design ideas

Edited by Bill Travis

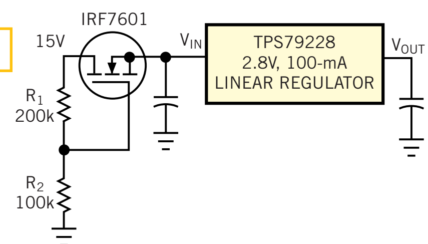
Extend the input range of a low-dropout regulator

Jeff Falin, Texas Instruments, Dallas, TX

Because of process limitations, all ICs have an input-voltage limitation. This limitation can be cumbersome when you try to step down a high supply voltage to a lower, regulated voltage using a dc/dc converter, such as a linear regulator. Adding a FET to the input of a linear regulator creates a dc/dc converter with a wider input-voltage range than the range of the regulator alone. The excess voltage and, therefore, power occurs in the FET.

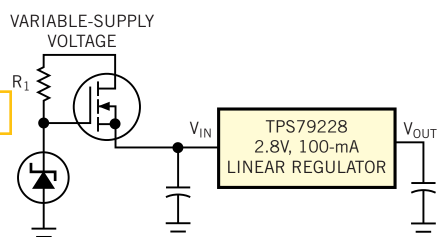
Figure 1 shows an IRF7601 n-channel MOSFET on the input of a TPS79228 2.8V, 100-mA, low-noise, high-power-supply rejection ratio low-dropout regulator. R_1 and R_2 provide a bias voltage to the gate of the MOSFET, and the load current determines the voltage at the source of the MOSFET. (In other words, the FET's on-resistance adjusts to accommodate the load current.) In this example, the maximum power-supply voltage is 15V, but the TPS79228 has a maximum recommended operating input voltage of 5.5V, so this design uses a MOSFET with a 20V breakdown voltage.

Figure 1



You can use a MOSFET switch to expand a low-dropout regulator's input-voltage range.

Figure 2



A zener diode provides fixed gate drive to the MOSFET when the input voltage varies significantly.

To determine the minimum bias voltage for the gate of the MOSFET, you need the MOSFET's drain-current, I_D,-versus-gate-source voltage, V_{GS}, data-sheet curves. For the IRF7601, the curves indicate that the device needs V_{GS} slightly below 1.5V for 100-mA output current. Because the maximum dropout voltage of the regulator is 100 mV at 100 mA, the regulator's input voltage must stay above 2.9V. Therefore, you must bias the gate of the MOSFET to at least 1.5V+2.9V=4.4V. Thus, when the MOSFET provides 100 mA, its source voltage does not drop below 2.9V. The maximum gate-bias voltage is simply the maximum recommended operating voltage for the regulator, or 5.5V. This voltage provides more than enough gate drive to provide the regulator's 1 μA of quiescent current during shutdown mode. Although you can bias the gate between 4.4 and 5.5V, this design uses a bias voltage of 5V to account for variations in the threshold voltage. Maximum power dissipation for the FET is 100 mA×(15V−2.9V)=1.21W. The IRF7601, in a Micro 8 package, can handle this power figure at an ambient temperature of 55°C.

The circuit in **Figure 2** is slightly more complicated but may be necessary if the input voltage varies significantly. A zener diode replaces the R_2 in **Figure 1** and provides a fixed gate drive to the MOSFET. You select the output voltage of the zener diode in a manner similar to that explained above. Either of the two methods is acceptable for creating a dc/dc converter with a wider input-voltage range than the converter IC allows. The single-MOSFET solution is the simplest and least expensive solution. The MOSFET biased with a zener diode is the best choice when the supply is unregulated.

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

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

Convert your DMM to a pH meter

Bill Donofrio, Nu-Products, Cordova, TN

It's often necessary to know the acidity of a solution to control a process. Even inexpensive pH meters can be relatively costly, and many of the inexpensive models have no output that you can readily connect to a computer interface. A simple solution to this problem is to attach a pH probe to a high-impedance input of an op amp and read the output with a digital voltmeter (**Figure 1**). Then, convert these readings to pH units using a calculator that can calculate the slope of a line. To calibrate the system, you can use pH standards. Generally, you would use three standards: 4-, 7-, and 10-pH units. These standards are inexpensive and available at any chemical-supply house. The calibration procedure is as follows:

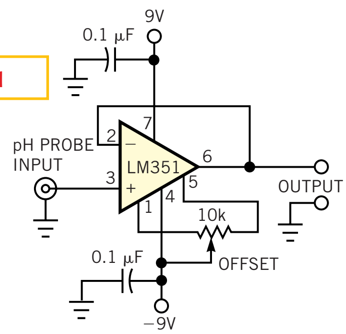
1. Short the input leads together and adjust the offset potentiometer such that the output reads 0 mV.
2. Place the pH probe in each standard and record the output (in millivolts) for each standard.
3. Enter the values in your calculator and determine the slope of the line.

Figure 2 shows a typical plot of pH versus millivolt output. (Note that pH is temperature-dependent; you have to recalibrate if the temperature changes.)

If you want to measure directly in pH units, you can use the circuit in **Figure 3**. At approximately 24°C, the equation for the slope of the line is $Y=-0.017X+7$. To obtain readings in tenths of volts, you multiply the equation by 10. The new equation is $10Y=-0.17X+70$. The circuit in **Figure 3** comprises three sections. The voltage using the LM351 provides the high-impedance input. The inverting amplifier using one-half of the LM353 controls the slope. It multiplies the LM351's output by −0.17. The other half of the LM353 functions as a summing amplifier and controls the Y intercept by adding 70 mV to the input signal. When you build this circuit, solder the BNC directly to the op amp's input pin. This connection prevents slightly conductive pc boards from affecting the impedance levels of the probe. Another point to remember is to remove the pH probe from the unit when power is off.

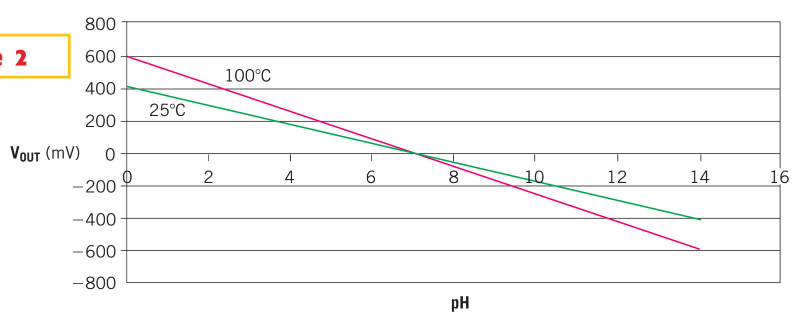
To calibrate the circuit, first short the inputs together and adjust the offset potentiometer to obtain 0-mV output from the LM351. To calibrate the circuit for pH units, place the pH probe into a pH standard. Measure the voltage at the output of the LM351. Multiply this voltage by 0.17 and adjust the slope potentiometer until the output of the second op amp reads this inverted (negative) value. Then, connect the meter to the output of the circuit and adjust the Y-intercept potentiometer until the circuit yields the pH of the standard you use. (For example, a pH of 10.1 reads 0.101V.) To tweak the circuit, place the pH probe in other standards and adjust the Y-intercept potentiometer. Note again, if the temperature changes, you must recalibrate. The accuracy of this circuit is generally ±0.1 pH units. When you order pH probes, you should order low-impedance units. This circuit uses Cole-Parmer (www.colepalmer.com) U-59001-65 probes.

Figure 1



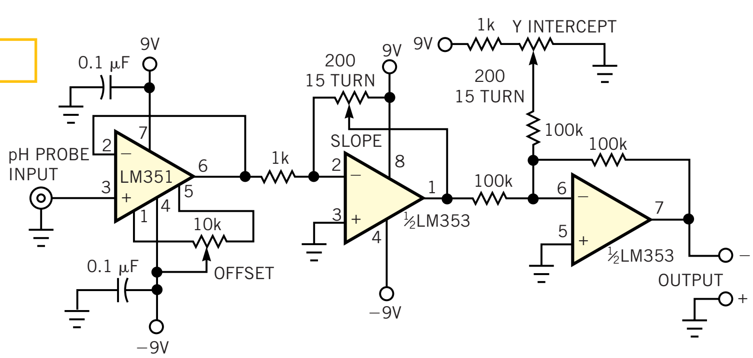
This simple pH tester uses a digital voltmeter and a calculator to determine the slope of a line.

Figure 2



Available pH probes produce a linear slope of output voltage versus pH.

Figure 3



With this circuit, you can directly measure pH units.

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

AVR microcontroller makes improved motor controller

Anthony Di Tommaso, Cranberry Township, PA, and George Simonoff, Petersburg, OH

The circuit in **Figure 1** provides a novel method of reading the pulse train using an Atmel (www.atmel.com) AVR processor, from a typical radio-controlled receiver, and to determine the velocity of a motor. To capture the pulse train from a typical receiver, you need an external interrupt that triggers based on a rising and a falling edge. Three timers are also necessary: one 16-bit, free-running timer to determine the period of the input pulse and two 8-bit timers con-

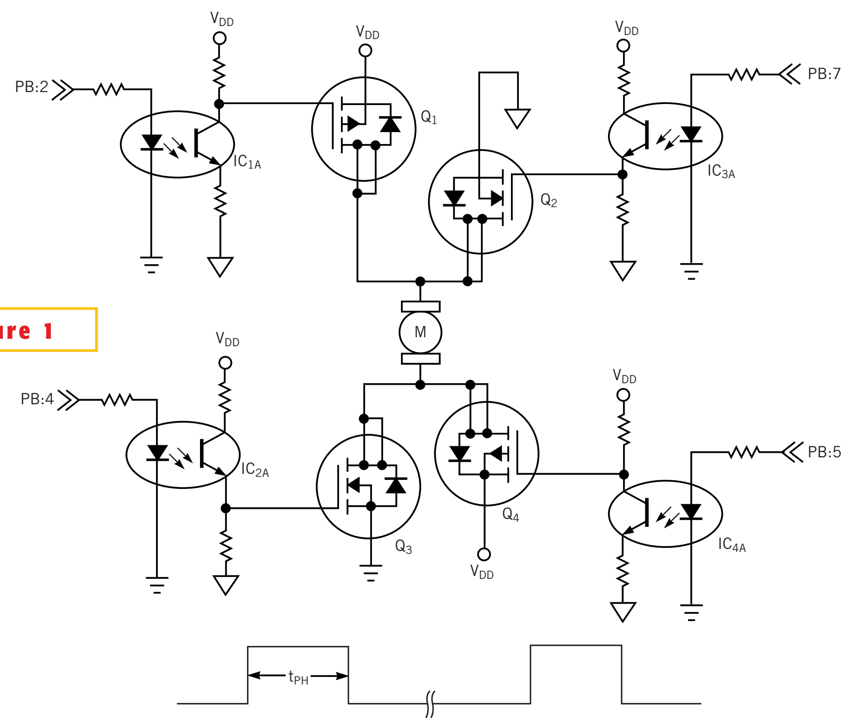
LISTING 1—MOTOR-CONTROL C LISTING

```
#include <io.h>
#include <interrupt.h>
#include <signal.h>

unsigned char c17;                    //Used to indicate the present direction of motion.
unsigned short s23;                   //Used to hold the 16 bit Timer 1 count.

SIGNAL(SIG_INTERRUPT4)
{
volatile unsigned char ptr;
unsigned char lower_byte;
unsigned char upper_byte;

        lower_byte = 0;               //Clear bytes in anticipation of receiving
        upper_byte = 0;               //new data.
        c17 |= 0x04;                  //Set global for control is with receiver.
        ptr = inp(EICR);              //Get present state of EICR.
        if ((ptr & 0x01) == 0)        //If falling edge capture enabled, then ...
        {
                c17 |= 0x08;                  //Indicate arithmetic operation can proceed
                lower_byte = inp(TCNT1L);     //Move low part of count to low byte.
                upper_byte = inp(TCNT1H);     //Move high part to high byte.
                s23 = upper_byte;             //Create 16 bit word that represents period
                s23 <<= 8;                    //of pulse.
                s23 += lower_byte;
        }
        else                                  //If rising edge capture enabled, then ...
        {
                c17 &= 0xF7;                  //Indicate arithmetic operation should wait
                outp(0,TCNT1H);               //Reset timer counter register.
                outp(0,TCNT1L);               //When writing to 16 bit timer, must write
        }                                     //to upper register first.
        ptr = inp(EICR);                      //Change state of last bit of EICR
        ptr ^= 1;                             //in order to be able to trap on the
        outp(ptr,EICR);               //opposite edge.
}

int initialize(void);
int receiver_cntrl(void);

int main(void)
{

        initialize();                         //Call initialize in order to set
                                              //up registers and peripherals.
        s23 = 0;                              //Initialize useful variables,
        c17 = 0;                              //some that are global and some
                                              //that are local to main.
        while (1)
        {
                receiver_cntrl();
        }
}
int initialize(void)
{
        //Timer 0 Setup Code
        outp(0x61,TCCR0);       //No prescale to clk, enable PWM, clear PWM output
        outp(0,OCR0);           //Set output compare register to 0.
        outp(0,TCNT0);                  //Set timer counter to 0.
        //Timer 1 Setup Code
        outp(0,TCCR1A);         //Set clk source as clk/8, compare timer to overflow
        outp(0x0A, TCCR1B);     //Set output compare register to 0xFF00.
        outp(0xFF, OCR1AH);     //Not used, however.
        outp(0,OCR1AL);         //Set output compare register to 0.
        outp(0,TCNT1H);         //Set timer 1 counter to 0.
        outp(0,TCNT1L);
        outp(0x10,TIMSK);       //Enable timer overflow interrupt.
        //Timer 2 Setup Code
        outp(0x61,TCCR2);       //No prescale to clk, enable PWM, clear PWM output
        outp(0,OCR2);           //Set output compare register to 0.
        outp(0,TCNT2);                  //Set timer counter to 0.
        //External Interrupts Setup Code
        outp(0x10,EIMSK);       //Enable interrupts for externals 0 through 4.
        outp(0x03,EICR);        //The rising edge between two samples of INT4
                                //generates an interrupt.
        outp(0x80,ACSR);        //Disable comparator.
        //Port B Setup Code
        outp(0xff,DDRB);        //Set initial direction of port pins to output.
        outp(0,PORTB);                  //Set initial state of pins.
        //Port D Setup Code
        outp(0x0,DDRD);         //Set initial direction of port pins to input.
        outp(0xff,PORTD);       //Set initial state of pins.
        //Port E Setup Code
        outp(0,DDRE);           //Set initial direction of port pins to input.
        outp(0,PORTE);                  //Set initial state of pins.

        sei();                  //Globally enable all active interrupts.

        return(0);
}
int receiver_cntrl(void)
{
volatile unsigned char ptr;

        c17 &= 0xF7;            //Enable receiver mode
        s23 -= 0x01EE;          //Set up map range
        if ((s23 >= 0xF0) && (s23 <= 0x110))    //See if s23 is in the deadband.
        {                                       //If so, then stop the motor.
                ptr = inp(PORTB);       //Get present state of output drivers.
                ptr &= 0xDB;    //Block movement by setting PB:2 and PB:5 to 0
                outp(ptr,PORTB);        //Update state of driver.
                outp(0,OCR0); //Set PWM0 to 0% disabling forward.
                outp(0,OCR2); //Set PWM2 to 0% disabling reverse.
                c17 &= 0xFC;   //Modify global to indicate direction is stop.
        }
        else if ((s23 > 0x110) && (s23 <= 0x1FF))       //See if s23 is in fwd range
        {                                               //If so, then proceed with fwd.
                if ((c17 | 0xFD) == 0xFD)               //If global indicates present
                {                               //direction is not reverse. then ...
                        outp(0,OCR2); //Block reverse by shutting down PWM2.
                        outp(ptr, PORTB);       //Update value of port B.
                        ptr = (volatile unsigned char)s23; //Assign movement to ptr.
                        outp(ptr, OCR0);        //Assign PWM to new movement value.
                        ptr = inp(PORTB);       //Get present state of port B.
                        ptr |= 0x04;    //Indicate that the direction is forward.
                        outp(ptr,PORTB);        //Update the value of the port.
                        c17 |= 0x01;    //Set global bit to indicate forward.
                        c17 &= 0xFD;   //Set global bit to not indicate reverse.
                }
        }
        else if ((s23 >= 0) && (s23 < 0xF0))    //See if s23 is in rev range
        {                                       //If so, then proceed with rev.
                if ((c17 | 0xFE) == 0xFE)       //If global indicates present
                {                               //direction is not forward, then ...
                        outp(0,OCR0); //Block forward by shutting down PWM0.
                        ptr = inp(PORTB);       //Get present value of port B.
                        ptr &= 0xFB;    //Stop forward by setting driver to 0.
                        outp(ptr,PORTB);        //Update value of port B.
                        ptr = -(volatile unsigned char)s23;//Assign movement to ptr.
                        outp(ptr, OCR2);        //Assign PWM to new movement value.
                        ptr = inp(PORTB);       //Get present state of port B.
                        ptr |= 0x20;    //Indicate that the direction is forward.
                        outp(ptr,PORTB);        //Update the value of the port.
                        c17 |= 0x02;    //Set global bit to indicate reverse.
                        c17 &= 0xFE;   //Set global bit to not indicate reverse.
                }
        }
        return(0);
```

figured as PWMs (pulse-width modulators) for driving the motor in both forward and reverse. Finally, two digital outputs act in concert with the PWMs to move the motor. You can find all these features in an AVR microcontroller. Assuming that you use a typical transmitter and receiver, such as a pair from Futaba (www.futaba.com), the range of pulse width, t_{PH}, should vary from 1 to 2 msec for full reverse and full forward, respectively. If the Timer 1 clock-source frequency is 500 kHz, the number of counts possible between full reverse and full forward is 500, beginning at 500 for a pulse width of 1 msec and ending at 1000 for a pulse width of 2 msec. A pulse width of 1.5 msec identifies no-input or full-stop conditions. **Listing 1** provides an example of this motion-control application.

Subtract 494 ($1EE) from the value of Timer 1, captured after the falling edge of the pulse, assuming that the pulse is positive-going and that, at the rising edge of the pulse, the value of the counter resets. The range of values between full reverse and full forward becomes 6 to 506 ($6 to $1FA). Forward motion is $101 to $1FA, with $101 providing the least forward motion and $1FA providing the most forward motion. Reverse motion ranges from $FF to $6, with $FF providing the least amount of reverse motion and $6 providing the most amount of reverse motion. If negated, the reverse range becomes $01 to $FA. If you use the lower byte of the adjusted Timer 1 value directly to set the output comparison registers OCR0 or OCR2 of Timer 0 or Timer 2, you can use as much as 98% of the range of input values to the PWM. You can choose which PWM-configured timer to use based either on where the value occurs in the range or on the upper byte. Note that when the upper byte is 1, the direction is forward; when the upper byte is 0, the direction is reverse. You can download the C-based **Listing 1** from the Web version of this Design Idea at www.ednmag.com.

Figure 1



An Atmel AVR microcontroller provides a novel method of controlling a motor.

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

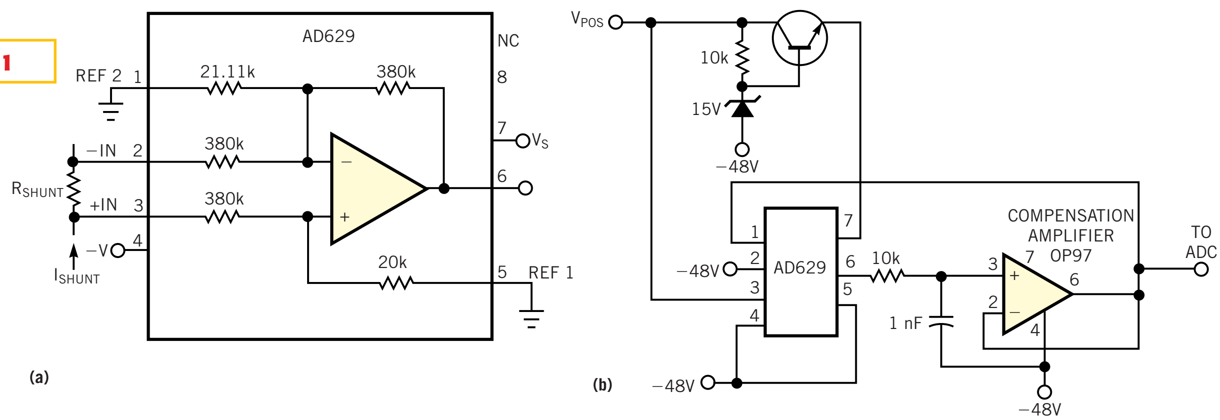
Precision circuit closely monitors –48V bus

Paul Smith and Jim Staley, Analog Devices, Wilmington, MA

Ever-greater complexity in communications systems has spurred a need for power-supply management. POTS (plain-old telephone systems) obtain power from –48V supplies backed by arrays of batteries in central offices and distributed throughout copper lines. Although nominally –48V, the voltage on the lines can vary from –40 to –80V, and the voltage is subject to surges and fluctuations. Supply regulation at the source has little effect on remote voltage levels. Equipment failures resulting from surges, brownouts, or other line faults may sometimes be undetectable. Capturing power-supply information from remote communications equipment requires precise measurement of the voltages—sometimes in outdoor temperatures. High-common-mode, voltage-difference amplifiers have been useful in monitoring current. However, you can also use these versatile components as voltage dividers, enabling remote monitoring of voltage levels as well.

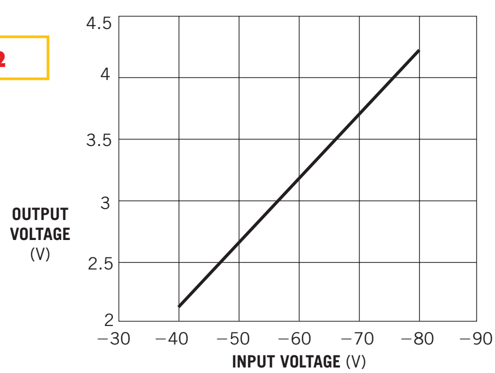
Figure 1a shows the basic circuit connections when you use a difference amplifier for high-line current sensing. With the addition of a few parts, you can operate this amplifier in a negative-supply system. **Figure 1b** shows a precision monitor using just two ICs and deriving its power from the –48V supply. All you need to power the circuit are a transistor and a zener diode to reduce the supply voltage for the amplifiers. The AD629

Figure 1



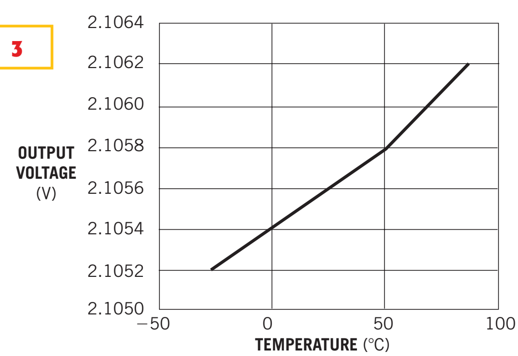
A basic monitoring circuit (a) uses a difference amplifier for high-line current sensing; the complete circuit (b) requires just two ICs.

Figure 2



The circuit in Figure 1b produces an output that's a linear function of the negative supply voltage.

Figure 3



Temperature drift in the circuit of Figure 1b is negligible.

shown in **Figures 1a** and **1b** is a self-contained, high-common-mode, voltage-difference amplifier with unity gain. Connected as shown, however, it reduces the differential-input voltage by approximately 19, thus acting as a precision voltage divider. You need an additional amplifier for loop stability. The circuit features several advantages over alternative approaches. The laser-trimmed divider resistors exhibit essentially perfect matching and tracking over temperature. Linearity errors from −40 to −80V are nearly immeasurable. **Figures 2** and **3** show the linearity and temperature-drift curves for the monitoring circuit.

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

PLD code reveals pc-board revisions

Clive Bolton, Bolton Engineering Inc, Melrose, MA

The PLD (programmable-logic-device) code in **Listing** 1 implements a pc-board-level revision-detection system that detects whether PLD pins are shorted together on a pc board. It is often advantageous to field a single PLD programming file that works for several generations of physical hardware. The PLD needs to understand what the board revision is, so that it can enable or disable functions, pins, or both to external circuitry. If a designer has not placed physical straps to indicate a pc-board-revision level from the start, it may be difficult to add them later. In PLD families that have no integral pin-pullup or -pulldown resistors, redefining previously unused pins as inputs means that these pins float, either causing erratic operation or indicating an improper pc-board-revision level.

TABLE 1—PATTERN-GENERATOR ACTIONS

State	REV_OUT line	REV_IN[x] lines
0	Tristated	Drive low
1	Tristated	Tristated
2	Drive with pattern	Look for pattern
3	Tristated	Tristated

The software module generates a short, simple pattern, such as a square wave, onto a driver pin, REVO_OUT. The input-detection pins, REVI_IN[x], look for this pattern. If they detect the full pattern on a pin, the module indicates that the pins are connected by setting the respective Q[x] high. If they do not detect the full pattern, the module sets the Q[x] line low. The pattern generator avoids leaving the revision-detection inputs floating by alternately driving and tristating the REVO_OUT and REVI_IN[x] lines (**Table 1**). If you drive the circuit at more than a few megahertz, the parasitic pin capacitance of a few picofarads is sufficient to ensure that the REV_I[x] pins stay low, even while they are tristated. When the detection cycle is finished, the COMPLETE line goes high.

The module generates the required hardware from two compile-time parameters: LPM_WIDTH and CHANNELS. LPM_WIDTH sets the number of times the detection cycle runs (for example, 5 bits yields 2^5, or 32 cycles), and CHANNELS sets the number of strap inputs. Note that this function does not include tristate buffers; you must instantiate them at the design top level. The REVO_EN and REVI_EN pins enable the REVO_OUT and the REVI_IN[x] tristate buffers, respectively. The PLD code is written in Altera's (www.altera.com) high-level design language (AHDL); you can directly compile it into any of Altera's PLDs. An implementation of the design with the parameters set to the default values takes 16 logic cells in an Altera EP1K50BC256-3—less than 1% of the device—and runs at rates as high as 185 MHz. Although the code is written for Altera's devices, the design structure and flow are readily translatable into VHDL or Verilog.

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

LISTING 1—AHDL CODE FOR REVISION DETECTION

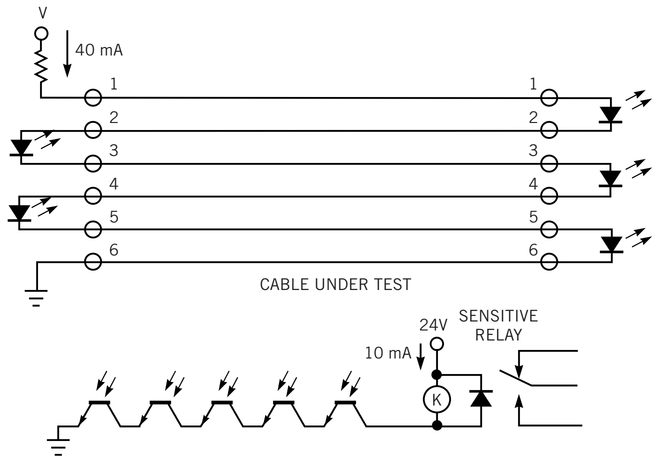
```
% TITLE rev_detect.tdf: AHDL file                                   %
% path: f:\project3\altera\rev_detect.tdf                       %
% (c) 2002 Bolton Engineering, Inc. All rights reserved         %
% size of 16 LCs in 1K10-3, 185MHz clk operation                %
% 02/18/02. Code started, tested, and entered into RCS          %

INCLUDE "lpm_counter.inc";

PARAMETERS   (
      LPM_WIDTH    = 5,  -- in bits; sets the length of time for detection
      CHANNELS     = 1           -- number of strap inputs (1 or more)
);

SUBDESIGN rev_detect
(
      clock                         :     INPUT;              -- global clk
      aclr                          :     INPUT;

      revi_in[CHANNELS-1..0]   :     INPUT;             -- strap inputs for all
"inputs"
      revi_en                             :     OUTPUT;              -- tristate
enable

      revo_out                      :     OUTPUT;             -- strap output
      revo_en                             :     OUTPUT;              -- tristate
enable

      q[CHANNELS-1..0]         :     OUTPUT;             -- 1 if connected, 0 if
N.C.
      complete                      :     OUTPUT;
)
VARIABLE
      phase_cntr                    :     LPM_COUNTER WITH (      LPM_WIDTH
= 2,

      LPM_DIRECTION = "UP");
      seq_cntr                      :     LPM_COUNTER WITH (      LPM_WIDTH
= LPM_WIDTH,

      LPM_DIRECTION = "UP");
      dffe_q[CHANNELS-1..0]    :     DFFE;
      complete_unr                  :     NODE;
BEGIN
      -- at 0, drive strap "inputs" only
      -- at 1, drive nothing
      -- at 2, drive strap output only
      -- at 3, drive nothing
      phase_cntr.clock  = clock;
      phase_cntr.aclr          = aclr;
      phase_cntr.cnt_en = !complete;

      revi_en             = DFF( (phase_cntr.q[] == 0) OR complete, clock, !aclr,
VCC);
      revo_en             = DFF( (phase_cntr.q[] == 2) OR complete, clock, !aclr,
VCC);

      seq_cntr.clock           = clock;
      seq_cntr.aclr            = aclr;
      seq_cntr.cnt_en          = !complete_unr AND phase_cntr.cout;
      complete_unr             = seq_cntr.cout;
      revo_out                 = seq_cntr.q[0];

      complete                 = DFF(complete_unr, clock, !aclr, VCC);

      FOR i IN 0 to (CHANNELS-1) GENERATE
            dffe_q[i].clk      = clock;
            dffe_q[i].prn      = !aclr;
            dffe_q[i].ena      = revo_en AND !complete;
            dffe_q[i].d        = dffe_q[i].q AND (revi_in[i] XNOR revo_out);
            q[i]               = dffe_q[i].q;
      END GENERATE;

END;
```

Simple method tests cables

Jim Keith, Bell-Mark Technologies, Dover, PA

Engineers have long known how to test a cable for continuity by simply connecting all conductors in series and checking with an ohmmeter. This method is sometimes impractical, however, because it cannot check for short circuits. The method shown in **Figure 1** solves the short-circuit problem. Connecting LED indicators at each shorting loop provides a visual indication. The beauty of this scheme is that any short circuit causes at least one LED to extinguish. You can then diagnose the malfunction by the visual signature of the LEDs. Taking the method one step further, let the LEDs be the emitters of photocouplers. Connecting the phototransistors in series provides a simple go-no-go test that requires no visual observation. Note that the 2V LED forward-voltage drop presents a challenge when you're testing a cable with a large number of conductors, so be sure to apply a high enough voltage.

Figure 1



This simple cable-testing method tests for continuity as well as short circuits.

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