You can improve a capacitive-sensor circuit with a modulator and an RF transmitter (Reference 1) by modifying the modulator portion to obtain better accuracy. More improvements result from adding a “negatron” circuit, a configuration that uses equivalent negative capacitance. Reference 2 gives some insight into the uses of negative impedance. The circuit in Figure 1 works with the RF transmitter of Reference 1 (Figure 2) as follows: The modulator uses op amp IC1 in a standard flip-flop configuration (with the addition of the bias resistor R4, because of single-supply operation). The negatron portion of the circuit uses op amp IC2. The output frequency, f, of the modulator without the negatron is a function of the time constant \( \tau = \frac{R_3C_S}{R_4} \). Thus, the frequency is a function of the physical value of the input (for example, pressure or humidity). The potential accuracy and stability of the modulator in Figure 1 are greater than those of the modulator in Reference 1 because of the low input capacitance and temperature coefficient of the LTC1124 op amp and the high stability of the \( R_4-R_6 \) voltage threshold.

Adding the negatron portion in Figure 1 increases the relative sensitivity of the frequency modulator. The equivalent capacitance of the negatron is \( C_N = C_i \frac{R_2}{R_1} \). Assuming the physical value of the input causes the change \( \Delta C \) (compared with the sensor’s initial capacitance, \( C_S \)), the relative output-frequency change without the negatron (for small \( \Delta C \)) is \( \Delta f/f_o = \Delta C/C_S \). With the negatron added, the equation is: \( \Delta f/f_o = \Delta C/(C_S + C_i \frac{R_2}{R_1}) \). The result is higher relative sensitivity because of the reduction in the denominator value. The value of \( C_N \) is ap...
proximately 100 pF for the circuit in Figure 1. In other words, a given change in the input value causes a greater relative-frequency deviation. Note that adding the negatron changes the equivalent time constant: \( \tau = R \left( C_e - \frac{C_1}{(R/R_1)} \right) \). You can make value adjustments in \( R_3 \) to obtain the desired initial frequency, \( f_0 \).

Also, note that the measured values of \( f_0 \) and \( \Delta f \) may differ from the calculated ones because of parasitics and the input capacitance of the op amp.

**References**


---

**Get buck-boost performance from a boost regulator**

*Tom Gross, Linear Technology Corp, Milpitas, CA*

The SEPIC (single-ended, primary-inductance-converter) topology is generally a good choice for voltage regulators that must produce an output voltage that falls in the middle of the input-voltage range, such as a 5V output from a 2.7 to 6V input. The topology has some disadvantages, however. The efficiency of a SEPIC circuit fares worse than that of buck and boost regulators, and SEPIC designs often involve the use of large, complex magnetic components, thereby complicating the design task. Figure 1 shows a simple and efficient alternative topology. When the input voltage is lower than the output voltage, the circuit operates as a normal boost regulator. The inductor, \( L_1 \), stores energy when switch \( Q_1 \) is on and the boost diode, \( D_1 \), is reverse-biased. While \( D_1 \) is off, the output capacitor, \( C_1 \), delivers the load current. When \( Q_1 \) turns off, \( L_1 \) reverses its polarity, thereby forward-biasing \( D_1 \). \( L_1 \) then charges \( C_1 \) and delivers current to the load. The inductor voltage adds to the input voltage to generate the output voltage.

The low-battery comparator in IC1, which usually checks battery levels, monitors the output voltage through its LBI pin. IC1’s internal comparator output switches low (sinking current). This action turns the p-channel \( Q_1 \) fully on, cre-

---

![Figure 1](image-url)

This circuit can both boost and step down the output voltage, depending on whether the input voltage is lower or higher than the output voltage.
ating a low-impedance path to the output. When the input voltage is the same value as or greater than the output voltage, the circuit functions like a linear regulator. In this case, the internal comparator’s output assumes a high-impedance state. The voltage at the gate of Q1 begins to pull up through R1, a 220-kΩ resistor, so Q1 begins to turn off. This action forces the output voltage to decrease, and the comparator eventually again switches states from high to low (sinking current). The comparator’s low state causes the output voltage to rise again, and the cycle repeats. Thus, the circuit begins to operate as a linear regulator, with Q1 acting as the pass transistor.

The circuit can also disconnect the input-to-output current path, unlike a conventional boost regulator. The shutdown signal connects to the gate of Q2, a logic-level, p-channel MOSFET, as well as to IC1’s shutdown (SD) pin. When the shutdown signal goes low, it turns off IC1 and turns on Q2. This action delivers VIN (via R2, a 100kΩ resistor) to the gate of Q1, thereby turning off the transistor. Hence, the shutdown operation disconnects the input-to-output current path. **Figure 2** shows how the efficiency of this linear-boost regulator depends on its mode of operation. When the input voltage is lower than the output voltage, the efficiency of the regulator is that of a boost regulator.

The efficiency of the circuit in Figure 1 depends on whether the circuit acts as a boost converter or a linear regulator. When the input voltage exceeds the output voltage, the circuit operates as a linear regulator in which efficiency is approximately $V_{OUT}/V_{IN}$.


---

**Circuit transmits ARINC 429 data**

Steve Woodward, University of North Carolina, Chapel Hill, NC

The ARINC (Aeronautical Radio Inc) 429 specification defines the air transport industry’s hardware and protocol standards for the transfer of digital data between avionics systems. Circuitry that can implement elements of the 429 spec is often an essential part of control and sensor electronics intended for the aviation environment. ASIC chips for this purpose are commercially available, but they typically require nonstandard power supplies (for example, ±15V) and wide parallel interfaces. Therefore, it’s sometimes inconvenient to accommodate them in 5V microcontroller-based designs. The circuit in **Figure 1** serves the Tx (transmit/output) portion of the 429 spec. The design implements a high-speed ARINC-429-compliant transmit function using a single 5V supply rail and 74HC series chips.

The physical transmission medium for the 429 standard is 78Ω shielded, twisted-pair cable that uses a complementary, differential bipolar RZ (return-to-zero) waveform (**Figure 2**). The voltages are the net differentials that the biphasedrive develops: For example, the differential is 10V when you drive the Data A signal in **Figure 1** to 5V and the Data B signal to −5V. In addition to the signal levels, a 429 system must closely control the rise and fall times to conform to the specification. This control limits both in-

---

**TABLE 1—FORMAT OF SPI BITS**

<table>
<thead>
<tr>
<th>Byte</th>
<th>ARINC field</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Label</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>2</td>
<td>LS data</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>S</td>
<td>D</td>
</tr>
<tr>
<td>3</td>
<td>Data</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>4</td>
<td>Parity+MS data</td>
<td>OPB</td>
<td>SSM</td>
<td>SSM</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>5</td>
<td>Spacer byte</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

L…L = ARINC label (most-significant bit transmitted first)
SDI = ARINC source/destination identifier
N…N = 19 data bits (may include SDI and SSM fields; transmit least-significant bit first)
SSM = ARINC sign/status matrix
OPB = Odd parity bit (makes total count of “1” bits in the 32-bit word odd)
tra-cable signal crosstalk and EMI radiation that might interfere with sensitive aircraft communication and navigation systems. The operation of the transmitter in Figure 1 centers around IC1, a 4-bit × 16-word FIFO memory. The serial ARINC bit stream comes from the microcontroller’s synchronous serial-peripheral interface (SPI); the C input of IC1 buffers the data stream. In addition, the D input of IC1 serves as a buffered ARINC-enabled bit (the microcontroller’s J port, bit 2). When low, this bit disables the ARINC transmitter logic and permits other system peripherals to use the SPI hardware.

Bits A and B of IC1 go unused. The 100-kHz ARINC high-speed baud rate comes from the 1-MHz reference supplied to the IC3A divide-by-10 circuit. The 100-kHz signal drives IC1’s shift-out (SO) pin and the IC2 pulse gate. The presence of bits in the IC1 FIFO (indicated by QR=1) resets the ARINC RDY bit in IC3B and enables IC3A. If IC1’s D bit (Pin 10) is also high, it gates the 100-kHz square wave to the IC4 multiplexer. This action causes the sequential gating of 5, 0, and 5V onto the A/B data-output signals in ARINC-compatible waveforms. The LRC network at the output ensures compliance with the 429 requirements for rise and fall times. The circuit must process five 8-bit SPI bytes to generate each 32-bit, 429-compliant output word. Table 1 shows the format of the SPI bits. The first four bytes in Table 1 combine to form a 32-bit ARINC 429 word. The 32-bit word, reading from right to left, starts with byte 1 (again, reading from right to left), then tacks on byte 2, and so on.

The ARINC spec imposes rigid requirements on waveforms.

Digital frequency dividers usually use flip-flop stages that connect the Q pin to the D data-input pin of the following stage. This configuration creates a binary waveform that you can feed back to the input. You can divide any integer lower than \(2^N\) with minimal stages, where \(N\) is the number of stages. These dividers can easily select one frequency from 100 for a receiver. However, as the applied clock rate approaches the ratings of the devices, decoding spikes appear. As a result, you'd be ill-advised to use the dominant pulse in such a waveform for clocks or strobes. The divider in Figure 1 uses a ring configuration, and the stages connect Q-to-D, without using the Q output, to provide a binary sequence. Consequently, the circuit can divide only by \(N+1\), but it produces a clean waveform at an applied clock rate that's substantially higher than you can apply to conventional binary dividers using flip-flops from the same process families.

If the last Q in a cascade of 74xx174 flip-flops connects to the D input, the loop becomes a shift-register ring counter. Moreover, the circuit can operate at a clock rate higher than that of any other configuration. Unfortunately, when you turn on the power or ground the reset pin, all Q outputs go low and remain low when you apply the clock. To circumvent installing a preset circuit that must operate at a high clock rate, you can place one NOR gate with its propagation delay in the loop. This addition ensures that the divider always starts and continues to function properly. Because this gate receives inputs from all stages at once, it features parallel carry and has the properties of a parallel-carry counter. For simplicity, Figure 1 does not show the reset line, and you can delete the broken feedback lines, providing that you ground the corresponding input pins. You can take the output from any Q pin and use an additional stage, such as Q5, for a buffer. Although the output of the NOR gate resembles one of the Qs, you should not use it as an output.

Figure 2 shows the timing diagram for the divider. Section A of the diagram shows the state of D0 and the Q outputs in the start-up condition before the first and second clock pulses arrive. Note that when the power turns on or when the reset pin connects to ground, the Q outputs are all low, and D0 is high. D0 then transfers to Q0 on the rising edge of the first clock pulse. Section B of the diagram shows the output for a repetitive sequence starting with Q0. Section C is an expanded representation of the end of the sequence. Note that Q2 falls after the clock pulse rises. Then, with all Q outputs low, D0 rises a short time later to allow the sequence to repeat. D0's transfer can take place only after the next clock pulse rises. This factor creates an additional time slot to make the total \(N+1\). Section C shows the propagation delay attributable to the NOR gate—approximately 10 nsec for most logic-circuit families and less than 5 nsec for the F and S series. This propagation delay is the only additional delay in the loop. A comparison of the output waveforms with those from a 74xx90 divide-by-5 counter shows prominent decoding spikes from this counter. The \(N+1\) divider had no spikes and operates at a much faster clock rate.

Precise timing control is paramount in data acquisition and analysis and especially in digital-signal processing. The easiest way of maintaining timing control in a PC is to use delay loops. The disadvantage of this implementation is that the delay loop’s elapsed time depends on the system’s operating frequency. Hence, a program works accurately with a single operating frequency, but you must modify it for other frequencies. You can use add-on cards to achieve timing control, but they’re costly, and, for simple operation, they remain underused. This Design Idea explores another way to obtain timing control. You can implement a stable delay loop with the aid of the PC’s speaker logic. In PC/AT architecture, a 16-bit 8255 timer/counter IC is available, and the IC’s operating frequency of 1.1931817 MHz is fixed across the entire range of PC families. You can use this fixed-frequency feature to implement a delay loop.

In PC/AT architecture, Counter 0 serves as a system timer and to maintain the time of day. Counter 1 generates pulses and serves as the DRAM refresh-rate generator. The PC uses Counter 2 for sound generation through the PC’s speaker. Only Counter 2 is available for an implementation of the delay loop. You can enable or disable Counter 2 by setting bit 0 of the 8255’s port B (in other words, port 61h) to one. Once enabled, Counter 2 automatically decrements by one count every 0.8380958 sec. Before calling the delay routine, the software computes the total of Counter 2’s count required for the desired delay (desired_delay_time). A “while” loop repeatedly reads back Counter 2’s data from its input port (042h) in a 2-byte operation.

Because the count depends on a fixed operating frequency across the entire PC family, you can use this method to implement a precise and stable timing loop. You can achieve delays of approximately 20 μsec to 54.9 msec.

The two bytes combine to produce 16-bit data (counter_data). This data helps to compute the elapsed counts (count_elapsed); as soon as the required count is reached, the software exits this loop. The routine disables Counter 2 by setting bit 0 of the 8255’s port B to 0 before returning to the calling program. Thus, it achieves the desired delay.

The software is in Turbo C++ and assembly language and was tested in MS-DOS mode on a 450-MHz Pentium II computer.

Listing 1—Delay-Loop Routine

```c
#include <conio.h>
define counter_clock_time 0.000839055855
#define Address_of_Read_State 0x378
#define Address_of_Write_State 0x00
#define desired_delay_time 0.8380958
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
#define desired_delay_time_in_milliseconds 838
#define desired_delay_time_in_microseconds 838095
#define desired_delay_time_in_seconds 0.000838095855
```
Voice feedback enhances engineering calculator
Alexander Bell, Infosoft International Inc, Rego Park, NY

The screen shot in Figure 1 represents a Microsoft Excel 2002 worksheet designed to implement VFI (voice-feedback interface) for an engineering calculator. The voice-interface technique has both practical and educational aspects. It automates the common task of finding the values of two resistors for a given ratio. It also demonstrates the latest advances in natural-language programming technology with an example of the technology’s actual implementation in CAD/CAE systems. A single user-defined function RR (resistor ratio) encapsulates both the computation engine and the VFI. The function uses VBA (Visual Basic for Applications) with the code placed in the standard code module of Excel File (Listing 1). You can download Listing 1 and the Excel worksheet from the Web version of this Design Idea at www.ednmag.com.

Cell A2 serves for data (ratio) entry; cell B2 contains the user-defined formula 5RR(A2). In automatic calculation mode, every time you enter a new ratio value in A2, the system recalculates R1, R2, and Relative Error.

LISTING 1—CODE MODULE FOR VOICE FEEDBACK

```vbnet
Option Explicit

Function RR(ByVal Ratio As Variant) As String
    Dim Error As Double
    Dim R1 As Double, R2 As Double
    Dim CurrentRatio As Double
    Dim ErrorLimit As Double
    Dim MaxError As Double

    CurrentRatio = Ratio
    ErrorLimit = 0.05
    MaxError = 0.01

    If CurrentRatio < 1 Then
        R1 = CurrentRatio + 1
        R2 = CurrentRatio
    Else
        R1 = CurrentRatio
        R2 = CurrentRatio + 1
    End If

    Error = Abs(R1 - R2) / R1
    If Error > ErrorLimit Then
        Error = MaxError
    End If

    RR = Format($R1$2 + Error, "#.##") & "Ω"
End Function
```

The core search algorithm, which contains outer and inner loops, sequentially
tests each pair of values, R₁ and R₂, to find the best approximation of the target ratio. In other words, the algorithm tries to minimize the absolute error: (ABS (R₁/R₂ - Ratio)). The values of R₁ and R₂ come from the E24 EIA-standard series, but you can apply the same algorithm to any other standard series, such as E48, E96, or E192. The VFI sends the status-notification message in a verbal form instead of showing the Message Box. The VFI uses the built-in Excel 2002 Speech Object with the following syntax:

```
Application.Speech.Speak <String Variable or Constant>, True,
```

where `<String Variable or Constant>` contains the actual spoken text, and the second property is set to True for asynchronous mode. **Listing 2** is an example of voice-error notification in the case that you enter non-numeric data as the ratio (data-validation error message):

```
LISTING 2—VOICE-ERROR NOTIFICATION

IfNot IsNumeric(Ratio) Then
    Str_MSG5 = "Wrong ENTRY: PLEASE, ENTER THE NUMERIC VALUE."
    Exit Function
End If
```

Using the technique is simple. Open Excel File, switch to the Visual Basic Editor window (press Alt-F11), add the standard module, and paste the code from **Listing 1**. From the Debug menu item, choose Compile VBA Project, save the file with any name, and close the Visual Basic Editor window. Make sure you activate the text-to-speech tools (in the menu: Tools—Speech—Show Text To Speech Toolbar). Check whether you are in automatic- or manual-calculation mode (in the menu: Tools—Options—Calculation). Set the mode to automatic; otherwise, you'd have to use the F9 key to force a new calculation every time you enter a new ratio value. Choose any cell (for example, A2 in **Figure 1**) for the ratio (data-entry cell) and another cell (for example, B2) to display the results. Enter the formula =RR(A2) into cell B2. Now, every time you enter a new ratio value into cell A2, the system automatically calculates and displays R₁, R₂, and Relative Error, and sends the status voice notification. The notification is either “OK” to confirm the successful completion of the calculation or an error notification in the case of a data-validation or computation error. Note that some macros in Microsoft Office applications could result in potentially dangerous and harmful actions, and some may contain viruses. You use the macros at your own risk without warranties of any kind.

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