: Interrupt-Flag Checker

```c
#define SwitchMask 0x01 // Port 1 bit(s) connected to switches.
#define Ndebounce 3 // (# of consecutive checks w/o an edge to wait for valid state)-1

void chk_sw(void) {
    static unsigned char debounced_sw; // The debounced switch state(s)
    // (output)

    void main() {
        P1DIR = ~SwitchMask; // Set unused pins to output mode, unless
        debounced_sw = P1IES = P1IN; // Init so we catch the first state
        P1IFG = P1IE = 0; // Keep Port 1 interrupts disabled.
        _EINT(); // Set General Interrupt Enable (if used).

        while (1) {
            // Main program loop.
            debounced_sw = P1IES; // Do this here if loop execution period
            if (P1IFG & SwitchMask) { // Is fixed by a timer,
                // o/w use a timer to run chk_sw(). If
                _EINT(); // chk_sw() is made a timer ISR, then
                // debounced_sw will be volatile here.
            }
        }
    }

    // Port1 interrupt request
    void chk_sw(void) {
        static int db_count = Ndebounce; // Update on first pass of chk_sw().
        if (P1IFG & SwitchMask) { // Try to force P1IES
            P1IFG &= ~SwitchMask; // Clear the switch interrupt flag(s).
            db_count = Ndebounce; // Must see P1IFG not set for
            // Ndebounce+1 passes.
        } else if (db_count) {
            if (P1IES = P1IE = 0) { // Cock for opposite edge (repeated in case we
                _EINT(); // miss an edge while doing it).
                db_count = 0;
            } else // Finished debounce
                debounced_sw = P1IES; // Debounced switch output
                // Can put code here that you want to run only when switch
                // inputs are stable.
        } else // If you care about the switch inputs only in this block, you can read the debounced
        // read the debounced state directly from P1IES, no need to save it
        // in debounced_sw.
    }
}
```

#### Table 1: Examples of Configurations

<table>
<thead>
<tr>
<th>Frequency (Hz)</th>
<th>$R_1$ (Ω)</th>
<th>$C_1$ (nF)</th>
<th>$R_2$ (Ω)</th>
<th>$C_2$ (F)</th>
<th>Approximate duty cycle (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>120k</td>
<td>100</td>
<td>1M</td>
<td>220k</td>
<td>10n</td>
</tr>
<tr>
<td>700</td>
<td>100k</td>
<td>100</td>
<td>100k</td>
<td>22k</td>
<td>100n</td>
</tr>
<tr>
<td>62k</td>
<td>4.7k</td>
<td>10</td>
<td>100k</td>
<td>22k</td>
<td>10n</td>
</tr>
<tr>
<td>100k</td>
<td>4.7k</td>
<td>10</td>
<td>100k</td>
<td>22k</td>
<td>100p</td>
</tr>
<tr>
<td>200k</td>
<td>2.2k</td>
<td>3.3</td>
<td>100k</td>
<td>22k</td>
<td>3.3n</td>
</tr>
<tr>
<td>200k</td>
<td>1.8k</td>
<td>10</td>
<td>100k</td>
<td>22k</td>
<td>100p</td>
</tr>
<tr>
<td>400k</td>
<td>2.2k</td>
<td>1</td>
<td>100k</td>
<td>22k</td>
<td>10p</td>
</tr>
<tr>
<td>400k</td>
<td>5.6k</td>
<td>2.2</td>
<td>100k</td>
<td>22k</td>
<td>470p</td>
</tr>
</tbody>
</table>

The circuit has slow load-transient response, which you can adjust by altering the time constant that $R_2$ and $C_2$ form. Note that $R_2$ and $C_2$ form a voltage divider that ensures the lowest error voltage at the PWM comparator is above the ramp’s lowest point. The converter cannot operate without $R_p$. EDN

---

**MSP430’s port-interrupt-request logic helps debounce contacts**

Richard Neubert, Manchester, NH

Contact debouncing requires monitoring an input and waiting for it to stop toggling or at least establish that it’s definitely switching from its initial state. You can use either analog or digital filtering plus hysteresis to accomplish contact debouncing, but this approach uses a lot of resources, including parts, board space, and CPU time, when multiple inputs need to be conditioned. Alternatively, you can either detect just the first state change or sample the input at least twice as often as the contacts can bounce. Sampling at mechanical vibration frequencies must be avoided. Both methods also require time delays to ensure that the contacts have finished bouncing.

These 1-bit approaches are attractive when you must condition multiple contact inputs. The first method requires conservative delay setting to avoid resuming the edge monitoring before the last bounce, and it’s unsuitable for re-

---

**TABLE 1: EXAMPLES OF CONFIGURATIONS**

<table>
<thead>
<tr>
<th>Frequency (Hz)</th>
<th>$R_1$ (Ω)</th>
<th>$C_1$ (nF)</th>
<th>$R_2$ (Ω)</th>
<th>$C_2$ (F)</th>
<th>Approximate duty cycle (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>120k</td>
<td>100</td>
<td>1M</td>
<td>220k</td>
<td>10n</td>
</tr>
<tr>
<td>700</td>
<td>100k</td>
<td>100</td>
<td>100k</td>
<td>22k</td>
<td>100n</td>
</tr>
<tr>
<td>62k</td>
<td>4.7k</td>
<td>10</td>
<td>100k</td>
<td>22k</td>
<td>10n</td>
</tr>
<tr>
<td>100k</td>
<td>4.7k</td>
<td>10</td>
<td>100k</td>
<td>22k</td>
<td>100p</td>
</tr>
<tr>
<td>200k</td>
<td>2.2k</td>
<td>3.3</td>
<td>100k</td>
<td>22k</td>
<td>3.3n</td>
</tr>
<tr>
<td>200k</td>
<td>1.8k</td>
<td>10</td>
<td>100k</td>
<td>22k</td>
<td>100p</td>
</tr>
<tr>
<td>400k</td>
<td>2.2k</td>
<td>1</td>
<td>100k</td>
<td>22k</td>
<td>10p</td>
</tr>
<tr>
<td>400k</td>
<td>5.6k</td>
<td>2.2</td>
<td>100k</td>
<td>22k</td>
<td>470p</td>
</tr>
</tbody>
</table>
jecting noise. The second method adds to the actual bouncing time only the delay necessary for bridging the longest quasistatic input state during bounce—not the longest duration of bouncing. When you implement this function in software, however, it can add substantial overhead for monitoring the input for further transitions when the system detects a transition.

The Texas Instruments (www.ti.com) MSP430-series microcontrollers have I/O ports with configurable interrupt logic for each bit. You can select the rising or falling edge of the bit as the trigger. Even with the interrupt disabled, the microcontroller can read its interrupt-request flag to determine whether an active edge has occurred. You can use this technique in place of high-rate sampling in software. You must periodically check the interrupt-request flag at a rate high enough only to keep the switch response delay to an acceptable time. Calling the routine at the frequency of a mechanical vibration isn’t an issue; the interrupt logic monitors the switch between calls.

You can use multiple switch inputs, provided that the switches either never change state simultaneously—for example, with a keypad—or you don’t mind delaying response to a switch until all simultaneously changing switch inputs have settled. You would lose the sequence of switch operations in this case; in the debounced output, they would all change at once.

Listing 1 periodically calls function chk_sw() to check the interrupt-request flags and update the output value, debounced_sw. The time interval between calls times Ndebounce should be short enough to satisfy the required response time after the last bounce and longer than the longest time between transitions when the contacts are bouncing. In a noisy environment, making the delay too long is counterproductive because noise transients during the delay extend the delay.

If your application has no task to run except in response to a switch, it’s fairly simple to make a version in which the CPU sleeps with the port interrupts enabled (P1IE=SwitchMask) and the P1IES bits set to the last input state when there is no switch input activity. A port interrupt-service routine must respond to the first input change to set P1IE=0 and set a timer to periodically call chk_sw() until chk_sw() resolves the input state. When reacting to a brief noise impulse, the CPU would wake up once for the port interrupt and Ndebounce+1 times to run chk_sw().