design ideas

Edited by Bill Travis

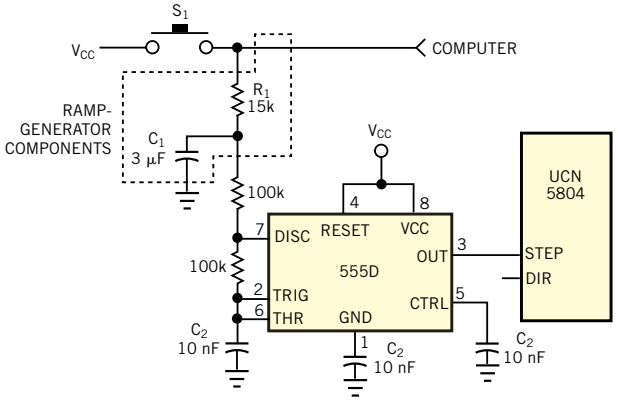
Make a simple ramp generator for stepper motors

Richard Brien, GSI Lumonics, Wilmington, MA

Stepper motors are synchronous motors that step at the pulse rate of the driving signal. For the motor to move quickly, the stepping rate must be fast. However, because of motor and load inertia, the motor often cannot go from 0 rpm to the desired number of revolutions per minute in one step. Therefore, most stepper motors receive their drive from a pulse chain that starts out slowly and then increases in rate until the motor reaches the desired rate. To stop the motor, the drive signal must not abruptly stop; it must gradually decrease or ramp down to zero. Microprocessors can easily generate the needed ramp-up and then ramp-down signals, often called a trapezoidal profile, but in any circuit without a microprocessor, this ramp is difficult to generate.

The 555-based bistable circuit in **Figure 1** can easily generate a pseudo-trapezoidal move profile. Note that the timing string of R_1 and associated components does not connect to V_{CC}, as it would in a normal circuit, but instead receives its power through a pushbutton switch.

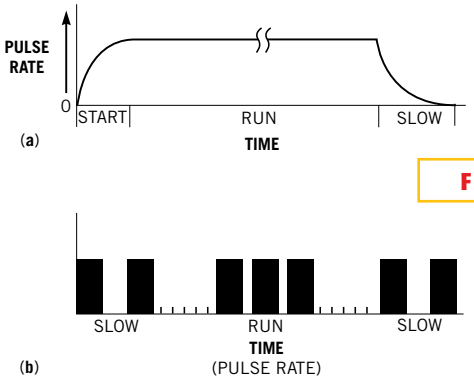
Figure 1



This circuit generates a pseudo-trapezoidal motion-control profile for controlling stepper motors.

When you push the button, capacitor C_1 starts charging up to a point at which C_2 can start charging. As C_1 charges, the output frequency of the 555 starts off slowly and gradually increases to a frequency or pulse rate that's a function of all the components in the timing string. This final frequency is lower than the frequency the circuit would adopt, if C_1 and R_1 were not in the string. When you release the pushbutton, the 555 does not immediately stop running but ramps down in frequency until it finally stops (**Figures 2a** and **2b**). The ramp frequencies generated do not follow a linear profile, but neither do those in most microprocessor-driven circuits. The ramp-frequency profile of the circuit should resemble that of **Figure 2a**, depending on the component values.

Figure 2



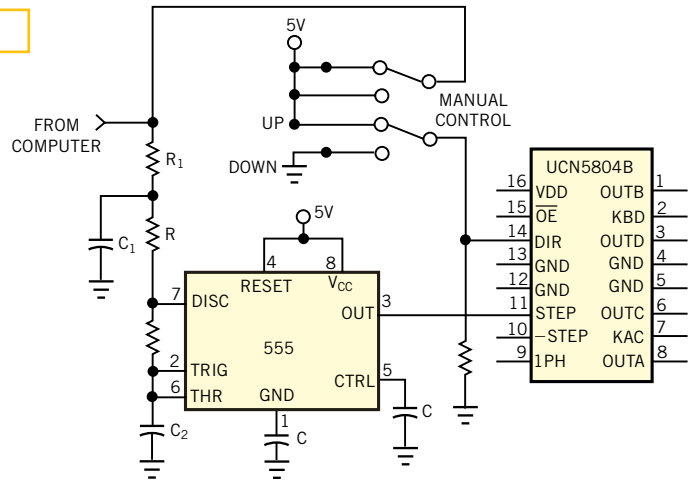
The move profile from the circuit in Figure 1 is roughly trapezoidal (a); the step-rate profile exhibits low frequencies at ramp-up and -down (b).

You can operate this circuit with a simple pushbutton. This concept opens a world of manual control to stepper motors. Usually, stepper motors do not use manual control, because of the difficulty of generating a trapezoidal frequency profile in hardware. With this circuit, you can use low-torque, low-revolu-

Publish your Design Idea in *EDN*. See the What's Up section at www.ednmag.com.

tions-per-minute stepper motors in systems in which you formerly needed dc gearbox motors. By changing the pushbutton to a dpdt switch, you can make a stepper motor run clockwise and then counterclockwise without microprocessor control (**Figure 3**). These concepts also apply to stepper-motor-based linear actuators. You could also replace the pushbutton with a control signal from a computer or a controller, thus allowing stepper motors to take their drive from controllers that do not have a built-in ramp-generating function.

Figure 3



Is this the best Design Idea in this issue? Select at www.ednmag.com.

By changing the pushbutton in Figure 1 to a dpdt switch, you can make a stepper motor run clockwise and then counterclockwise without microprocessor control.

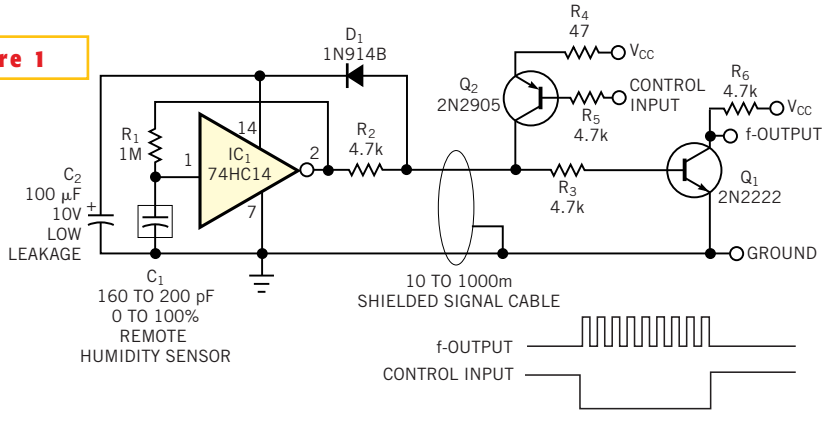
Remote humidity sensor needs no battery

Shyam Tiwari, Sensors Technology Private Ltd, Gwalior, India

Using AC-line power sources and batteries for remote humidity sensors is undesirable because these sources can be troublesome if you mount them in inaccessible points, such as smokestacks, cold-storage chambers, or darkrooms, where maintenance is difficult and inconvenient. **Figure 1** shows a simple way to remove the power source from the humidity-sensor circuit. The circuit uses a 160- to 200-pF, capacitor-type HS1101 humidity sensor from Humirel for the circuit. IC_1 forms a classic oscillator of time constant R_1C_1. The center frequency is 5 to 10 kHz, depending on the value of R_1. The circuit charges the 100-µF, low-leakage capacitor, C_2, through the diode, D_1, directly from the output line of the sensor oscillator, IC_1. C_2 becomes the power source for IC_1.

When you make measurements, you stop charging C_2 for a short time by removing the Control Input signal (0V for charging and V_{CC} or floating for measurement). You then measure the frequency of the signal at the output of transistor Q_1. Transistor Q_2 supplies enough current to charge C_2 through R_4. The HS1101 sensor measures approximately 160 pF at 0% relative humidity and 200 pF at 100% relative humidity. Therefore, frequency decreases with increasing relative humidity. Although the sensor has a linear response within ±5% of full-scale range over 1 to 25°C, you should calibrate the circuit at several humidity points. You can find more details about the HS1101 sensor at www.humirel.com or www.sensorstechnology.tripod.com/humidity.html.

Figure 1



Eschew troublesome power sources for remote humidity sensors by using this simple scheme.

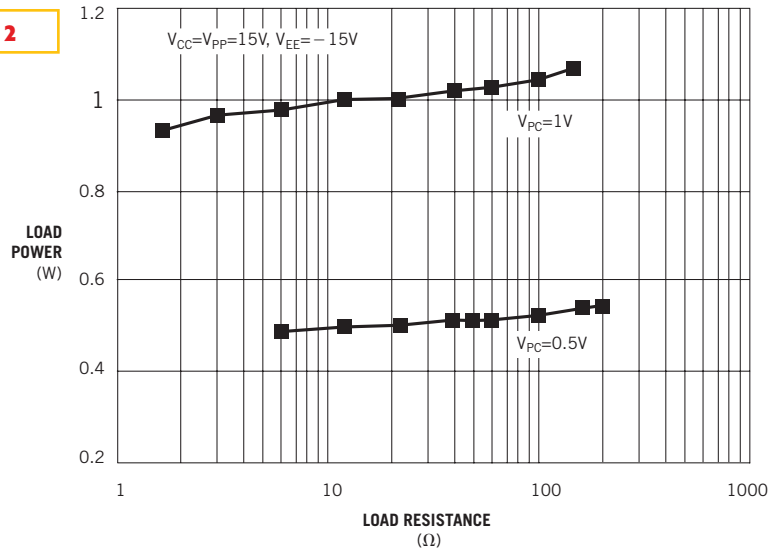
Is this the best Design Idea in this issue? Select at www.ednmag.com.

Power source is insensitive to load changes

Ken Yang, Maxim Integrated Products, Sunnyvale, CA

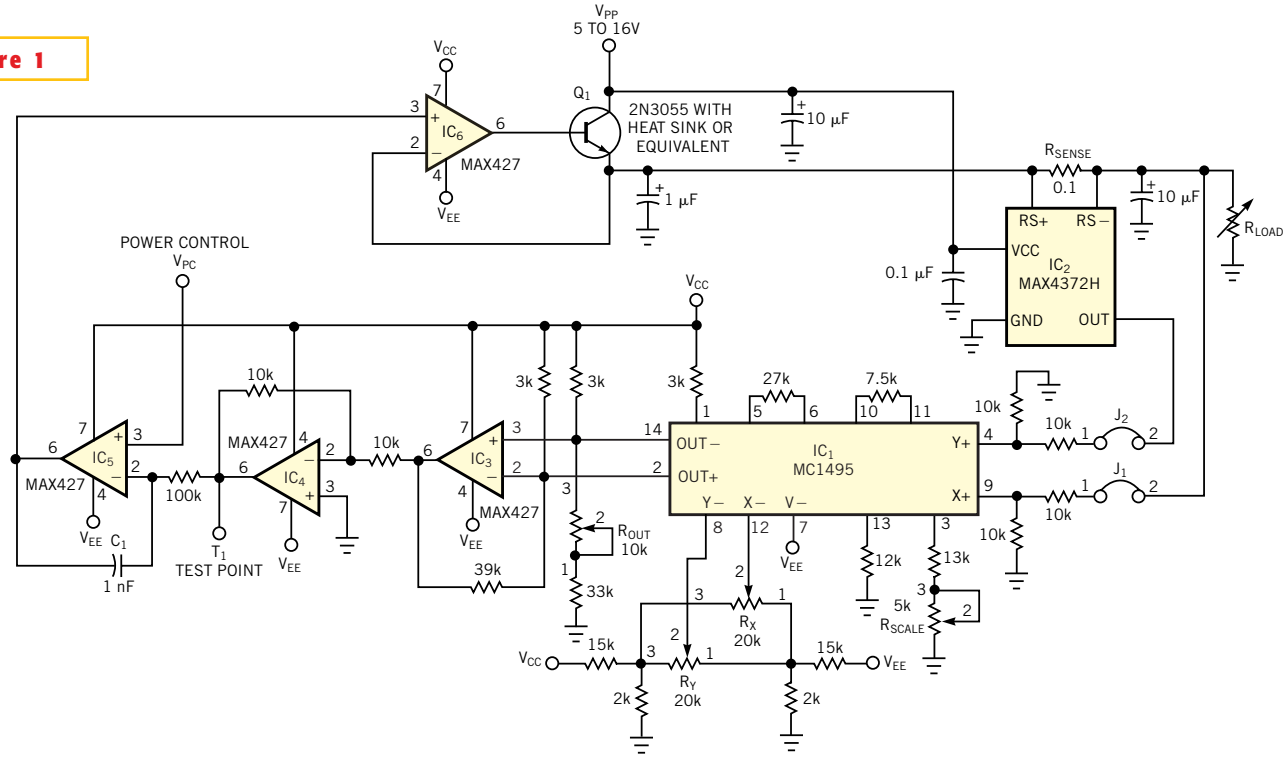
For the heating and cooling elements common in industrial systems, resistance is not a fixed quantity. These elements include such devices as positive-temperature-coefficient heaters and thermoelectric coolers. Their resistance can change more than 100% during operation, and the result is a change in power dissipation for elements receiving drive from a fixed voltage or current source. Worse, excessive power can damage the heating or the cooling element. Driving the element with a fixed and regulated power driver overcomes these problems (**Figure 1**). The circuit is analogous to a voltage or a current source but delivers fixed power levels that are independent of the load resistance. A feedback loop senses load power and automatically adjusts the output voltage to maintain the desired pow-

Figure 2



Power delivery to the load is nearly independent of load resistance.

Figure 1



This regulated power source delivers fixed power to a varying load.

er level. The circuit measures the output current with a current sensor, IC_2 and then determines the output power by multiplying the current by the voltage with the use of a four-quadrant analog-voltage multiplier, IC_1 and IC_3.

Because the multiplier's output is inverted, you add a unity-gain-inverting stage, IC_4, to reinvert the output-power signal. Op amp IC_5 then compares output power to the reference power, V_{PC} input, and integrates any difference between them. The integrator provides an automatic power adjustment by increasing or reducing the output voltage until output power equals reference power. IC_6 and Q_1 form a voltage follower that drives the load. The following formula sets output power: $V_{PC}=10PR_{SENSE}$, where P is the desired output power in watts, R_{SENSE} is the sensing resistor in ohms, and V_{PC} is the reference-power input in volts. If, for example, the desired load power is 1W, and $R_{SENSE}=0.1\Omega$, then set V_{PC} to 1V.

Curves for load power versus load resistance for 0.5 and 1W loads show that power delivered to the load changes less than ±7% for a change of 10,000% (two decades) in load resistance (**Figure 2**). If you define load regulation as the change in output power divided by the output power, then for a load change of 6 to 40Ω at 1W, the load regulation is ±2%. For the circuit to work properly, you must calibrate the analog multiplier as Motorola's MC1495 data sheet delineates. You repeat that calibration procedure below for convenience. Remove jumpers J_1 and J_2 for the calibration.

1. X-input offset adjustment: Connect a 1-kHz, 5V p-p sine wave to the Y input. Connect the X input to ground. Using an oscilloscope to monitor test point T_1, adjust R_X for an ac null (zero amplitude) in the sine wave.
2. Y-input offset adjustment: Connect a 1-kHz, 5V p-p sine wave to the X input. Connect the Y input to ground. Using an oscilloscope to monitor test point T_1, adjust R_Y for an ac null (zero amplitude) in the sine wave.
3. Output-offset adjustment: Connect the X and Y inputs to ground. Adjust R_{OUT} until the dc voltage at T_1 is 0V dc.
4. Scale-factor (gain) adjustment: Connect the X and Y inputs to 10V dc. Adjust R_{SCALE} until the dc voltage at T_1 is 10V dc. Repeat steps 1 through 4 as necessary.

Is this the best Design Idea in this issue? Select at www.ednmag.com.

Encrypted routines impede hackers, protect licenses

Lawrence Arendt, Oak Bluff, MB, Canada

The use of public-key-encrypted algorithms within licensed applications can prevent hackers from cracking the licensed algorithms. Moreover, you can use them to disable licensed features that a user doesn't purchase. Licensing schemes eventually arrive at some critical decision point at which an algorithm decides whether the user is entitled to run that application. If not, the algorithm usually generates a console message or a message box to notify the user of his or her ineligibility. Then, the application terminates or continues in a feature-limited mode. The decision code generally resolves to one of the following (or similar) commands in assembly code:

Jump/Branch if not equal (for example, JNE, BNE)

Jump/Branch if equal (for example, JE, BEQ).

Typically, a hacker searches a disassembled listing for the termination message, finds the routine that displays the message, and then traces backward through the calling sequence until he locates the decision code. Often, changing a single hex byte in a binary image of the executable code completely bypasses the decision logic. Or changing the byte can reverse the decision logic. (The new copy runs only if it *does not* have a license.) The edited file can then generate a new .exe file. For example, you could replace the hex code for a JNE with a JMP (jump always) or a JE. To thwart hackers,

1. Select the code, C_{PT}, consisting of a single routine or a block of contiguous routines, that implements the licensing scheme.
2. Compile the application, linking in temporary code C_{EN} that generates a CRC of the associated object bytes and then compress and encrypt the associated object bytes. Then, dump the output C_{CT} to a file formatted for direct pasting into a C/C++ file. This file serves to initialize a *const char* array called *EncryptedRoutines*.
3. Now, insert runtime code, C_{DE}, that copies the contents of *EncryptedRoutines* to a buffer and then decrypts and decompresses the contents. Also, remove the original unencrypted code, C_{PT}, and the temporary code, C_{EN}.
4. Finally, recompile the application to link in C_{DE} and C_{CT} and to exclude C_{EN} and C_{PT}.

When the application starts up, it executes code C_{DE}, which copies the encrypted bytes from the *EncryptedRoutines* array into a dynamically created buffer. It then decrypts the buffer's contents using a public key, K_2, which the vendor provides, followed by decompression. The

routine computes a CRC of the decrypted block. If this CRC matches the original CRC, the application then constructs a pointer to a decrypted entry-point function and uses the pointer to execute that function. CRCs that differ from each other indicate the use of the wrong public key, and the application terminates, because executing the still-encrypted code could be catastrophic. As a result, any system calls made from within the encrypted code do not show up in any disassembled listing. A hacker can detect them only if he or she is willing to acquire and use an emulator and trace through the execution.

Compression of the code to be encrypted—before encryption—prevents the hacker from simply replacing the encrypted bytes with the decrypted bytes. Now, even if the hacker recovers the decrypted bytes, he cannot subvert or change the licensing logic and then encrypt and replace the existing encrypted bytes, because he does know the private key, K_1. You should also encrypt all messages that the encrypted routines use. Most compilers for Windows/DOS generate position-independent code that results in relative calls: Jumps and Branches. When the system compiles and links the original application, the compiler assumes that all code executes from code space. Subroutine calls compile into PC-relative call instructions with the correct displacements. However, the decrypted routines run out of dynamically allocated data memory, so the displacements are all wrong.

The solution is to force all subroutine calls from the decrypted routines to be made to absolute addresses. However, the absolute addresses of all linked functions depend on the compiler and the options used, the order of the linked libraries, and *the size of the user code*, which will change because of Steps 2 to 4. Three solutions to these problems are:

1. If necessary, pad the code with dummy instructions, such that the size of $C_{PT}+C_{EN}$ equals the size of $C_{DE}+C_{CT}$.
2. Link all subroutines called from within the decrypted code to fixed, user-specified addresses that will not change because of Steps 2 to 4.
3. Use an encrypted table of absolute-runtime-function addresses.

You can use the same concepts to protect code that implements a licensed feature. If you enter the correct public key, the decrypted code is usable. If you enter the wrong public key, the "decrypted" code remains encrypted, rendering that function unusable. You should use a unique private/public key pair for each licensed feature. **Listing 1** shows Borland 5.02 code fragments that illustrate the concept of running decrypted code from dynamically allocated memory. You can download **Listing 1** from the Web version of this article at www.ednmag.com. The programmer can decide the implementation details of the CRC generation, compression, and private/public key algorithms.

Is this the best Design Idea in this issue? Select at www.ednmag.com.

LISTING 1—ROUTINE FOR GENERATING ENCRYPTED CODE

```
#ifdef GENERATE_ENCRYPTED_CODE
// This is the licensing function we want to protect
void CheckForLicense (void *pRuntimePrintf, void *pRuntimeExit)
{
    // unscramble absolute addresses and create pointers to functions we will call
    void (*pExit)(int)              = (void (*)(int))((char *)          pRuntimeExit   - 0xD91E639A);
    int (*pPrintf)(const char *,...) = (int  (*)(const char *,...))((char *)pRuntimePrintf - 0x76532984);

    pPrintf ("Verifying license...\n");          // use a function call

    bool HaveLicense = false;
    // insert code here to determine if we have a license
    if (HaveLicense == false) pExit(0);
}
void End () { ; }       // just used to get size of the original code to be encrypted

// This code just scrambles for illustration. Use a true private/public key algorithm in your application.
void Encrypt (UCHAR *Buf, int szBuf)
{
    for (int i=0; i<szBuf; i++)
        Buf[i] = (UCHAR)((Buf[i] + i) ^ (~i));
}
void SaveBuf (UCHAR *Buf, int szBuf)
{
    FILE *fp = fopen ("code.cpp", "w");
    for (int i=0; i<szBuf; i++)
        fprintf (fp, "0x%02X, ", Buf[i]);
    fclose (fp);
}
void Generate (void)
{
    UCHAR Buffer[100];
    int szCode, nCBytes;
    szCode = (char *)End-(char *)CheckForLicense;  // get the size of the function
    memcpy  (Buffer, CheckForLicense, szCode);     // copy the original routine
    int CRC = GetCRC(Buffer, szCode);
    nCBytes = Compress(Buffer, szCode);            // user-supplied code. nCBytes = size after compression
    Encrypt (Buffer, nCBytes);                     // encrypt it
    SaveBuf (Buffer, nCBytes);                     // save output to disk
    printf ("code size: %d, CRC=%d\n", (char *)End-(char *)Encrypt, CRC);
}

#else if GENERATE_RUNTIME_CODE

// a table of encrypted absolute subroutine addresses linked for the runtime executable
void *AbsAddrTable[3] = { (void *)(char *)printf + 0x76532984),
                          (void *)(char *)exit   + 0xD91E639A),
                          (void *)0x81663874};    // can add random #'s to confuse hackers

const UCHAR EncryptedRoutine1[] = {
0xAA, 0x72, 0x13, 0xAA, 0x74, 0x98, 0xEB, 0x70, 0x3C, 0x99, 0x53, 0x18, 0xC1, 0x6A, 0xA2, 0xE7,
0xFA, 0x6J, 0x05, 0x53, 0x76, 0x97, 0x63, 0x6F, 0x8F, 0xFF, 0xFC, 0x0F, 0x96, 0x82, 0x3F, 0x43,
0x3F, 0x48, 0xFA, 0x51, 0xFF, 0xFE, 0x20, 0x58, 0x54, 0x50, 0x38};

#define CRC1  0x7682   // CRC of original code
#define SIZE1  43      // size of original code before compression
// This code just unscrambles for illustration. Use a true private/public key algorithm in your application.
void Decrypt (UCHAR *Buf, int szBuf)
{
    for (int i=0; i<szBuf; i++)
        Buf[i] = (UCHAR)((Buf[i] ^ (~i)) - i);
}
#endif

void main (void)
{
#ifdef GENERATE_ENCRYPTED_CODE
    Generate();                                    // generate the encrypted bytes, then quit
    exit(0);
#else if GENERATE_RUNTIME_CODE
    int nCBytes = sizeof(EncryptedRoutine1);       // get size of compressed code
    UCHAR  *Ram = new UCHAR[SIZE1];                // get a buffer from memory
    memcpy   (Ram, EncryptedRoutine1, nCBytes);    // copy the encrypted bytes
    Decrypt  (Ram, nCBytes);                       // decrypt the encrypted bytes
    Uncompress(Ram, nCBytes);                      // user-supplied decompression code

    if (GetCRC(Ram, SIZE1) != CRC1)                // check if have decrypted correctly
        exit(0);                                   // no, code is still encrypted, so quit

    // create a pointer to the code
    void (*pCheckForLicense)(void *, void *) = (void(*)(void *, void *))Ram;
    // execute the decrypted code
    pCheckForLicense(AbsAddrTable[0], AbsAddrTable[1]);

        :       :                                  // other user code
#endif
}
```

Capacitor improves efficiency in CPU supply

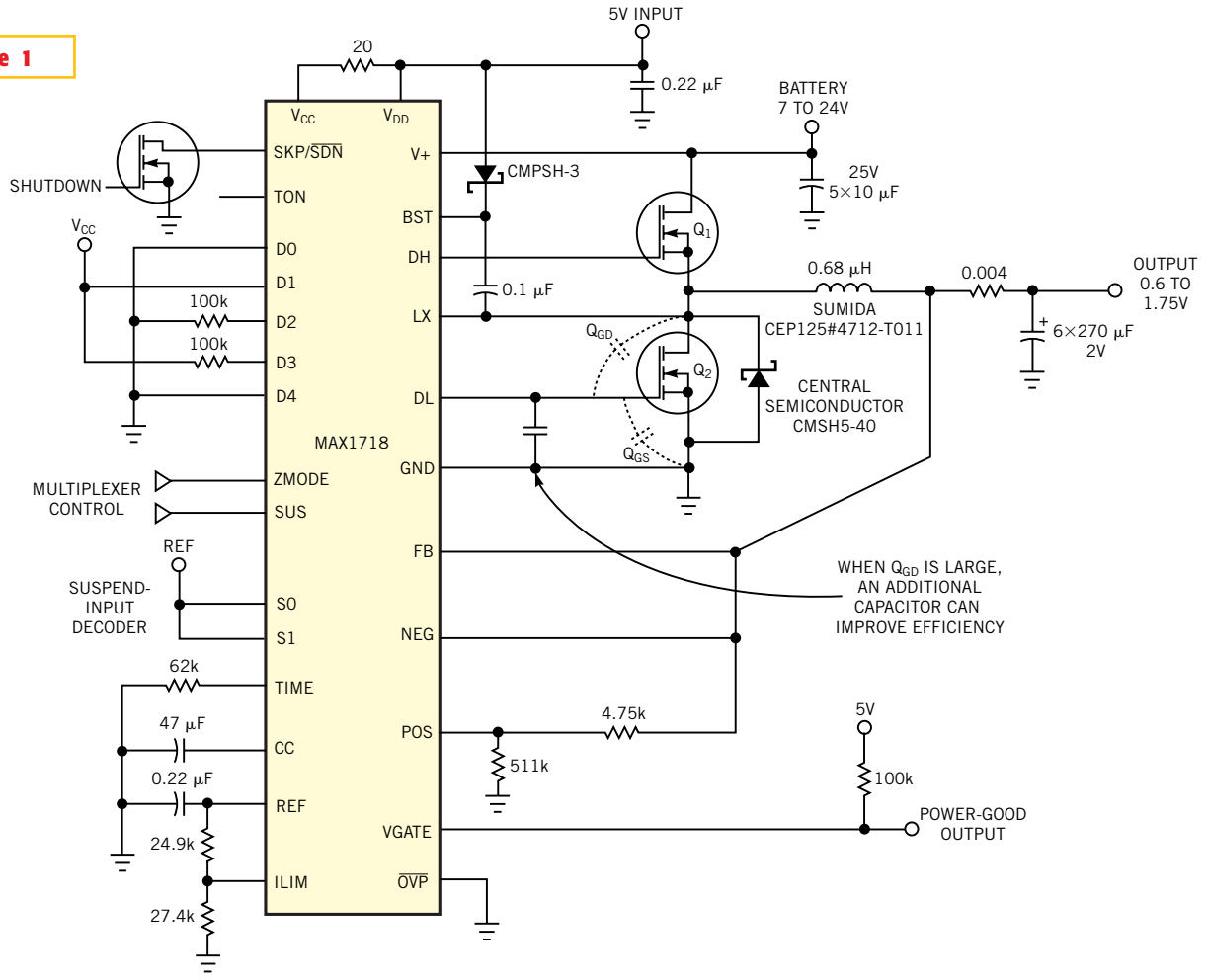
Masami Muranaka, Maxim Integrated Products, Tokyo, Japan

High efficiency is important for the dc/dc buck converters that supply high currents in notebook PCs. This efficiency extends battery life and minimizes temperature rise. A low-dissipation synchronous rectifier using an external MOSET provides this high efficiency. Synchronous rectifiers require special attention, however. Poor designs allow shoot-through current when the high- and low-side MOSFETs conduct simultaneously. Some designers believe that providing enough dead time between the turn-off of one MOSFET and the turn-on of the other can eliminate this problem, but using dead time is inadequate in some applications. **Figure 1** illustrates a step-down power supply in which a step-down controller, the MAX1718, provides the CPU's core supply. Recent CPU cores require a 1 to 2V supply rail at more than 20A of input current. The input-voltage range, on the other hand, is 7 to 20V. This scenario dictates a low duty cycle for the high-side MOSFET.

Obtaining high efficiency with a low duty cycle requires different types of MOSFETs for the high- and low-side devices, Q_1 and Q_2, respectively. Q_1 requires high switching speed even if its on-resistance is relatively high, but Q_2 requires low on-resistance even if its switch speed is relatively low. This combination of parameters allows no possibility of shoot-through current when Q_2 turns on, because Q_1's fast turn-off occurs first. Because Q_2's turn-off is slow, however, you must allow enough dead time before Q_1 turns on. The MAX1718 solves this problem by monitoring Q_2's gate voltage, thereby ensuring that Q_1 turns on only after Q_2 shuts completely off.

Now, consider a third condition leading to the possibility of shoot-through

Figure 1

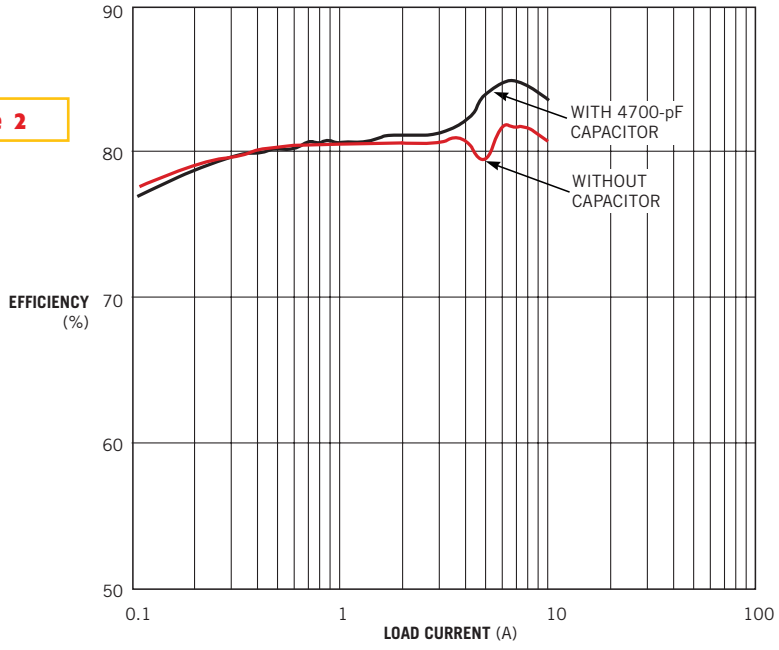


A buck-regulator IC drives external power MOSFETs to form a CPU-core power supply.

current: a rise in Q_2's gate voltage when Q_1 turns on because of high dV/dt at Terminal LX. That condition can appear even with sufficient dead time, because it involves high current flow into Terminal DL through Q_2's gate-drain capacitance, Q_{GD}. The MAX1718's ample current-sinking ability at DL solves this problem. Sometimes, however, if Q_2's gate-drain capacitance is large, the trace from DL is long, or both, you can eliminate the shoot-through current by adding a capacitor of several thousand picofarads between the gate and the source of Q_2. **Figure 2** shows that the addition of a 4700-pF capacitor improves the high-current efficiency of the circuit in **Figure 1** by a considerable margin. However, note that using a too-large gate-source capacitor, Q_{GS}, can increase the driver's losses.

Is this the best Design Idea in this issue? Select at www.ednmag.com.

Figure 2



NOTES:
BLACK LINE REPRESENTS TWO µPA2701s.
RED LINE REPRESENTS TWO µPA2700s.
V_{IN}=20V; V_{OUT}=1.3V.

The addition of a capacitor increases high-current efficiency by approximately 3%.