May 20, 2005

Signal Filtering with the MAXQ7654

Based on the high-performance, 16-bit RISC MAXQ20 core, the MAXQ7654 offers a 16-channel, 12-bit, analog-to-digital converter (ADC), and dual, 12-bit, digital-to-analog converters (DACs). In addition to analog components, the MAXQ7654 has a wealth of digital peripherals, including a CAN controller, an SPI interface, and four 8/16-bit timers usable as counters or programmed for pulse-width modulation. With 128kB of code space, the MAXQ7654 can handle many embedded mixed-signal applications.

The application presented in this article demonstrates the MAXQ7654's mixed-signal features. This microcontroller uses the first of its two DACs to output a noisy sinusoid, a low-frequency sinusoid injected with random noise. The DAC output is tied to one of the ADC input channels for voltage measurement. The input samples are run through a simple finite-impulse response (FIR) filter to attenuate the high-frequency components of the signal, resulting in a nice, smooth sinusoid that is output on the second DAC.

The application presented in this article demonstrates the mixed-signal features of the MAXQ7654. This microcontroller uses the first of its two DACs to output a noisy sinusoid, a low-frequency sinusoid injected with random noise. The DAC output is tied to one of the ADC input channels for voltage measurement. The input samples are run through a simple finite-impulse response (FIR) filter to attenuate the high-frequency components of the signal, resulting in a nice, smooth sinusoid that is output on the second DAC.

With its wealth of analog and digital peripheral support, there are many interesting applications that demonstrate the capabilities of the MAXQ7654. This article focuses on the device's signal-filtering abilities, emphasizing the ADCs, DACs, and the hardware multiply-accumulate unit. Using the IAR compiler and the MAXQ7654 evaluation kit (EV kit), a sample application demonstrates how to filter a noisy sinusoid and output the clean, low-frequency signal underneath.

Download: Source code, project files, and schematics that support this article.

Integrated Analog Functions and Peripheral Components Enable Signal Filtering

The MAXQ7654's 16-channel, 12-bit ADC completes a conversion in as little as 16 clock cycles. At the 8MHz maximum clock rate, it completes 500,000 samples per second. Applications can multiplex up to 16 input pins for single-ended analog measurement, or up to 8 pins for differential signal measurement. The ADC also measures temperature—the MAXQ7654 contains an internal temperature sensor for on-chip (die) temperature readings.

The MAXQ7654 includes a hardware multiply-accumulate unit for signal-processing applications. It can multiply two 16-bit numbers in a single cycle, and has an optional accumulate function that operates in signed or unsigned modes. This facilitates the implementation of FIR and IIR filters; each coefficient of the filter requires as little as three machine cycles to process, plus some overhead each time the filter is invoked.

A JTAG debug engine, which is common to the MAXQ platform, provides read and write access to registers and memory while applications are running on the real hardware. JTAG also eliminates the need for expensive emulators. Major C compiler vendors such as Rowley, IAR, and Python support the MAXQ7654 and its debugging capabilities.

A new peripheral for the MAXQ platform is a controller area network (CAN) 2.0B interface, a common network protocol in industrial and automotive applications. Capable of bit rates up to 1Mb per second, the MAXQ7654's CAN controller supports 15 message centers. Interrupts notify the system when messages are received or sent.

An SPI™ interface supports slave or master mode and 8- or 16-bit data transfers. SPI is commonly found in small integrated circuits such as programmable battery chargers, digital potentiometers, DACs, ADCs, and memory chips.

The MAXQ7654 has four multipurpose timers. These timers are configurable for 8- or 16-bit counting, and support auto-reload for periodic interrupts, pulse-width modulation, capture, and compare functions.

Software Architecture for the Filtering Application

The noisy sinusoid is output on the first DAC in a timer interrupt to ensure that output samples are transmitted at regular intervals. However, the code to generate a sinusoid involves complex floating-point calculations, and is computationally expensive. Plus, a sinusoid is periodic, repetitive data. Recalculating sinusoid data that will not change over time is a waste of resources. Therefore, upon startup, the application precomputes an array of sinusoid data.

After the sinusoid data is initialized, the timer is configured to generate periodic interrupts. In the timer interrupt code, a pseudorandom number generator computes noise, which is simply added to the clean sinusoid value. The result is passed to the DAC for output conversion.

To keep the demonstration code simple, the analog input signal is sampled in the same timer interrupt used to output the noisy signal. When an input sample is read, it is run through a simple FIR filter, which is implemented in assembly language for maximum efficiency. The filtered sample is then output on the second DAC. An oscilloscope is used to compare the two DAC outputs. One sinusoid is jagged and noisy, while the other sinusoid appears clean, with a slight phase delay due to the length of the FIR filter.

Generating and Sampling a Noisy Sinusoid

The timer interrupt code shown below starts with a precomputed sinusoid value and converts it to a noisy sinusoid value.

```
        sample =  static_sin_data[sinindex++];
        sinnoise = ((sinnoise ^ 0x5C) * 31) + 0xabcd;
        thisnoise = sinnoise;
        if (thisnoise & 0x01)
        {
            thisnoise = thisnoise & 0x1ff;
        }
        else
        {
            thisnoise = -1 * (thisnoise & 0x1ff);
        }
        sample += thisnoise;
        if (sample < 0)
            sample = sample * -1;
        if (sample > 4095)
            sample = 8192 - sample;

        DACI1 = sample;         // Send value to DAC #1
        if (sinindex >= SIN_WAVE_STEPS)
            sinindex = 0;
```

The variable sinnoise stores pseudorandom noise, which can be positive or negative. The noise factor is added to the value of the pure sinusoid, and the resulting noisy sinusoid value is simply assigned to the DACI1 register for digital-to-analog conversion.

Reading a sample from the DAC is nearly as simple. After selecting the input pin for the ADC to sample, software can either poll a busy bit or enable an interrupt to be notified that the conversion is complete. The sample code uses the polling technique.

```
    inputsample = ADC_Convert_Poll(AIN0 | START_CONV | CONTINUOUS);
    ...
    unsigned int ADC_Convert_Poll (unsigned int Control_Reg)
    {
      ACNT = Control_Reg;              // Set the ADC parameters
      while( ACNT_bit.ADCBY == 1);     // Wait till ADC is not busy
      return ADCD;                     // Return the ADC result
    }
```

Remember that the sampling rate for the ADCs on the MAXQ7654 is 500ksps. With an 8MHz clock, the code spends only 16 clock cycles waiting for a conversion to complete.

Designing a Simple Digital Filter

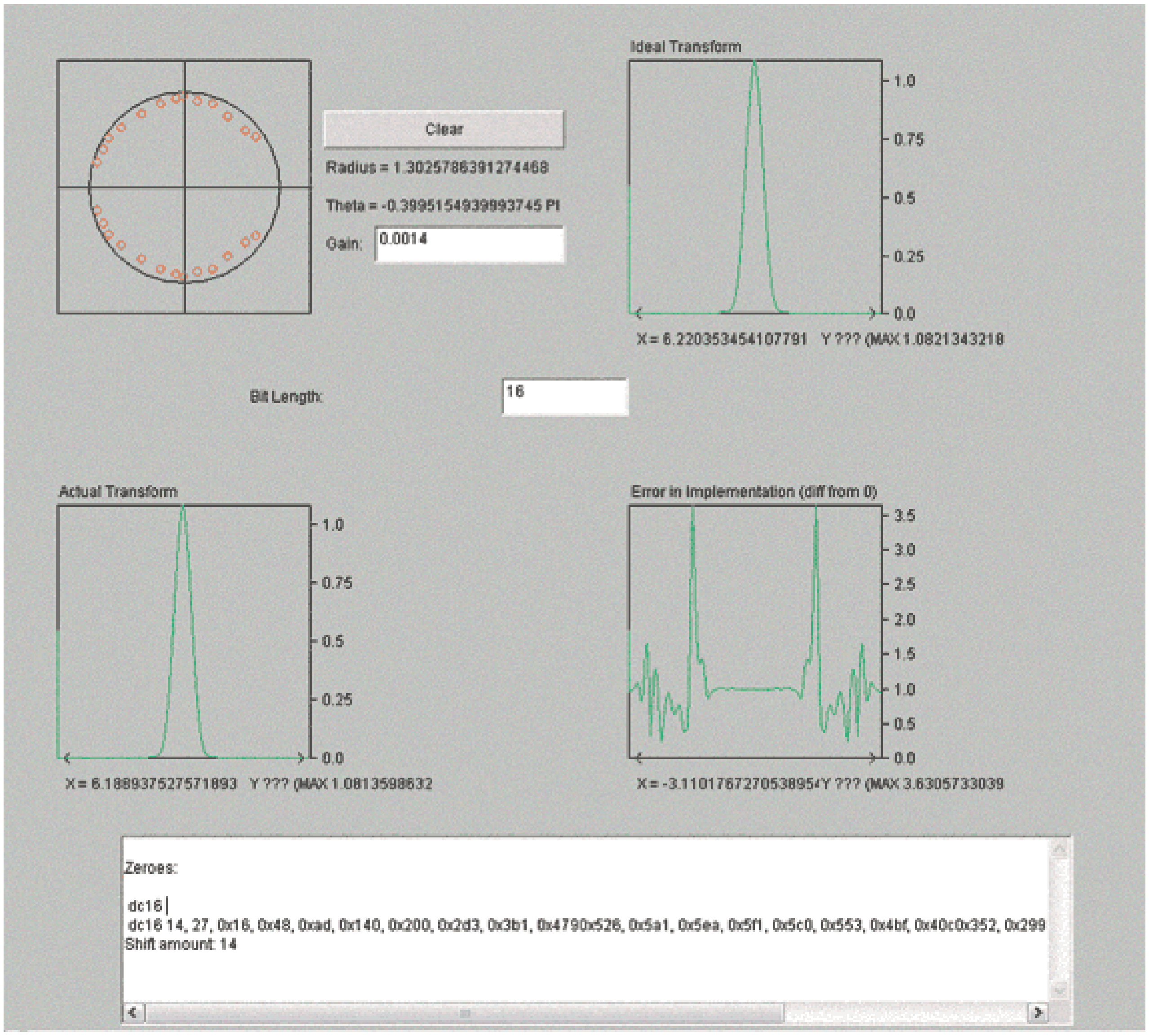
The noisy waveform generated in this application contains one strong low-frequency signal and a large amount of high-frequency noise. A simple lowpass filter cleans this signal.

A general FIR filter is an equation of the form:
$Y = \Sigma A_n X_n$

where A_n represents the filter coefficients, X_n is the previously sampled inputs, and Y is the current output of the filter. The filter coefficients determine the frequency response of the filter, or how the different frequency components are attenuated or accentuated.

A Java applet (available in the source code distribution for this article) was used to generate filter coefficients based on a pole-zero plot (**Figure 1**). The applet produces a set of high-precision floating-point filter coefficients. However, because the MAXQ7654 has a 16-bit hardware-multiply accelerator, the floating-point coefficients need to be converted to fixed-point coefficients with 16-bit precision. This conversion introduces error to the ideal filter transform. Therefore, the Java applet also outputs the actual transform realized by the fixed-point coefficients and a graphical representation of the error. Note that while the applet supports both poles (which accentuate frequency components) and zeros (which attenuate frequency components), the demonstration code only uses zeros. Infinite-impulse response filters (containing both poles and zeroes) can be implemented with additional software support.

The text box at the bottom of the applet produces the 16-bit fixed-point filter coefficients, plus the number of decimal places in the fixed-point numbers.



Radius = 1.3025786391274468

Theta = -0.3995154939993745 PI

Gain: 0.0014

X = 6.220353454107791 Y ??? (MAX 1.0821343218

X = 6.188937527571893 Y ??? (MAX 1.0813598632

X = -3.1101767270538954Y ??? (MAX 3.6305733039

```
Zeroes:

 dc16
 dc16 14, 27, 0x16, 0x48, 0xad, 0x140, 0x200, 0x2d3, 0x3b1, 0x4790x526, 0x5a1, 0x5ea, 0x5f1, 0x5c0, 0x553, 0x4bf, 0x40c0x352, 0x299
Shift amount: 14
```

Figure 1. This illustration shows the output of the Java applet that produces the filter coefficients. The applet produces the ideal transform, actual transform, error, and 16-bit filter coefficients.

Implementing an Efficient Digital Filter

This section discusses how the fixed-point coefficients are implemented in a real digital filter. The digital filter algorithm is coded in assembly for maximum performance. This allows application developers to optimize the filter routine based on the requirements of an individual application. Squeezing in an extra cycle or two can make a significant impact on the maximum filter length and sample rate that an application can support.

This demonstration makes two key decisions to maximize the filter's efficiency. First, this application uses an unrolled filter loop. This increases the code size of the algorithm, but produces a very fast filter, requiring three cycles and three codewords per coefficient. This design decision is not unrealistic. A high-quality filter designed with a Kaiser window might produce a filter with 250 coefficients, yielding a total code-size cost of 750 words. On a machine with 65,536 words of code space, this is a valid decision if filter performance is important.

The second key decision to improve filter efficiency is to dedicate 256 words of RAM to a circular buffer that stores the previous input data (the X_n portion of the general filter equation). If the filter has 250 coefficients, the application must store 250 previous input values anyway, so dedicating 256 words of RAM to the filter is not wasteful. The benefit of this decision is that the MAXQ's base-offset pointer can be used to create a circular buffer in hardware. The filter algorithm does not need to check if a pointer has reached the start of a data buffer, because the pointer automatically rolls over the buffer boundary. The following is the code for the digital filter.

```
filtersample:
    push  DP[1]                ; preserve IAR's software stack
    push  DPC                  ; probably needs this preserved
    move  AP, #0               ; select accumulator 0
    sub   #2048                ; normalize the input sample
    move  DPC, #10h            ; DP[0] byte mode, BP word mode
    move  BP, #W:sampletable   ; start of the sample table
    move  DP[0], #B:sampleindex  ; point to sample current index
    move  AP, #1               ; select accumulator 1
    move  ACC, @DP[0]          ; get current table index
    move  Offs, ACC            ; put it in the offset register
    add   #1                   ; increment the current index
    move  @DP[0], ACC          ; restore the table pointer
    move  @BP[Offs], A[0]      ; store the current sample
    move  MCNT, #22h           ; signed, accum, clear regs first

filterloop:
    ;
    ; Unroll the filter loop for speed.
    ;
    move  MA, #0x16
    move  MB, @BP[Offs--]
    move  MA, #0x48
    move  MB, @BP[Offs--]
    ...
    move  MA, #0x7
    move  MB, @BP[Offs--]
    move  MA, #0x2
    move  MB, @BP[Offs--]
          nop

    move  A[2], MC2            ; get MAC result HIGH
    move  A[1], MC1            ; get MAC result MID
    move  A[0], MC0            ; get MAC result LOW
```

The code first normalizes the input sample. Because the MAXQ7654 has a 12-bit ADC, the input values range from 0 to 4095. To use the digital filter, the input values should be normalized to -2048 to +2047, so subtraction by 2048 is performed ($2048 = 2^{11}$). Once the pointer to the input samples is initialized and the current input sample is stored, the code executes the filter.

The MAXQ's hardware multiply-accumulate unit is easy to use. Filter coefficients and input samples are loaded into the multiplier registers, and the multiplication result is ready after one clock cycle. The input samples are read from the BP[Offs] pointer, and the filter coefficients are hard-coded, taken directly from the output window in Figure 1 (reproduced here):

```
dc16 14, 27, 0x16, 0x48, 0xad, 0x140, 0x200, 0x2d3, 0x3b1, 0x479, 0x526, 0x5a1, 0x5ea, 0x5f1,
 0x5c0, 0x553, 0x4bf, 0x40c, 0x352, 0x299, 0x1f4, 0x163, 0xf0, 0x97, 0x58, 0x2e, 0x15, 0x7, 0x2
```

The "14" in the first line means that the numbers in the filter have 14 places after the radix point, and must be shifted 14 places to the right when the filter is complete. The "27" means that there are 27 coefficients in the filter. Following those control values, the coefficients are listed starting with A^0 (0x16, 0x48, 0xad, . . .).

After the filter algorithm is complete, the accumulated result is ready in the multiply-accumulate unit's registers MC0, MC1, and MC2. The result must be shifted to compensate for the fixed-point radix.

To change filters used by the application, simply alter the code underneath the filterloop label. For each coefficient output by the Java applet, add the instruction pair:

```
move  MA, #COEFFICIENT_n
move  MB, @BP[Offs--]
```

Also, make sure to change the shift count if necessary.

Results

The simple filter does its job perfectly. **Figure 2** shows an oscilloscope capture of the two MAXQ7654 DACs. Notice the phase shift on the clean output signal due to the length of the FIR filter.

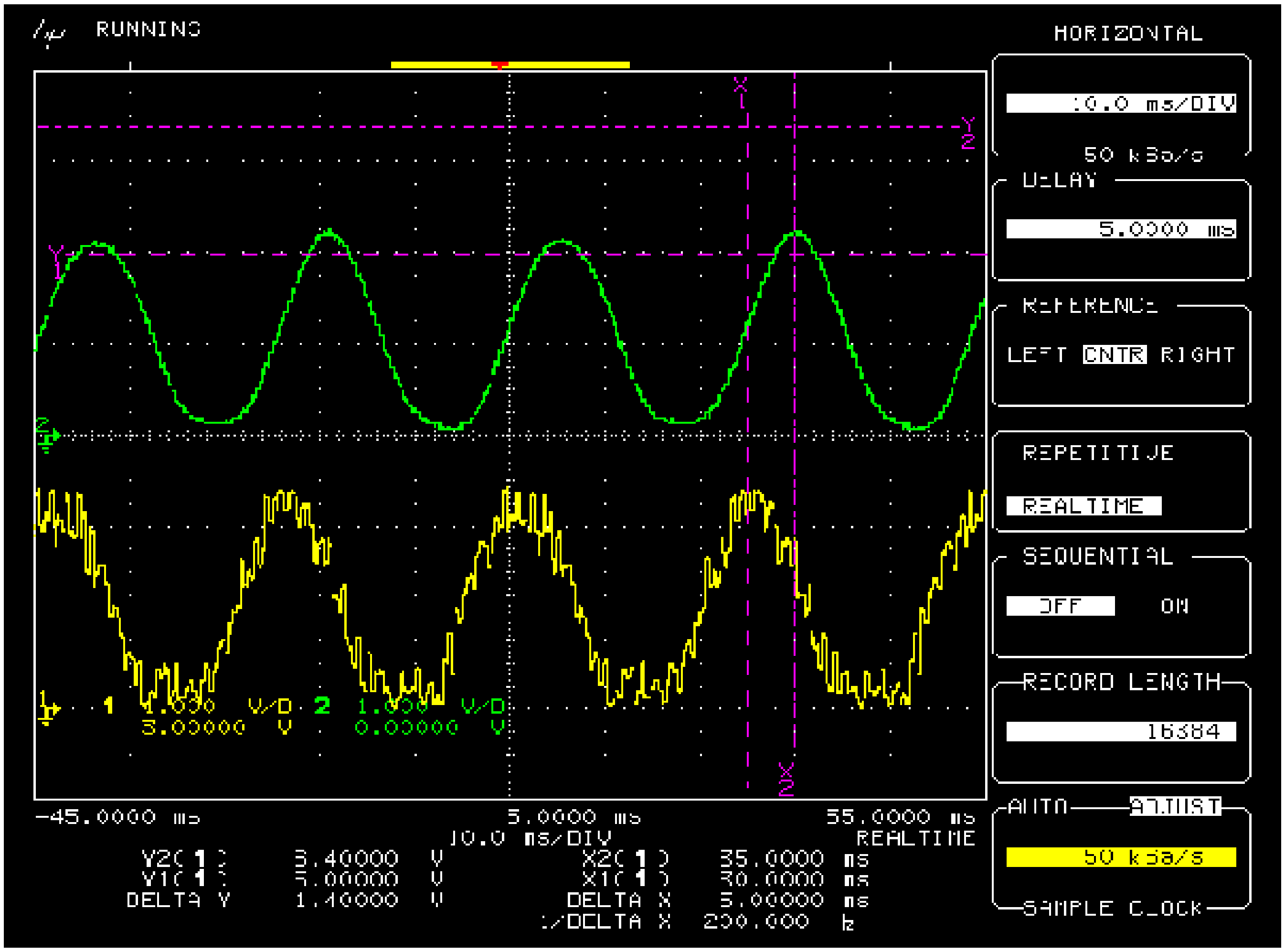


Figure 2. The bottom signal is the noisy DAC output from the MAXQ7654. It is sampled, filtered, and output as the top signal.

Evaluation Kit

Schematics for the MAXQ7654 EV kit are available with the source code distribution for this application. The kit board has many options for exploring the MAXQ7654 microcontroller. Jumpers select supply voltages and peripheral configurations, and every pin is accessible on the board. The MAXQ7654 EV kit (see **Figure 3**) also integrates the JTAG hardware, so no external board is required for loading or debugging.

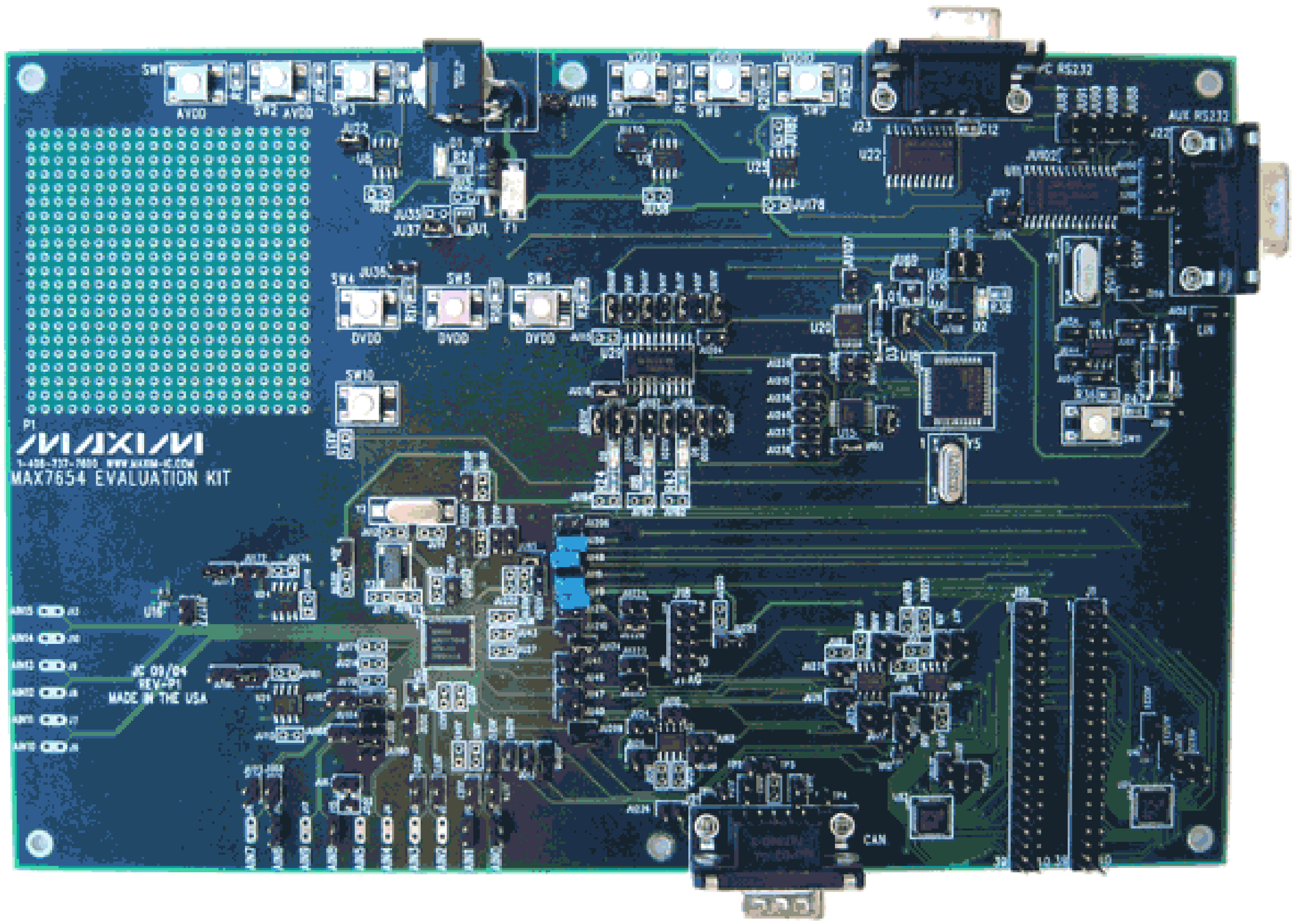


Figure 3. With plenty of I/O, pushbuttons, and a prototyping area, the MAXQ7654 EV kit is an ideal platform for evaluating the MAXQ7654.

Conclusion

As we have seen, the MAXQ7654 is a high-performance, mixed-signal microcontroller with a wide range of applications. With the MAXQ7654's simple demonstration code and integrated designs for maximum performance, the device offers designers easy-to-use elements for their signal-filtering application.

SPI is a trademark of Motorola, Inc.
MAXQ is a trademark of Maxim Integrated Products, Inc.

This article appears in the MER Vol 5.