Keywords: MAXQ, FFT, DSP, spectrum, ADC, DAC

Mar 23, 2006

APPLICATION NOTE 3722

Developing FFT Applications with Low-Power Microcontrollers

Abstract: As low-power microcontrollers (µCs) begin including peripherals that were formerly reserved for larger microprocessors, ASICs, and DSPs, new opportunities for computing complex algorithms at low power levels are becoming possible. This article describes a Fast Fourier Transform (FFT) application (developed using a low-power µC) that includes a single-cycle hardware multiplier.

This FFT application computes, in real time, the spectrum of an input voltage (V_{IN} in **Figure 1**). To accomplish this, an analog-to-digital converter (ADC) samples V_{IN} and transfers those samples to the µC. The µC then performs a 256-point FFT on the samples to obtain the spectrum of the input voltage. For testing purposes, the µC calculates the magnitude of the spectrum and transfers the results to a PC where they are displayed in real time.

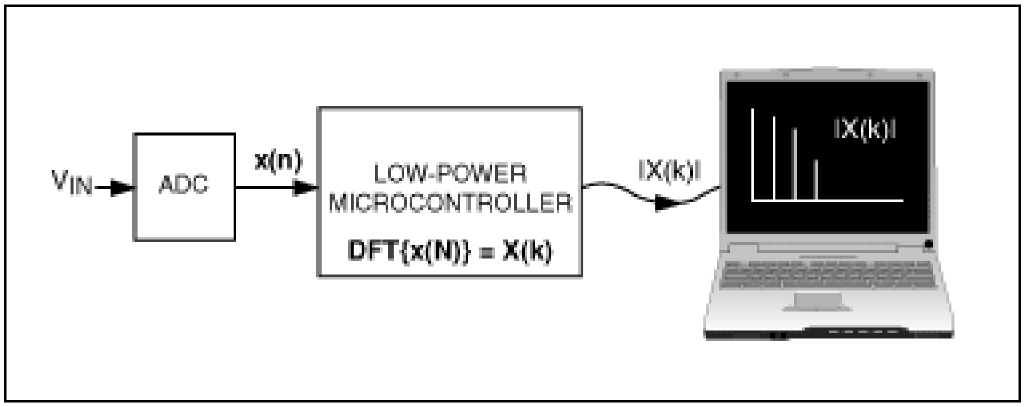

Figure 1. The spectrum of an input voltage is calculated using an FFT application.

The firmware for this FFT application is written in C for a 16-bit, low-power µC in the MAXQ2000 family. Interested readers can download (ZIP, 2.4kb) the firmware, along with the circuit schematic for this project.

Background

To determine the spectrum of the sampled input signal, we must compute the Discrete Fourier Transform (DFT) of the input samples. The DFT is defined as:

$$X(k) = \sum_{n=0}^{N-1} X(n) e^{\frac{-j2\pi kn}{N}} \quad \text{for } 0 \le k \le N-1 \qquad \text{Eq. 01}$$

where N is the number of samples, X(k) is the spectrum, and x(n) is the set of input samples. Expanding this summation using Euler's identity, and separating the input samples and spectrum into their real and imaginary components, yields the following equations:

$$X_{Re}(k) = \sum_{n=0}^{N-1} [X_{Re}(n)\cos(\frac{2\pi kn}{N}) + X_{Im}(n)\sin(\frac{2\pi kn}{N})] = \sum_{n=0}^{N-1} [X_{Re}(n)\cos(\frac{2\pi kn}{N})] \qquad \text{Eq. 02}$$

$$X_{Im}(k) = \sum_{n=0}^{N-1} [X_{Re}(n)\sin(\frac{2\pi kn}{N}) - X_{Im}(n)\cos(\frac{2\pi kn}{N})] = \sum_{n=0}^{N-1} [X_{Re}(n)\sin(\frac{2\pi kn}{N})] \qquad \text{Eq. 03}$$

The second term in the summation of equations 2 and 3 disappears because the input samples consist entirely of real numbers. Assuming we have N samples, computing equations 2 and 3 directly requires $2N^2$ multiplications and 2N(N - 1) additions. Therefore, our DFT with 256 input samples would require 131,072 multiplications and 130,560 additions. Enter the FFT!

Many FFT algorithms exist. The common radix-2 algorithm used in this implementation continuously decomposes the DFT into two smaller DFTs. For this to be possible, N must be a power of 2. The steps involved in the radix-2 FFT algorithm can be summarized with the butterfly computations illustrated in Figure 2. Observing these butterfly computations, we can determine that the radix-2 algorithm requires only $(N / 2)\log_2(N)$ multiplications and $N\log_2(N)$ additions. The values of W_N used in **Figure 2** are commonly known as "twiddle factors" and can be computed before the algorithm is performed.

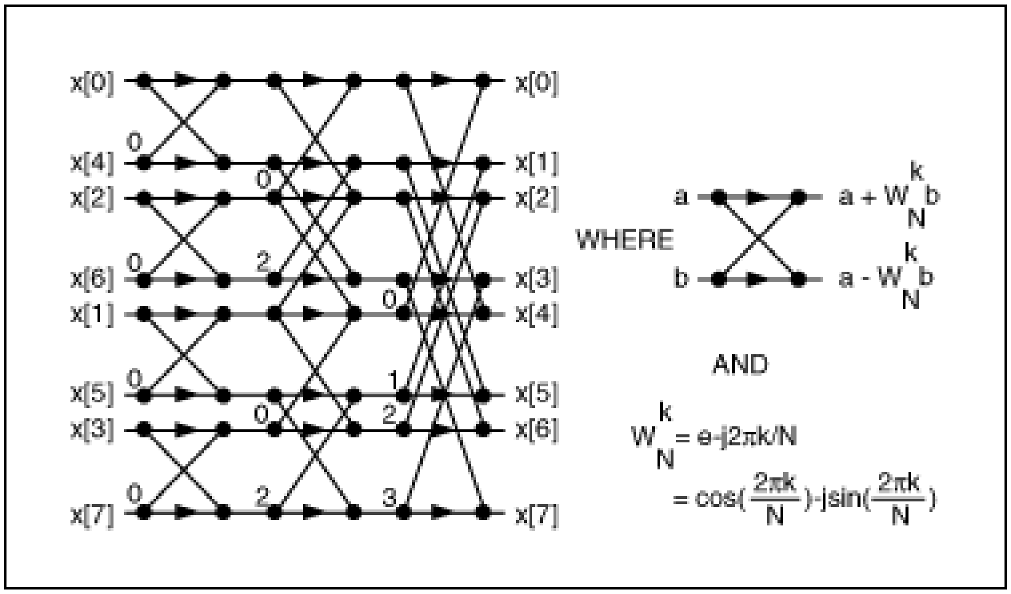

Figure 2. A butterfly computation is used to perform an FFT for N = 8.

In Figure 2, the input to the FFT is shown in its peculiar order with the indices bit-reversed. Therefore, computing the radix-2 FFT algorithm with N = 8 requires the input data to be reordered from:

0 (000b), 1 (001b), 2 (010b), 3 (011b), 4 (100b), 5(101b), 6(110b), 7(111b)

to:

0 (000b), 4, (100b), 2 (010b), 6 (110b), 1 (001b), 5 (101), 3 (011), 7 (111)

The FFT output appears in the correct order. Figure 2 also reveals that the results of the individual butterfly computations are the only data required for the next stage of the FFT. Because the computations are done "in place", new values can replace old values and only 2N variables are required to compute an FFT with N samples (2N variables are required because each value has a real and an imaginary part).

When the FFT is complete, the results are in complex notation. Equations 4 and 5 convert the results into polar notation:

$$X_{MAGN}(k) = \sqrt{X^2_{Re}(k) + X^2_{Im}(k)} \qquad \text{Eq. 04}$$

$$X_{PHASE}(k) = \arctan(\frac{X_{Im}(k)}{X_{Re}(k)}) \qquad \text{Eq. 05}$$

The DSP literature includes many optimizations for the DFT/FFT algorithm described above to make it faster and smaller. One of the most important optimizations (and also perhaps the easiest to implement) arises from the realization that the magnitude of the DFT of a real-valued signal is symmetric around X(N / 2), therefore:

$$X(k) = X \bullet (N/k) \qquad \text{Eq. 06}$$

Writing an FFT is not simple, and several limitations of low-power µCs can further complicate writing FFTs for these devices:

Memory The selected µC has 2kB of RAM. Knowing that the algorithm requires 2N 16-bit variables for FFT data, our µC can perform FFTs with N up to 512. However, other parts of the firmware also require a few bytes of RAM. For our implementation, we therefore limit N to 256. Using 16-bits variables to represent the real and imaginary parts of every value, 1024 bytes of RAM are required for FFT data.

Speed Despite having a high MIPS/mA rating, low-power µCs still require some optimization to minimize the number of instructions required to run the FFT. Fortunately, the C compiler used for this application (IAR Embedded Workbench for MAXQ at www.iar.com) includes a number of optimization levels and settings. Careful use of the hardware multiplier allows the code to be optimized to an acceptable level.

No Floating-Point Capability The µC chosen does not have floating-point capability (typical of devices from low-power families). Fixed-point arithmetic is therefore required for all computations. To represent fractional numbers, the firmware uses signed Q8.7 notation. The firmware therefore assumes:

- Bits 0 to 6 represent the fractional part of every number
- Bits 7 to 14 represent the integer part of every number
- Bit 15 represents the sign bit (two's complement)

This has no effect on additions and subtractions, but care must be taken during multiplications to align the numbers to Q8.7 format.

$$\begin{aligned}
& -0.5 \bullet 1.375 \\
&= -64 \bullet 176 \quad \Rightarrow \quad \text{Q8.7 notation} \\
&= (1111\ 1111\ 1100\ 0000)_b \bullet (0000\ 0000\ 1011\ 0000)_b \\
&= (1101\ 0100\ 0)_b \quad \Rightarrow \quad \text{result shifted to the right 7 times for alignment} \\
&= -88 \quad \Rightarrow \quad \text{Q8.7 notation} \\
&= -0.6875
\end{aligned}$$

The notation chosen also accommodates the largest number the FFT algorithm may encounter, while providing the highest accuracy. For example, our ADC provides signed 8-bit samples in two's complement format. If our input is a DC voltage with maximum amplitude (127 for signed 8-bit samples), the spectrum would be entirely contained in X(0) and be equal to 32512 in Q8.7 notation. This number fits into a single, signed, 16-bit value.

The Firmware

The following sections describe the firmware that computes a radix-2 FFT on a low-power µC. When the samples are read from the ADC, they are stored in the x_n_re array. This array represents the real values of x(n). The imaginary values, initialized to zero before the FFT begins, are stored in the x_n_im array. When the FFT is complete, the spectrum results will have replaced the original sampled values and be stored in x_n_re and x_n_im.

Gathering Samples

The FFT algorithm assumes that the samples are taken at a fixed sampling frequency. Gathering samples for an

FFT can cause difficulties if not done carefully. For example, jitter in the sample interval causes errors in the FFT results and should be minimized.

A decision statement in the ADC sample loop can cause jitter in the sample interval. For example, our system reads signed, 8-bit samples from an ADC and stores them in an array of 16-bit variables. Two pseudo-code algorithms for performing this ADC read-and-store function are shown in **Listing 1**. The method presented in Algorithm 1 will cause jitter in the sample interval because a negative sample requires more time to read and store than a positive sample.

Listing 1. Two pseudo-code algorithms for ADC sampling are illustrated. The second algorithm does not cause the same problem as the first—jitter in the sample interval.

```
// ALGORITHM 1: INCONSISTENT SAMPLING FREQUENCY - BAD!
// sample[] is an array of 16-bit variables
for i = 0 to (N-1)
begin
     doADCSampleConversion()             // Instruct ADC to sample Vin
     sample[i] = read8BitSampleFromADC() // Read 8-bit sample from ADC

     if (sample[i] & 0x0080)             // If the 8-bit sample was negative
          sample[i] = sample[i] + 0xFF00 // Make the 16-bit word negative
end

// ALGORITHM 2: FIXED SAMPLING FREQUENCY - GOOD!
// sample[] is an array of 16-bit variables
for i = 0 to (N-1)
begin
     doADCSampleConversion()             // Instruct ADC to sample Vin
     sample[i] = read8BitSampleFromADC() // Read 8-bit sample from ADC
end

for i = 0 to (N-1)
begin
     if (sample[i] & 0x0080)             // If the 8-bit sample was negative
          sample[i] = sample[i] + 0xFF00 // Make the 16-bit word negative
end
```

Trigonometric Lookup Tables

The FFT algorithm uses lookup tables (LUTs) instead of calculating the value of cosine or sine. The declarations for the sine and cosine LUTs are given in **Listing 2**, and comments in the actual firmware include source code for the program used to automatically generate these LUTs. Both LUTs have N / 2 elements because the indices of the twiddle factors vary from 0 to (N / 2) - 1 (see Figure 2).

Listing 2. LUTs are shown for sine and cosine functions.

```
const int cosLUT[N/2] = {+128,+127,+127, ... ,-127,-127,-127};
const int sinLUT[N/2] = {+0  ,+3  ,  +6, ... ,+9  ,  +6,  +3};
```

The arrays containing these LUTs are declared as **const**, forcing the compiler to store them in code space instead of data space. Because the LUT values must be in Q8.7 notation, they correspond to the actual cosine and sine values multiplied by 2^7.

Bit Reversal

The bit-reversal order (where N is known) can be calculated at runtime, indexed using a lookup table, or written directly with an unrolled loop. Each of these methods has trade-offs with regard to the size of the source code and execution speed. This FFT application performs bit reversal using an unrolled loop, which results in longer source code, but faster execution. The code in **Listing 3** shows the implementation of this unrolled loop. Comments in the actual firmware for this application include source code for the program used to automatically generate this unrolled loop.

Listing 3. An unrolled loop with N = 256 is used for bit reversal.

```
i=x_n_re[  1]; x_n_re[  1]=x_n_re[128]; x_n_re[128]=i;
i=x_n_re[  2]; x_n_re[  2]=x_n_re[ 64]; x_n_re[ 64]=i;
i=x_n_re[  3]; x_n_re[  3]=x_n_re[192]; x_n_re[192]=i;
i=x_n_re[  4]; x_n_re[  4]=x_n_re[ 32]; x_n_re[ 32]=i;
...
i=x_n_re[207]; x_n_re[207]=x_n_re[243]; x_n_re[243]=i;
i=x_n_re[215]; x_n_re[215]=x_n_re[235]; x_n_re[235]=i;
i=x_n_re[223]; x_n_re[223]=x_n_re[251]; x_n_re[251]=i;
i=x_n_re[239]; x_n_re[239]=x_n_re[247]; x_n_re[247]=i;
```

The Radix-2 FFT Algorithm

After the samples have been reordered using bit reversal, the FFT can be computed. The firmware for this implementation of the radix-2 FFT performs the butterfly computations seen in Figure 2 with three main loops. The outside loop counts through the $\log_2(N)$ stages of the FFT computation. The inner loops perform the individual butterfly computations of each stage.

The heart of the FFT algorithm is the short block of code that performs each butterfly computation. This block, shown in **Listing 4**, is unfortunately the only nonportable firmware in this implementation. The MUL_1 and MUL_2 macros use the µC's hardware multiplier to perform multiplications in a single instruction cycle. The contents of these macros, which are specific to the MAXQ2000, can be fully examined in the actual firmware.

Listing 4. Butterfly computation is performed in C.

```
/* (1) Macro MUL_1(A,B,C): C=A*B      (result in Q8.7)*/
/* (2) Macro MUL_2(A,C)  : C=A*last_B (result in Q8.7)*/
MUL_1(cosLUT[tf],x_n_re[b],resultMulReCos);
MUL_2(sinLUT[tf],resultMulReSin);
MUL_1(cosLUT[tf],x_n_im[b],resultMulImCos);
MUL_2(sinLUT[tf],resultMulImSin);

x_n_re[b] = x_n_re[a]-resultMulReCos+resultMulImSin;
x_n_im[b] = x_n_im[a]-resultMulReSin-resultMulImCos;
x_n_re[a] = x_n_re[a]+resultMulReCos-resultMulImSin;
x_n_im[a] = x_n_im[a]+resultMulReSin+resultMulImCos;
```

Complex to Polar Conversion

To determine the magnitude for the spectrum of V_{IN}, we must convert the complex values of X(k) into polar notation. The firmware that implements this conversion is shown in **Listing 5**. The magnitude values replace the original results of the FFT that are no longer needed by the firmware.

Listing 5. FFT results are converted from complex to polar notation.

```
const unsigned char magnLUT[16][16] =
{
{0x00,0x10,0x20, ... ,0xd0,0xe0,0xf0},
{0x10,0x16,0x23, ... ,0xd0,0xe0,0xf0},
                 ...
{0xe0,0xe0,0xe2, ... ,0xff,0xff,0xff},
{0xf0,0xf0,0xf2, ... ,0xff,0xff,0xff}
};
                 ...
                 ...
/* Compute x_n_re=abs(x_n_re) and x_n_im=abs(x_n_im) */
                 ...
                 ...
x_n_re[0] = magnLUT[x_n_re[0]>>11][0];

for(i=1; i<N_DIV_2; i++)
x_n_re[i] = magnLUT[x_n_re[i]>>11][x_n_im[i]>>11];

x_n_re[N_DIV_2] = magnLUT[x_n_re[N_DIV_2]>>11][0];
```

F A two-dimensional LUT determines the magnitude instead of the computation from equation 4. The first index is 4 most significant bits (MSB) of the real part of the spectrum, while the second index is 4 MSB of the imaginary part of the spectrum. To obtain these 4 MSB, the signed, 16-bit values are right shifted 11 times. Before the real and imaginary parts of the spectrum can be used as indices, they are replaced by their absolute values. Therefore, the sign bit will be zero.

Because it is known from equation 6 that the magnitude of the spectrum is symmetric with respect to X(N / 2), only the magnitudes of the first (N / 2) + 1 spectrum values are converted to polar notation. Also, it can be shown that the imaginary parts of X(0) and X(N / 2) are always zero for real-valued input samples. These two magnitudes are therefore calculated separately. Comments in the actual firmware for this project include source code for the program used to automatically generate the LUT for the magnitude of X(k).

Hamming or Hann Windows

The firmware for this project includes LUTs (in Q8.7 format) to apply a Hamming window or a Hann window to the input samples. Windowing is useful to reduce spectral leakage that can result from truncating x(n) in the time domain. The equation for the Hamming and Hann window functions are shown in equations 7 and 8, respectively.

$$h(n) = 0.54 - 0.46\cos\left(\frac{2\pi n}{N-1}\right) \qquad \text{Eq. 07}$$

$$h(n) = 0.5\left[1 - \cos\left(\frac{2\pi n}{N-1}\right)\right] \qquad \text{Eq. 08}$$

Listing 6 shows the code for the implementation of these functions. Again, comments in the actual firmware for this project include source code for the program used to automatically generate the LUTs for these windowing functions.

Listing 6. LUTs are shown for the implementation of Hamming and Hann window functions.

```
const char hammingLUT[N] = {+10, +10, +10, ... ,+10, +10, +10};
const char hannLUT[N]    = { +0,  +0,  +0, ... , +0,  +0,  +0};
...
...
for(i=0; i<256; i++)
{
   #ifdef WINDOWING_HAMMING
      MUL_1(x_n_re[i],hammingLUT[i],x_n_re[i]); // x(n)*=hamming(n);
   #endif
   #ifdef WINDOWING_HANN
      MUL_1(x_n_re[i],hannLUT[i]),x_n_re[i]);   // x(n)*=hann(n);
   #endif
}
```

Testing the Results

To test the result of the FFT application, the firmware uploads the magnitude of X(k) to a PC using the µC's UART port. *FFT Graph*, a software application (included with the firmware for this project) written specifically to read these magnitude values from the PC's serial port, graphs the magnitude of the calculated spectrum in real time. **Figure 3** shows the results displayed from *FFT Graph* for four different input signals with the µC sampling the input voltage at 200ksps:

1. 4.3V DC signal
2. 50kHz sine wave
3. 70kHz sine wave
4. 6.25kHz square wave

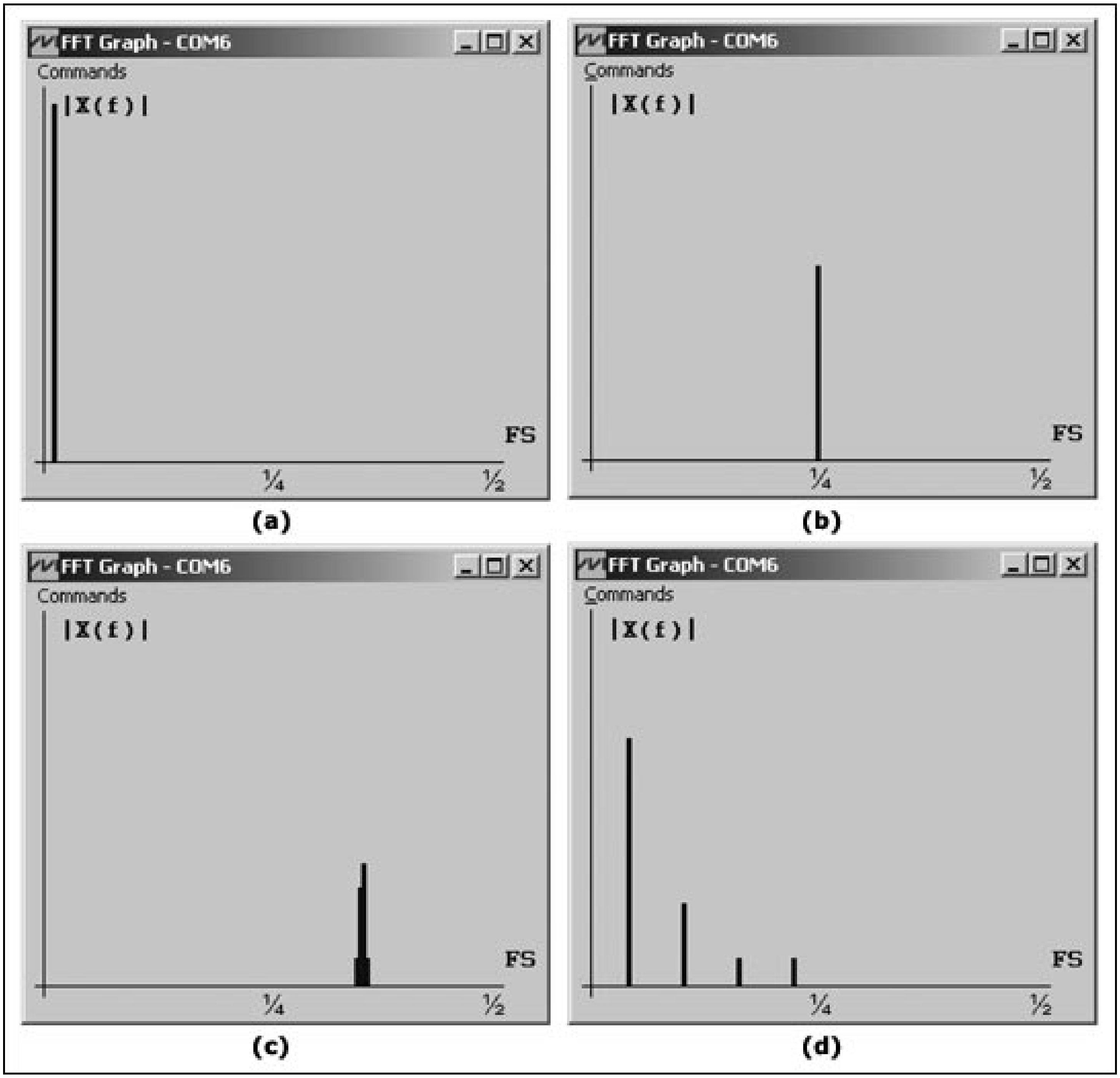

Figure 3. FFT Graph is used to plot the results of spectra calculated by a low-power µC.

What Is Next?

The interested reader can spend an unlimited amount of time optimizing and configuring this FFT implementation. Although the radix-2 algorithm was chosen for this article, others algorithms exist that can reduce the number of additions and multiplications required dramatically. Many optimizations not presented in this article also exist for increasing the speed of a FFT. For example, with real-valued input samples, the imaginary part of the input samples is always zero, and only the first half of the spectrum is significant. Using this information, the first and last stages of the FFT can be optimized for faster execution, but more program space may be required.

The algorithm presented in this article is, however, a good starting point for an FFT algorithm written specifically for a low-power µC. For more information and implementation details, please review the well-commented firmware for this application.

Resources

Cooley, J. W. and J. W. Tukey, An Algorithm for the Machine Computation of Complex Fourier Series, *Mathematics Computation*, Vol. 19, pp 297-301, 1965.
Lemieux, Joe, Fixed-point math in C, *Embedded Systems Programming*, October 2003.
Proakis, John G. and Dimitris G. Manolakism, *Digital Signal Processing Principles, Algorithms, and Applications*, 3rd Edition, Prentice Hall, 1996.
Smith, Steven W., *The Scientist and Engineer's Guide to Digital Signal Processing*, 2nd Edition, California Technical Publishing, 1999.

A similar article appeared in the Ocober, 2005 issue of *Embedded Systems Design*.