**Freescale Semiconductor**
Application Note

Document Number: AN3599
Rev.0, 07/2008

# Digital Signal Processing and ADC/DAC for DSP56800/E

by: XiangJun Rong
Systems and Applications Engineering
Asia/Pacific Region

Contents

1 Introduction . . . 1
2 DSP56800/E Features . . . 2
2.1 Overview . . . 2
2.2 On-Chip ADC Features . . . 3
2.3 ADC Specifications . . . 4
2.4 How to Calculate ENOB . . . 5
2.5 ADC Voltage Reference Circuit . . . 8
3 Filter Design Using QEDesign Lite . . . 9
3.1 FIR Filter . . . 9
3.2 IIR Filter Design . . . 15
3.3 Illustration of the Coefficients from QEDesign Lite 21
3.4 How to Call the IIR API Function . . . 24
3.5 Modify the Link Command File for Dynamic Memory Allocation . . . 27
3.6 How to Call the FIR API Function . . . 31
4 FFT . . . 35
4.1 FFT Implementation . . . 35
4.2 FFT Results . . . 38
4.3 General Code to Compute DFT for Comparison . . 39
5 DAC . . . 40
5.1 DAC of the MC56F802x Family . . . 40
5.2 Class D Amplifier . . . 42
6 Conclusion . . . 43
Appendix A IIR Filter Code . . . 44
Appendix B References . . . 45

## 1 Introduction

The DSP56800/E is Freescale's 16-bit fixed-point digital signal controller (DSC), which can be used in a variety of industrial control applications, such as motor control for BLDC, PMSM, ACIM, SR, and STEP motors, as well as in switch-mode power supplies and as an electronic lamp ballast.

Freescale devices that use the DSP56800/E core can perform up to 32–120 MIPS.

- The DSP56F80x can perform up to 40 MIPS
- The MC56F83xx can perform up to 60 MIPS
- The MC56F80xx can perform up to 32 MIPS
- The DSP5685x can perform up to 120 MIPS

Most of the devices have on-chip flash, GPIO, watchdog timer, Quad Timer, ADC, DAC, CAN bus, SPI bus, SCI bus, IIC bus, decoder, and JTAG/OnCE peripherals.

This application note provides:

- On-chip ADC specifications of the DSP56800/E

© Freescale Semiconductor, Inc., 2008. All rights reserved.

freescale™ semiconductor

- Procedures to design a finite impulse response filter (FIR) and an infinite impulse response filter (IIR) using CodeWarrior embedded QEDesign Lite for the DSP56800/E
- Examples by Processor Expert (PE) to create a digital filter and FFT based on the DSP56800/E core
- The relationship between PWM cycle and PWM resolution when the PWM is viewed as a DAC

Figure 1 is a common digital signal processing procedure. The ADC and DAC can be on-chip or off-chip. The DSC is capable of implementing the digital signal processing in real time based on the DSP architecture and compact instructions set. Most of the Freescale DSCs have on-chip ADC.

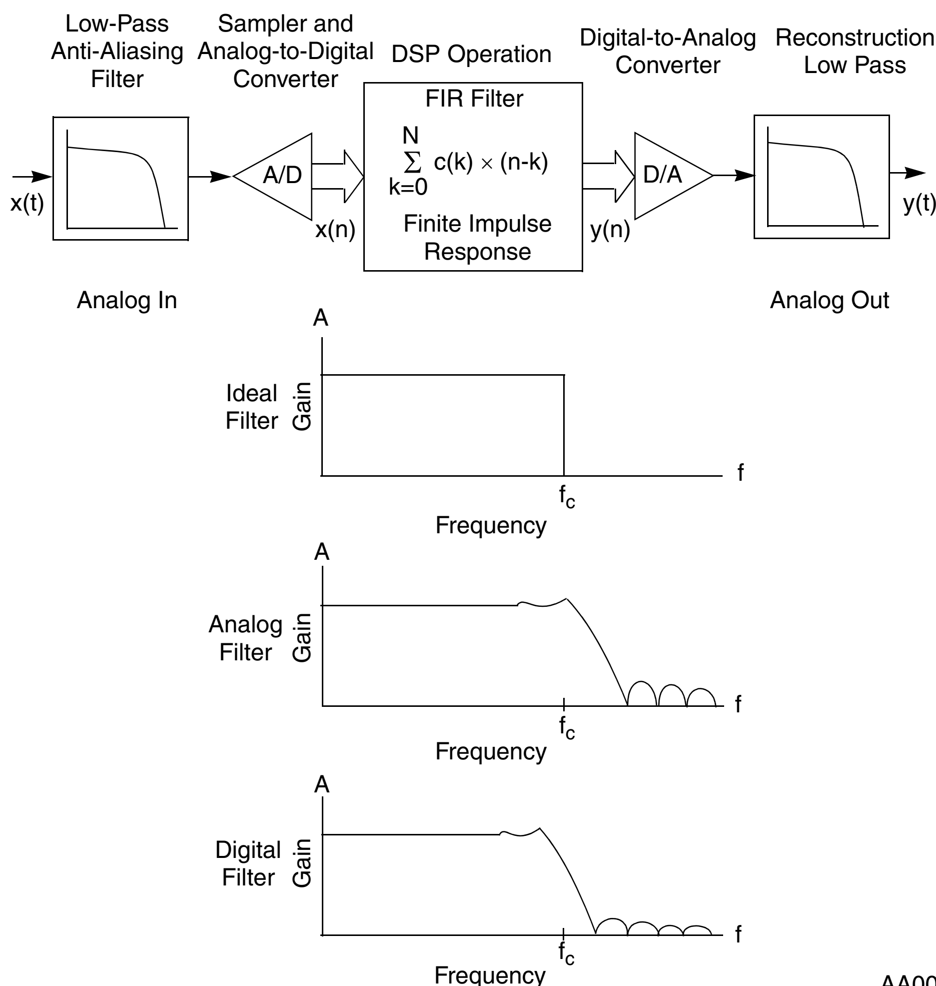



**Figure 1. Digital Signal Processing Scheme**

# 2 DSP56800/E Features

## 2.1 Overview

Every device in the DSP56800/E family is a general-purpose DSC for various control and signal processing applications.

As shown in Figure 1, the key attributes of the DSP core are:

- Multiply/accumulate (MAC) operation in one clock
- Harvard structure, fetching up to two operands per instruction cycle, for the MAC
- Program control to provide versatile operation
- Input/output transferring data to and from the DSC

The Freescale DSP56F80x family is well suited for digital signal processing and motor control. It combines a DSP's calculation capability with an MCU's controller features on a single chip.

DSP56F805, a typical member of the DSP56F80x family, provides these peripheral blocks:

- Two pulse-width modulator modules (PWMA and PWMB). Each provides six PWM outputs, three current-sense inputs, and four fault inputs. It has fault-tolerant design with dead-time insertion and supports both center-aligned and edge-aligned modes.
- A 12-bit analog-to-digital convertor (ADC). Supports two simultaneous conversions with dual 4-pin multiplexed inputs. The ADC can be synchronized by a PWM or by a timer.
- Two quadrature decoders (Quad Dec0 and Quad Dec1). Each provides four inputs or two additional Quad Timers, A and B.
- Two dedicated general-purpose Quad Timers totalling six pins: Timer C with two pins and Timer D with four pins.
- CAN 2.0 A/B module with 2-pin ports used to transmit and receive
- Two serial communication interfaces (SCI0 and SCI1). Each provides two pins, or four additional GPIO lines.
- Serial peripheral interface (SPI) with configurable 4-pin port, or four additional GPIO lines.
- Computer-operating-properly (COP) watchdog timer.
- Two dedicated external interrupt pins.
- 14 dedicated general-purpose I/O (GPIO) pins and 18 multiplexed GPIO pins.
- External reset pin for hardware reset.
- JTAG/On-chip emulation (OnCE).
- Software-programmable, phase-lock loop-based frequency synthesizer for the DSP core clock.

Other than the fast analog-to-digital converter and 16-bit quadrature timers, the pulse-width modulation block (PWM) offers a high degree of freedom in its configuration to control various motors in an efficient way.

## 2.2 On-Chip ADC Features

The ADC module has these features:

- 12-bit resolution
- A sampling rate up to 1.66 million samples per second
- Maximum ADC clock frequency of 5 MHz with 200 ns period
- Single conversion time of 8.5 ADC clock cycles ($8.5 \times 200$ ns = 1.7 µs)
- Additional conversion time of 6 ADC clock cycles ($6 \times 200$ ns = 1.2 µs)

- Eight conversions in 26.5 ADC clock cycles (26.5 × 200 ns = 5.3 µs) using simultaneous mode
- Simultaneous or sequential sampling
- Internal multiplexer to select two of eight inputs
- Ability to sequentially scan and store up to eight measurements
- Ability to simultaneously sample and hold two inputs
- Optional interrupts at end of scan, if an out-of-range limit is exceeded or if at zero crossing
- Optional sample correction by subtracting a pre-programmed offset value
- Signed or unsigned result
- Single-ended or differential inputs

In loop mode, the time between each conversion is six ADC clock cycles (1.2 µs). Using simultaneous conversion, two samples can be obtained in 1.2 µs. Therefore the module can process up to 1.6 million samples per second.

The external analog signal must be sampled in a constant cycle if FIR, IIR, or FFT is adopted. The DSP56800/E family provides two trigger modes to acquire samples. One is software trigger mode, which is set by setting the START bit in the ADC control register; the other is hardware trigger mode. In hardware trigger mode, the timer outputs a signal to trigger the ADC internally. The user can set the timer register to set up a constant sample cycle.

The DSP56800/E also supports a more complicated trigger mode. For example, the PWM reload event triggers the timer and the timer triggers the ADC after delays. This is an important mode for sampling three motor phase currents for a Clark transformation. Refer to the application note AN1933*, Synchronization of On-chip Analog to Digital Converter on DSP56F80x DSPs*, for details. Only the hardware trigger mode can guarantee the synchronization of the ADC samples, so the user must use the hardware trigger mode to get samples via on-chip ADC to implement FIR, IIR, or FFT.

## 2.3 ADC Specifications

This section defines all the ADC specifications used in this application note.

- Differential non-linearity (DNL) — the measurement of the maximum deviation from the ideal step size of 1 LSB.
- Effective number of bits (ENOB or effective bits) — another method of specifying signal-to-noise and distortion (SINAD). ENOB is defined as (SINAD – 1.76)/6.02, and it is equivalent to the number of bits in a perfect ADC.
- Gain error — the deviation from the ideal slope of the transfer function.
- Integral non-linearity (INL) — a measurement of the deviation of each individual code from a line drawn from ground (1/2 LSB below the first code transition) through positive full scale (1/2 LSB above the last code transition). The deviation of any given code from this straight line is measured from the bottom of that code value or ground.
- Offset voltage — the maximum input voltage of the ADC analog channel when the ADC sample reading is zero.

- Signal-to-noise ratio (SNR) — the ratio, expressed in dB, of the RMS value of the fundamental signal to the RMS value of the sum of all other spectral components below one-half of the sampling frequency, but not including harmonics from f2 to f9 or DC.
- Signal-to-noise plus distortion (S/N+D or SINAD) — the ratio, expressed in dB, of the RMS value of the fundamental signal to the RMS value of all the other spectral components below half of the sample frequency, including all harmonics but excluding DC.
- Spurious free dynamic range (SFDR) — the ratio, expressed in dB, of the RMS values of the fundamental signal to the RMS value of the peak spurious signal. The spurious signal is any signal presented in the output spectrum that is not presented at the input.
- Total harmonic distortion (THD) — the ratio, expressed in dB, of the RMS value of the fundamental signal to the RMS value of the total of the first nine harmonics. THD is calculated with f1 as the RMS power of the fundamental (output) frequency and f2 through f10 as the RMS power in the first nine harmonic frequencies.
- Crosstalk between channels — the mutual influence between two analog channels expressed in dB. For example, the crosstalk between two ADC analog channels is 60 dB in the MC56F8300 family. If the input of one channel is 1 V, the second channel may get 1 mV noise signal from the first channel.
- Common mode voltage — defined only in the ADC differential input mode, and equal to the average voltage of the two differential input signals. Note that each channel voltage must range from GND to $V_{refh}$ even in differential mode, because the on-chip ADC of the DSP56800/E cannot accept negative voltage.
- $V_{refh}$ current — the current the $V_{refh}$ pin consumes. When the user uses a voltage reference, this specification must be considered.
- Impedance of the analog channel — refer to Freescale application note AN1947, *DSP56F800 ADC*, for the formula to compute this value. It is about 200 kΩ.

**NOTE**

The root mean square (RMS) for a collection of n values $\{x_1,x_2,x_3,...,x_n\}$ can be expressed as $x_{rms} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2}$, for a sine signal. If the amplitude of the sine signal is A, the RMS of the sine signal is $A/\sqrt{2}$.

## 2.4 How to Calculate ENOB

Resolution is an important specification for an ADC. It indicates the minimum ADC recognition for the analog input variation. For example, if the ADC is 12 bits, the resolution of the ADC is 12 bits. But because the actual ADC cannot recognize the analog input variation as its resolution describes, we define the ENOB parameter to describe its actual resolution.

### 2.4.1 How to Derive the ENOB Parameter

From a mathematical point of view, quantization is the process of mapping the continuous analog signal to a discrete definite number set. Quantization error is introduced in the process. We can use the

quantization error to measure the difference between the actual analog value x(k) and the corresponding ADC digital sample. Assume the digital ADC sample is

$$\hat{x}^{(k)} = b_{11} \times 2^{11} + b_{10} \times 2^{10} + \ldots + b_0 \times 2^0 \quad \textbf{Eqn. 1}$$

We use the root-mean-square error to define the quantization error as Equation 2. The error x is the difference between the analog sample and the digital sample:

$$x = x(k) - \hat{x}^{(k)} \quad \textbf{Eqn. 2}$$

For a general ADC, we assume the analog signal voltage is uniformly distributed. Therefore the uniform quantization is the optimum quantization.

Suppose that:

- Analog signal ranges from –V to +V
- Nominal resolution is L bits
- Desired number of levels is $2^L$
- Quantization gap is $2 \times V/2^L$
- Sample frequency is $F_s$
- Signal frequency is F (for meeting the Nyquist sampling theorem, $F_s$ must be greater than 2F)

Suppose we quantize a sine waveform signal with the ADC — the amplitude of the sine signal is V. In an ideal ADC converter, the quantization error is uniformly distributed between –1/2 LSB and +1/2 LSB. The sum of the noise, or the root-mean-square quantization error (or the energy of the quantization error), is

$$J = \int_{-\Delta/2}^{\Delta/2} x^2 \times P(x)dx = \int_{-\Delta/2}^{\Delta/2} x^2 \times \frac{1}{\Delta}dx = \frac{1}{3\Delta}x^3 \Big|_{-\Delta/2}^{\Delta/2} = \frac{\Delta^2}{12} \quad \textbf{Eqn. 3}$$

In the above formula, x denotes the quantization error between the continuous analog signal and the digital sample, and p(x) is the possibility of the quantization error assumed to be uniformly distributed. So

$$p(x) = \frac{1}{\Delta} \text{ and } \Delta = 2 \times V/2^L$$

and the noise energy is

$$J = \frac{V^2}{3 \times 2^{2L}} \quad \textbf{Eqn. 4}$$

For a sine signal with amplitude V, the signal energy is $V^2/2$, so

$$10 \times \log_{10}(\text{Signal Energy/Noise Energy}) = 10 \times \log_{10}((V^2/2)/J) = 6.02L + 1.76 \quad \textbf{Eqn. 5}$$

$$\text{ENOB L} = [10 \times \log_{10}(\text{Signal Energy/Noise Energy}) - 1.76] / 6.02 \quad \textit{Eqn. 6}$$

## 2.4.2 One Method to Compute ENOB Parameter

According to Equation 6, if we can calculate the signal and noise energy, we can derive L. L is the ENOB in the data sheet.

Suppose we input an arbitrary initial-phase sine signal to the ADC, which can be expressed as Asin(wt)+Bcos(wt). The amplitude of the sine signal is $\sqrt{A^2 + B^2}$. Assume that the frequency of the signal is $F_{REQ}$ and the signal energy is $(A^2 + B^2)/2$. Assume that the sample frequency is $F_S$. Select the input sine signal frequency $F_{REQ}$ to make the result of $F_S/F_{REQ}$ to be an integer, such as 100, to increase accuracy.

Assume we get one whole cycle sine discrete signal y(k) by sampling the sine signal. The number of points is NP, which equals $F_S/F_{REQ}$. Use Equation 7 to calculate the noise value.

$$J = \sum_{k=0}^{NP-1} \left[ y(k) - A \times \sin\frac{2\pi k}{NP} - B \times \cos\frac{2\pi k}{NP} \right]^2 \quad \textit{Eqn. 7}$$

Estimate the optimum values for parameters A and B that will yield a minimum value for the noise energy J. After calculating the optimum A and B, we can compute J. Let the following partial derivatives with respect to A and B equal zero.

$$\frac{\partial J}{\partial A} = 0$$

$$\frac{\partial J}{\partial B} = 0$$

To calculate signal energy, we use Equation 8 and Equation 9, where parameter k is from 0 to NP–1.

$$\sum \left[ y(k) - A\sin\left(\frac{2\pi k}{NP}\right) - B\cos\left(\frac{2\pi k}{NP}\right) \right] \times \sin\frac{2\pi k}{NP} = 0 \quad \textit{Eqn. 8}$$

$$\sum \left[ y(k) - A\sin\left(\frac{2\pi k}{NP}\right) - B\cos\left(\frac{2\pi k}{NP}\right) \right] \times \cos\frac{2\pi k}{NP} = 0 \quad \textit{Eqn. 9}$$

Using these two equations, we can calculate A and B. With the values for A and B, use Equation 7 to calculate the noise, J.

$$\text{Signal Energy} = (A^2 + B^2) / 2 \quad \textit{Eqn. 10}$$

Note that the average of the y(k) sequence is zero. We can subtract the original sample sequence by the average and get the y(k) sequence.

Using Equation 6 we can calculate L, which is ENOB as defined in the data sheet.

We can also calculate ENOB according to a discrete Fourier transformation (DFT) of the y(k) sequence — they yield the same result. For example, assume we have computed the DFT of the y(k) sequence, and get the power spectrum sequence from bin0 to bin50. Obviously, bin0 is the DC component which is zero here — bin1 is the fundamental component. In computing the SNR, because bin10 to bin50 are regarded as noise, we sum from bin10 to bin50 to get noise. Bin2 to bin10 are regarded as harmonics. In computing S/N+D or SINAD, bin2 to bin50 are regarded as noise. Therefore, to compute the value for the noise, sum from bin2 to bin50. Note that when we compute for all the specifications, the input sine signal voltage must be in full scale.

We can use Equation 11 to compute the ENOB after we get the SINAD.

$$ENOB = (SINAD - 1.76) / 6.02 \quad \textit{Eqn. 11}$$

## 2.5 ADC Voltage Reference Circuit

The on-chip ADC of the DSP56800/E has a $V_{refh}$ pin that can be used to provide reference voltage to the ADC.

Use a dedicated voltage reference device to improve the ADC precision. Refer to the data sheet of each individual DSP chip to get the typical current requirements of the $V_{refh}$ pin that provides a reference voltage for the on-chip ADC. For example, the $V_{ref}$ pin of DSP56F805 typically needs 12 mA DC current, and the optimum voltage of the $V_{ref}$ is about 3.0 V. But for the MC56F8300 family and MC56F8000 family, the optimum input voltage for the $V_{refh}$ is 3.3 V. This application note uses the LM385-ADJ chip as the voltage reference device for the $V_{refh}$ pin of DSP56F805. The user can connect the $V_{refh}$ pin to $V_{DDA}$ for a low-cost design.

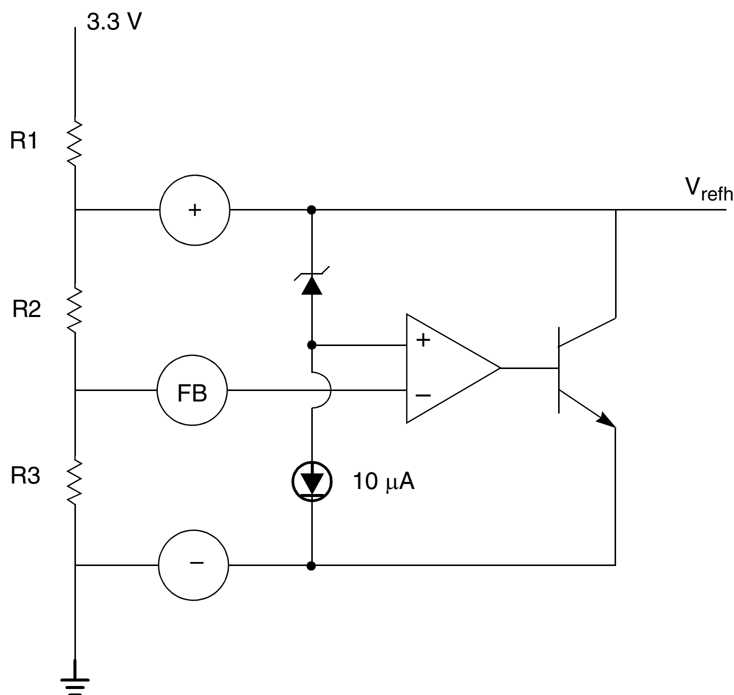


**Figure 2. Voltage Reference Circuit**

Here, the “+”, “–”, and “FB” are nodes copied from the data sheet of the LM385-ADJ chip. Because the clamp diode clamps the voltage between the FB node and the $V_{refh}$ node to 1.24 V, and the current flow through R2 and R3 is the same, we can derive the formula given here:

$$\frac{V_{refh} - 1.24}{R3} = \frac{1.24}{R2} \quad \text{Eqn. 12}$$

$$V_{refh} = 1.24\left(1 + \frac{R3}{R2}\right) \quad \text{Eqn. 13}$$

If R3 is 14.3 kΩ and R2 is 10 kΩ, then $V_{refh}$ will be 3.0 V.

R1 must be a tri-port adjustable resistor. The maximum resistance is R1 = 0.3 V / 0.012 A = 25 Ω. It must be less than 25 Ω. Typically it is about 16–17 Ω, because the $V_{refh}$ pin consumes 12 mA current.

# 3 Filter Design Using QEDesign Lite

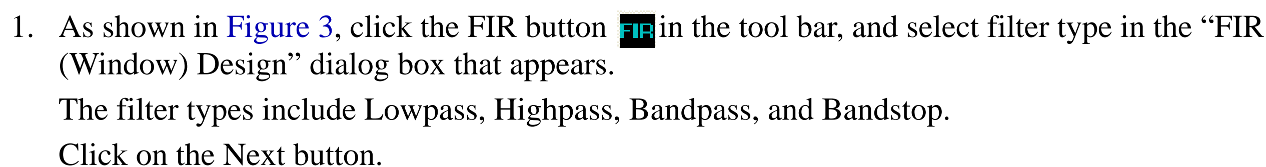
## 3.1 FIR Filter

We provide a filter solution, including digital filter design and digital filter implementation, using the DSP. For filter design, CodeWarrior for DSP56800/E integrates QEDesign Lite, which can help the user design an individual filter as required.

**NOTE**

> QEDesign Lite must be installed in your computer and working normally. This tool is integrated into CodeWarrior.

Use the procedure below to design a filter using QEDesign Lite.



1. As shown in Figure 3, click the FIR button in the tool bar, and select filter type in the “FIR (Window) Design” dialog box that appears.

   The filter types include Lowpass, Highpass, Bandpass, and Bandstop.

   Click on the Next button.

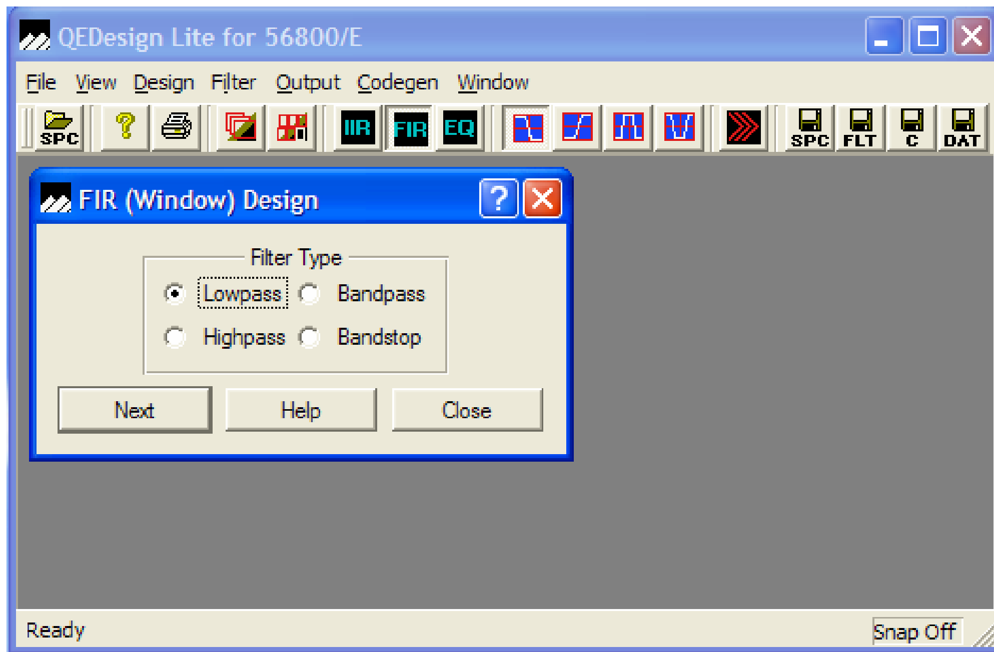



**Figure 3. QEDesign Lite FIR Filter Type Selection**

2. As shown in Figure 4, input filter parameters vary according to the application.

   In the Lowpass Filter window, passband ripple (dB) is the gain of the filter, a gain that fluctuates with the frequency within the passband frequency field. Stopband ripple (dB) works exactly the same way for the stopband frequency field.

   If you input x as the passband ripple within the passband, the gain ripple is $10^{-x/20}$ within the passband frequency. For example, if you input 3, the filter gain ripple is 0.708. The larger the ripple is in dBs, the smaller the filter gain ripple will be.

   For a low pass FIR filter design, assume that the sample frequency is 1000 Hz, the passband is 100 Hz, the stopband is 200 Hz, the passband ripple is 3 dB (the filter gain fluctuates between 70.7% and 100% of the maximum gain within the passband), and the stopband ripple is 20 dB (the filter gain fluctuates within 10% of the maximum gain within the stopband).

   You can input these typical values:

   — Sampling frequency: 1000

   — Passband frequency: 100

   — Stopband frequency: 200

   — Passband ripple (dB): 3

   — Stopband ripple (dB): 20

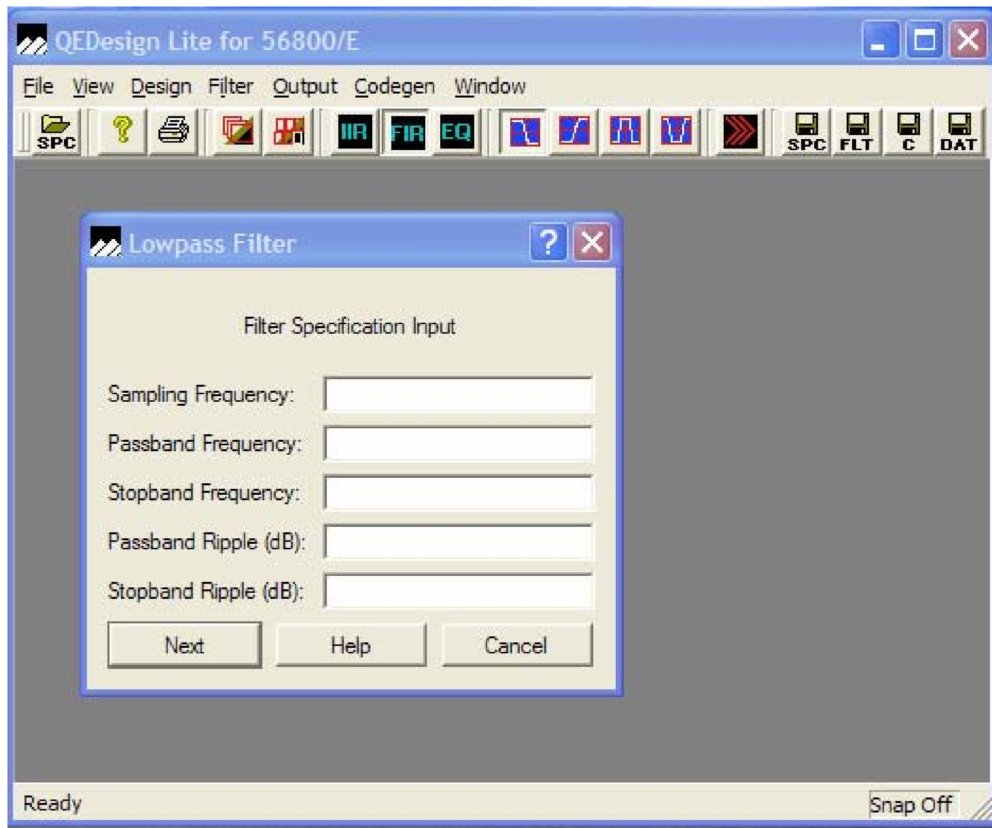



**Figure 4. QEDesign Lite FIR Filter Specification Input**

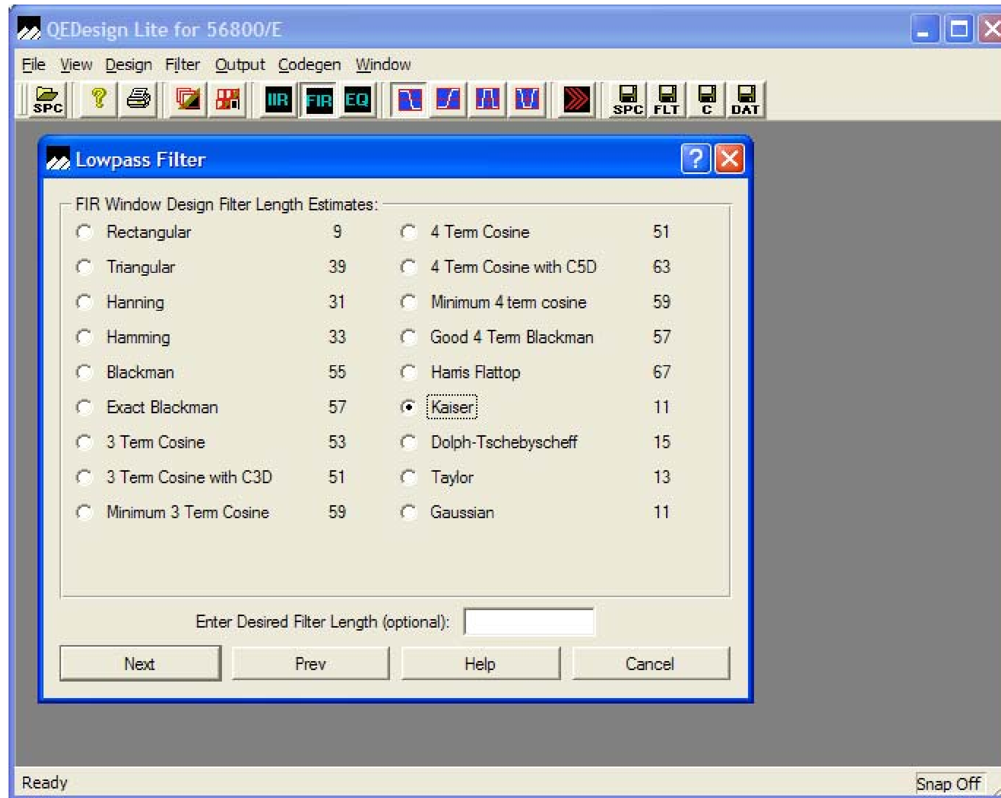



**Figure 5. QEDesign Lite FIR Filter Length Estimates**

3. Select one of the types seen in Figure 5 so that the tools can truncate the infinite series. Then click on the Next button. After the procedure is finished, the user can get the figure and data files that contain the FIR coefficients.

   The user can dump the coefficient and filter specifications into different files by clicking the tool bar. For example, these are the FIR filter coefficients with 11 taps that are saved in the data file:

   — –5.96008300781250e–002
   — –4.37927246093750e–002
   — 3.07006835937500e–002
   — 1.41784667968750e–001
   — 2.41210937500000e–001
   — 2.80975341796875e–001
   — 2.41210937500000e–001
   — 1.41784667968750e–001
   — 3.07006835937500e–002
   — –4.37927246093750e–002
   — –5.96008300781250e–002

   These are the filter specifications that are saved in the specification file:

   — Filter Specification File

- Filter Type: Low Pass 1 Hz
- Passband Ripple in –dB –.3000E+01
- Stopband Ripple in –dB –.2000E+02
- Passband Cutoff Frequency 0.100000E+03 Hz
- Stopband Cutoff Frequency 0.200000E+03 Hz
- Sampling Frequency 0.100000E+04 Hz

**Figure 6. QEDesign Lite FIR Filter Result Curve**

For the 11-tap FIR filter, the transfer function is expressed as:

$$H(z) = h(0) + h(1) \times z^{-1} + h(2) \times z^{-2} + h(3) \times z^{-3} + \ldots + h(10) \times z^{-10} \qquad \textbf{Eqn. 14}$$

The digital filter is designed in a normalized frequency: $F_{normal}=F_{actual}/F_{sample}$.

$F_{actual}$ is the actual frequency of the analog signal.

$F_{sample}$ is the ADC sample frequency.

In magnitude versus frequency:

$$H(jw) = \sum_{n=0}^{N-1} h(n) \times \frac{e^{-jnw}}{\text{Overall Gain}} \qquad \textbf{Eqn. 15}$$

N is the total tap number of the FIR filter: $w = 2\pi F_{normal} = 2\pi \times (F_{actual}/F_{sample})$. The overall gain is the gain of the filter in all frequency ranges, so that the maximum gain of the filter can equal one by dividing the actual filter gain by the overall gain parameter.

For the above 11-tap FIR filter, the curve for magnitude versus frequency can be calculated by

$$H(jw) = [h(0) + h(1) \times e^{-jw} + h(2) \times e^{-2jw} + h(3) \times e^{-3jw} + \ldots + h(10) \times e^{-10jw}] / \text{Overall Gain} \qquad \textbf{Eqn. 16}$$

The normalized frequency ranges from 0 to π. Assume we use a step of π/100 — that is, w=0, π/100, 2π/100, 3π/100, …π. The actual frequency, $F_{actual}$, is from 0 to $F_{sample}/2$.

Thus, we can get a continuous curve for magnitude versus frequency. The actual frequency corresponding to the normalized frequency is $F_{actual} = F_{normal} \times F_{sample}$.

In the curve of log magnitude versus frequency, the curve is computed by $20 \times \log_{10}[|H(jw)|/\text{overall gain}]$, the normalized frequency w is from 0 to π, and the actual frequency is from 0 to half of the ADC sample frequency.

In phase versus frequency

$$\text{Phase}(jw) = \text{Arctan}(\text{image}[H(jw)] \,/\, \text{real}[H(jw)]) \qquad \textbf{Eqn. 17}$$

the w ranges from 0 to π.

Regarding the group delay concept of a filter: if a sine signal passes through a linear system, the output is also a sine signal with the same frequency as the input and without any other frequency component, but the amplitude and phase will change. A linear phase system is a linear system: the phase shift between input and output is proportional to the input signal frequency. Group delay, as one of the linear phase system features, is a constant for all frequencies.

Two features of the linear phase are that the phase difference between the output sine signal and the input sine signal is proportional to the input signal frequency, and all frequency signals delay the same time after the sine signal passes through the linear phase system.

Regarding the impulse response versus time: input an impulse to the system (the filter can be treated as a linear system), and the output signal will be the impulse response.

Use the convolution formula to compute the impulse response versus time in the discrete system.

$$y(n) = \sum_{k=0}^{N-1} h(k) \times x(n-k) \qquad \textbf{Eqn. 18}$$

Use Equation 19 to compute the output sequence y(n).

$$y(n) = h(0) \times x(n) + h(1) \times x(n-1) + h(2) \times x(n-2) + h(3) \times x(n-3) + \ldots + h(N-1) \times x(n-N+1) \quad \textbf{Eqn. 19}$$

For the impulse response, the input data sequence is:

$$x(n) = \begin{cases} 1, & n = 0 \\ 0, & n > 0 \end{cases} \quad \textbf{Eqn. 20}$$

So y(n) = h(n) and n ranges from 0 to N–1. In conclusion, impulse response means that only x(0)=1 for the input series. The output series is the same as h(n).

When a step function inputs to the linear system, the output response is the step response.

For the step input response, the input data sequence is:

x(n)={1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,………}

After the interim process is finished, y(n) will converge to:

$$y(n) = \sum_{k=0}^{N-1} h(k) \quad \textbf{Eqn. 21}$$

The value of n is greater than the tap number of the FIR filter.

## 3.2 IIR Filter Design

This section introduces the IIR filter design procedure and results. It also introduces how to implement the IIR filter. The user can get the code from Appendix A, "IIR Filter Code."

### 3.2.1 IIR Filter Design Procedure and Results

1. As shown in Figure 7, you will input filter parameters according to the application.

   You can input these values as an example:

   — Sampling frequency: 1000

   — Passband frequency: 100

   — Stopband frequency: 200

   — Passband ripple (dB): 3

   — Stopband ripple (dB): 20

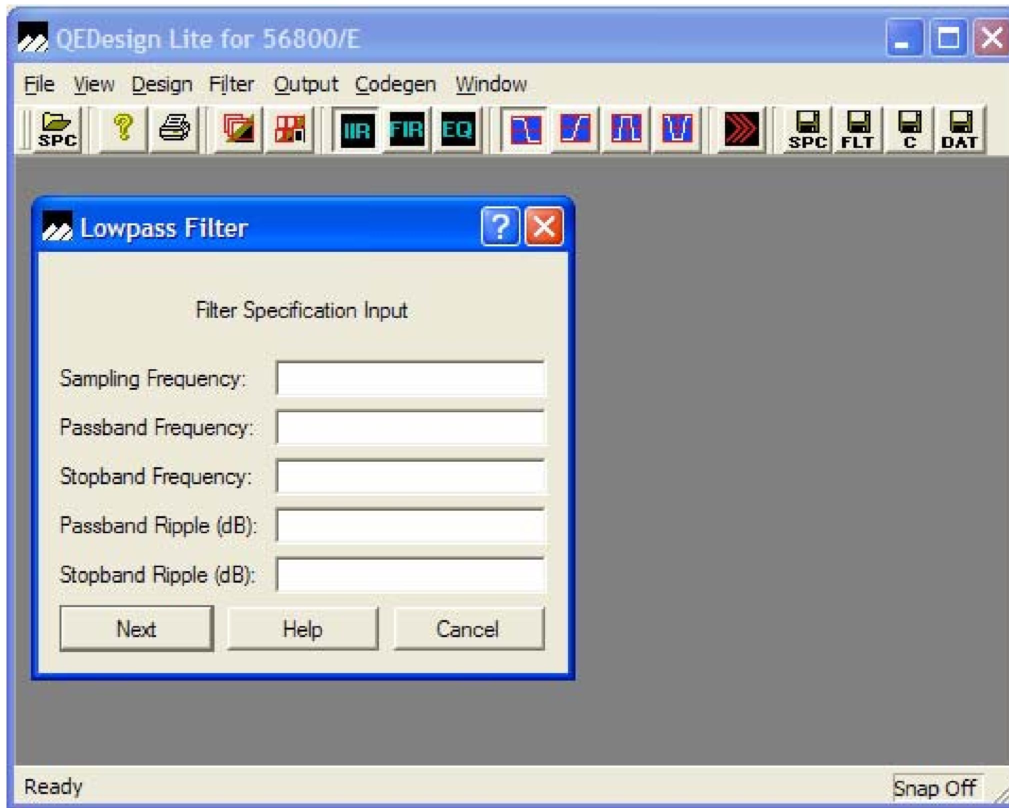



**Figure 7. QEDesign Lite IIR Filter Specification Input**

2. As shown in Figure 8, select the filter type according to the application requirement. You can select Butterworth.

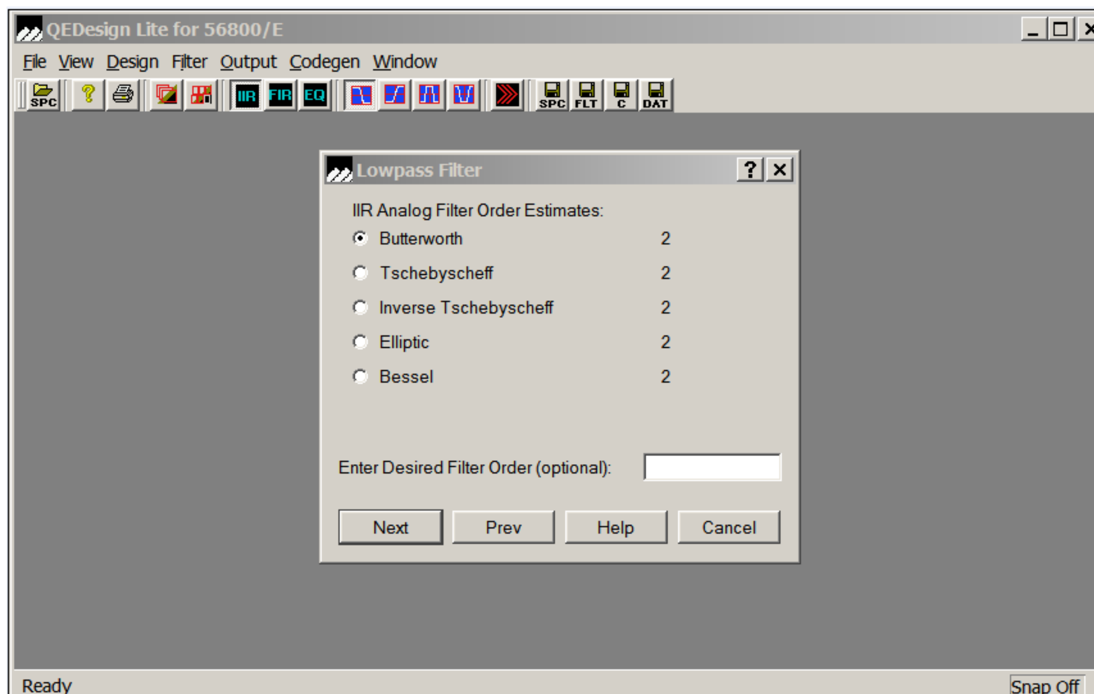



**Figure 8. QEDesign Lite IIR Filter Type Selection**

IIR filter features are:

- Butterworth filter: a Butterworth filter is a maximally flat filter and has no ripples in the passband or the stopband.
- Tschebyscheff filter: a Tschebyscheff filter has ripples in the passband but is monotonic in the stopband. (Tschebyscheff is frequently spelled Chebyshev, but in this application note we will use the same spelling as in the GUI.)
- Inverse Tschebyscheff filter: an inverse Tschebyscheff filter is the reverse of a Tschebyscheff filter, and is monotonic in the passband but has ripples in the stopband.
- Elliptic filter: an elliptic filter has ripples in the passband and in the stopband.
- Bessel filter: a Bessel filter is monotonic in both passband and stopband. It has no sharp rolloff characteristic and is primarily used for a constant group delay filter (using the impulse invariant method). It is an approximate linear phase filter.

Butterworth and Bessel filters require the highest filter orders. Tschebyscheff and inverse Tschebyscheff filters require fewer, and the elliptic filter requires the least.

After inputting the parameters, QEDesign Lite will generate the IIR coefficients and draw the curves for magnitude vs. frequency, phase vs. frequency, impulse response, step response, and pole/zero position maps, as shown in Figure 9.

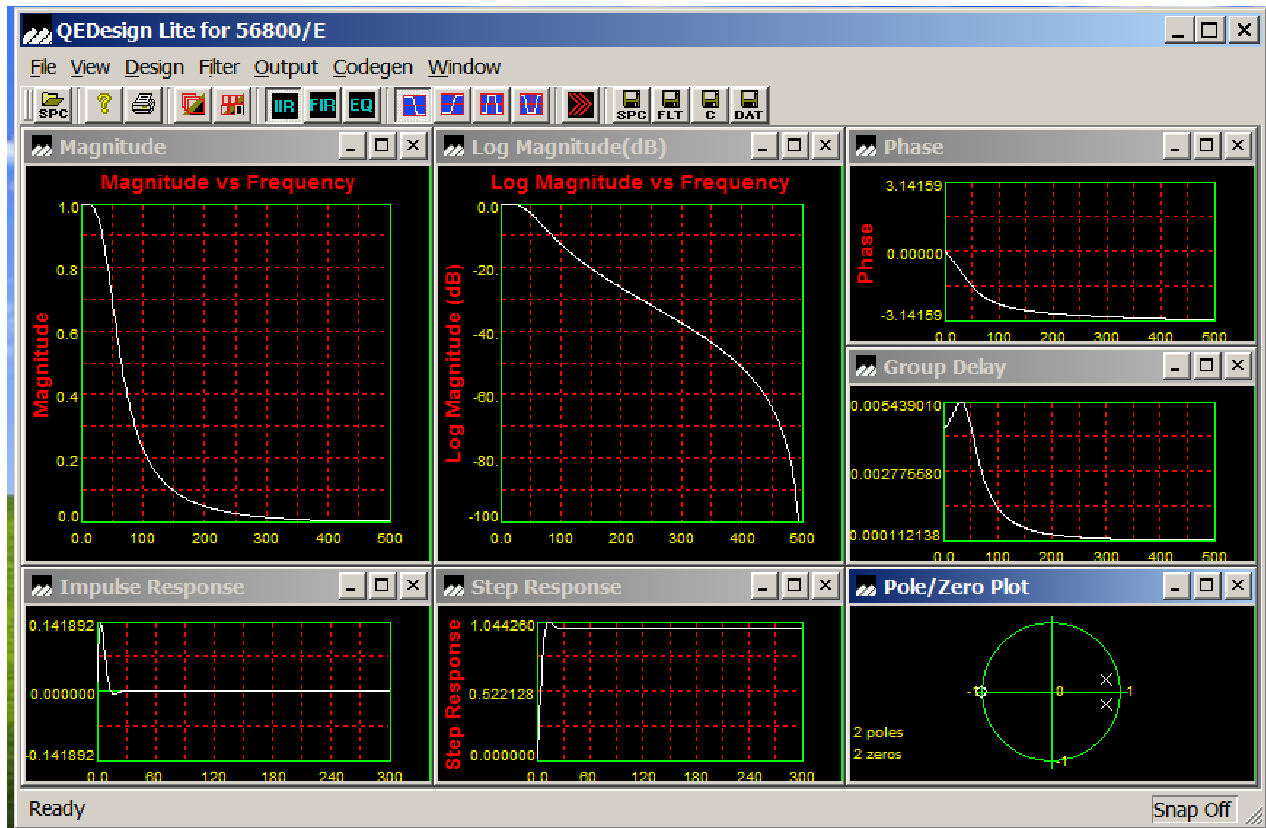



**Figure 9. QEDesign Lite IIR Filter Result Curve**

For a two-order IIR filter, the transfer function can be expressed as:

$$H(z) = [b_0 + b_1 \times z^{-1} + b_2 \times z^{-2}]/[1 - a_1 \times z^{-1} - a_2 \times z^{-2}] \qquad \textbf{Eqn. 22}$$

The magnitude versus frequency curve is expressed as:

$$H(jw) = \frac{b0 + b1 \times e^{-jw} + b2 \times e^{-2jw}}{1 - a1 \times e^{-jw} - a2 \times e^{-2jw}} \qquad \textbf{Eqn. 23}$$

The curve is computed by |H(jw)|/overall gain, the normalized frequency w is from 0 to π, and the actual frequency is from 0 to half of the ADC sample frequency.

We can sample w from 0 to π and take the step as π/100. Then we can draw the curve for magnitude versus frequency.

In Equation 23, w is a normalized frequency. The actual frequency corresponding to the normalized one is

$$F_{actual} = w \times F_{sample}/(2\pi) \qquad \textbf{Eqn. 24}$$

To determine the log magnitude versus frequency curve:

The amplitude corresponding to w is $20 \times \log_{10}(|H(jw)|/\text{Overall Gain})$.

The phase difference between output and input is

$$\text{Phase}(w) = \arctan\{\text{imag}[H(jw)]/\text{real}[H(jw)]\} \qquad \textbf{Eqn. 25}$$

Group delay is the delay time vs. frequency, which can be computed based on normalized frequency or actual frequency $F_{actual}$.

$$\text{Group delay time} = \text{Phase}(w)/w = \text{Phase}(w) / (2\pi F_{actual} / F_{sample}) \qquad \textbf{Eqn. 26}$$

The impulse response is the response to an impulse input. The filter can be treated as a linear system. Input an impulse to the system, and the output sequence will be the impulse response.

Input data sequence $x(n)= \begin{cases} 1, n= 0 \\ 0, n > 0 \end{cases}$ to

$$y(n) = a_1 \times y(n–1) + a_2 \times y(n–2) + b_0 \times x(n) + b_1 \times x(n–1) + b_2 \times x(n–2) \qquad \textbf{Eqn. 27}$$

Assume that history y(–1), y(–2), x(–1), and x(–2) all equal zero. We can compute the output sequence y(n).

The step response is the response for a step input. The filter can be treated as a linear system. Input a step sequence to the system, and the output sequence will be the step response.

Input data sequence x(n)={1, 1,1,1,1,1,1,1,1,1……} to Equation 27.

Assuming that input and output history y(–1), y(–2), x(–1), and x(–2) are all equal to zero, we can compute the output sequence y(n).

The zero/pole figure is derived from H(z).

$$H(z)= \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 - a_1 z^{-1} - a_2 z^{-2}} = \frac{b_0 z^2 + b_1 z^1 + b_2}{z^2 - a_1 z^1 - a_2} = \frac{(z - r_1)(z - r_2)}{(z - p_1)(z - p_2)} \qquad \textbf{Eqn. 28}$$

In the unit circle coordinator system, r1 and r2 are zero points, and p1 and p2 are pole points.

The IIR filter can be expressed in several forms. The normal format of an IIR filter is shown in Equation 29.

$$H(z)= \left(\sum_{r=0}^{M} b_r z^{-r}\right) \Big/ \left(1 - \sum_{k=1}^{N} a_k z^{-k}\right) \qquad \textbf{Eqn. 29}$$

It can also be expressed in another form using a two-order module for any IIR digital filter, as in Equation 30.

$$H(z)=\frac{b_{10}+b_{11}\times z^{-1}+b_{12}\times z^{-2}}{1-a_{11}\times z^{-1}-a_{12}\times z^{-2}}\times\frac{b_{20}+b_{21}\times z^{-1}+b_{22}\times z^{-2}}{1-a_{21}\times z^{-1}-a_{22}\times z^{-2}}\times\ldots \qquad \textbf{Eqn. 30}$$

Any IIR transfer function H(z) can be expressed as in the forms above. To use the IIR API function that Processor Expert provides, the user must denote H(z) in the format given here, if the user uses other tools to generate the IIR coefficients.

This application note picks only one item to analyze. Assume that

$$H1(z) = \frac{b_{10}+b_{11}\times z^{-1}+b_{12}\times z^{-2}}{1-a_{11}\times z^{-1}-a_{12}\times z^{-2}} \qquad \textbf{Eqn. 31}$$

To change Equation 31 into zero-pole format, we can change the function to

$$H1(z) = \frac{b_0+b_1\times z^{-1}+b_2\times z^{-2}}{1-a_1\times z^{-1}-a_2\times z^{-2}}=\frac{(z-r_1)(z-r_2)}{(z-p_1)(z-p_2)} \qquad \textbf{Eqn. 32}$$

In Equation 32, $p_1$ and $p_2$ are pole points, $r_1$ and $r_2$ are zero points. In a stable trans-function, the pole points $p_1$ and $p_2$ must be within a unit circle — that is, the absolute values of $p_1$ and $p_2$ must be less than 1. Also, $p_1$ and $p_2$ may be a pair of conjugate complex numbers or real numbers. If the absolute values of $p_1$ and $p_2$ are greater than 1, the trans-function is unstable or the output cannot converge to a value for a step input. So this application note discusses only the case in which the absolute values of $p_1$ and $p_2$ are less than 1.

The $a_1$ and $a_2$ restrictions for a stable system will be deduced as in Equation 33.

$$z^2-a_1\times z-a_2 = (z-p_1)(z-p_2) \qquad \textbf{Eqn. 33}$$

Therefore $a_1$ and $a_2$ can be defined as in Equation 34.

$$a_1 = p_1+p_2;\ a_2 = -p_1\times p_2 \qquad \textbf{Eqn. 34}$$

The value of $a_1$ may be greater than 1, but must be less than 2; $a_2$ must be less than 1.

Because devices in the DSP56800/E family are fixed-point processors, they are suitable for fixed-point computation. They cannot express a number greater than one using the fixed-point data type, so scale is required in case the IIR coefficients are greater than 1.

### 3.2.2 How to Implement the IIR Filter

For a two-order IIR filter

$$H(z) = \frac{b_0+b_1\times z^{-1}+b_2\times z^{-2}}{1-a_1\times z^{-1}-a_2\times -z^{-2}} \qquad \textbf{Eqn. 35}$$

assume that the input sequence is x(n). Then the output sequence is y(n), and its corresponding Z transformation is X(z) and Y(z).

$$y(n) = a_1 \times y(n-1) + a_2 \times y(n-2) + b_0 \times x(n) + b_1 \times x(n-1) + b_2 \times x(n-2) \qquad \textit{Eqn. 36}$$

We can use another format to implement the IIR filter:

$$Y(z) = H(z) \times X(z) = \frac{b_0 + b_1 \times z^{-1} + b_2 \times z^{-2}}{1 - a_1 \times z^{-1} - a_2 \times -z^{-2}} \times X(z) \qquad \textit{Eqn. 37}$$

We introduce a temporary variable W(z) and let

$$W(z) = \frac{X(z)}{1 - a_1 \times z^{-1} - a_2 \times z^{-2}} \qquad \textit{Eqn. 38}$$

and

$$Y(z) = (b_0 + b_1 z^{-1} + b_2 z^{-2}) \times W(z) \qquad \textit{Eqn. 39}$$

For each biquad calculation:

$$w[n] = x[n] + a_1 w[n-1] + a_2 w[n-2] = 2.0 \times \left(\frac{x[n]}{2.0} + \frac{a_1}{2.0} \times w[n-1] + \frac{a_2}{2.0} \times w[n-2]\right) \qquad \textit{Eqn. 40}$$

$$y[n] = 2.0 \times \left(\frac{b_0}{2.0} \times w[n] + \frac{b_1}{2.0} \times w[n-1] + \frac{b_1}{2.0} \times w[n-2]\right) \qquad \textit{Eqn. 41}$$

The C code is located in the directory:

\Code\PE_exampleDownloadFromPE\IIR_Lab|PE_Based|56858\dfr16IIRC_Fixed.c

## 3.3 Illustration of the Coefficients from QEDesign Lite

This application note adopts a low pass Butterworth filter using the QEDesign Lite software.

Use the procedure introduced in Section 3.1, “FIR Filter,” or Section 3.2, “IIR Filter Design.” Input your parameters, then click “OUTPUT” -> ”Create Filter Coefficients File.” A file will be generated.

### 3.3.1 Example Coefficients in IIR Output File

This is the contents of the IIR filter output file. It lists the filter type, the filter parameters that the user has input, and the IIR coefficients generated by QEDesign Lite.

```
FILTER COEFFICIENT FILE
IIR DESIGN
FILTER TYPE                  LOW PASS
```

```
ANALOG FILTER TYPE            BUTTERWORTH
PASSBAND RIPPLE IN -dB          -.3000E+01
STOPBAND RIPPLE IN -dB          -.1000E+02
PASSBAND CUTOFF FREQUENCY     0.500000E+02 HERTZ
STOPBAND CUTOFF FREQUENCY     0.100000E+03 HERTZ
SAMPLING FREQUENCY            0.100000E+04 HERTZ
FILTER DESIGN METHOD: BILINEAR TRANSFORMATION
FILTER ORDER                2     2h
NUMBER OF SECTIONS          1     1h
NO. OF QUANTIZED BITS   16    10h
QUANTIZATION TYPE - FRACTIONAL FIXED POINT
COEFFICIENTS SCALED FOR CASCADE FORM II
         -3 FFFFFFFD     /* shift count for overall gain */
      21096     5268     /* overall gain                 */
          1        1     /* shift count for section  1 values */
       4097     1001     /* section  1 coefficient B0 */
       8194     2002     /* section  1 coefficient B1 */
       4097     1001     /* section  1 coefficient B2 */
      25567     63DF     /* section  1 coefficient -A1*/
     -10502 FFFFD6FA     /* section  1 coefficient -A2*/
  0.1250305175781250E+00 3FC0010000000000 0.25006104E+00 /* section  1 B0 */
  0.2500610351562500E+00 3FD0010000000000 0.50012207E+00 /* section  1 B1 */
  0.1250305175781250E+00 3FC0010000000000 0.25006104E+00 /* section  1 B2 */
  0.7802429199218750E+00 BFE8F7C000000000 0.15604858E+01 /* section  1 -A1*/
  -.3204956054687500E+00 3FD4830000000000 -.64099121E+00 /* section  1 -A2*/
```

### 3.3.2 Explanation of the IIR Coefficients

This section details the IIR coefficients and the overall gain of the IIR filter.

The gain of the filter varies with the frequency. The overall gain is the gain of the filter in all frequency ranges, so the maximum gain of the filter can be one if we divide the actual filter gain by the overall gain of the filter.

According to the overall gain parameter:

```
-3 FFFFFFFD                /* shift count for overall gain */
21096     5268              /* overall gain */
```

The overall gain of the IIR filter is $1/(21096\times2^{-3}/32768)$.

For the IIR coefficient:

```
          1        1     /* shift count for section  1 values */
       4097     1001     /* section  1 coefficient B0 */
       8194     2002     /* section  1 coefficient B1 */
       4097     1001     /* section  1 coefficient B2 */
      25567     63DF     /* section  1 coefficient -A1*/
     -10502 FFFFD6FA     /* section  1 coefficient -A2*/
  0.1250305175781250E+00 3FC0010000000000 0.25006104E+00 /* section  1 B0 */
  0.2500610351562500E+00 3FD0010000000000 0.50012207E+00 /* section  1 B1 */
  0.1250305175781250E+00 3FC0010000000000 0.25006104E+00 /* section  1 B2 */
  0.7802429199218750E+00 BFE8F7C000000000 0.15604858E+01 /* section  1 -A1*/
  -.3204956054687500E+00 3FD4830000000000 -.64099121E+00 /* section  1 -A2*/
```

For the shift count of the IIR coefficient parameters:

```
1        1     /* shift count for section  1 values */
```

In the filter coefficient file, the IIR coefficients are scaled down by a factor of $2^{ShiftCount}$. In this case, the IIR coefficients are scaled down by a factor of two.

The IIR coefficients are these values:

$B0 = 4097 \times 2^1 = 8194$ or 8194/32768=0.25

$B1 = 8194 \times 2^1 = 16388$ or 16388/32768=0.5

$B2 = 4097 \times 2^1 = 8194$ or 8184/32768=0.25

$A1 = 25567 \times 2^1 = 51134$ or 51134/32768=1.56
(this cannot be expressed as a fixed-point value because it is greater than one)

$A2 = –10502 \times 2^1 = –21004$ or –21004/32768 = –0.640991

## 3.3.3 Transfer Function of the IIR Filter

### 3.3.3.1 IIR Transfer Function

The transfer function is

$$H(z) = \frac{b_0 + b_1 \times z^{-1} + b_2 \times z^{-2}}{1 - a_1 \times z^{-1} - a_2 \times z^{-2}} = \frac{0.250061 + 0.500122 \times z^{-1} + 0.250061 \times z^{-2}}{1 - 1.560486 \times z^{-1} + 0.640991 \times z^{-2}} \quad \text{Eqn. 42}$$

### 3.3.3.2 IIR Transfer Function Gain

For a transfer function, the gain changes with frequency. But for the above low-pass Butterworth filter, the overall gain parameter of the filter can be taken at the DC component, because the amplitude versus frequency curve of a Butterworth filter decreases monotonically.

The normalized frequency of the DC component w=0, so z=exp(jw)=1. Therefore we can compute the gain of the filter with z=1.

The gain of the filter at the DC component is the same as in Equation 42, which computes as

(0.250061 + 0.500122×1+0.250061×1)/(1–1.560486+0.640991)=1.00024/0.08051
=1/0.080490=12.424

For a step input function, if the value is x, the output value of the filter will converge to 12.424x after the interim process is finished.

The user hopes that the gain of the filter is 1 — thus, QEDesign Lite will give an overall gain and corresponding shift count specification, which can be used to scale down the filter output. In the above low-pass Butterworth filter, the overall gain is 21906, and the shift count for overall gain is –3. Hence, the scale is

$21096\times 2^{-3}$=21096/8=2637

If we express the scale as a fixed-point data type, the value is 2637/32767=0.08048, which exactly equals 1/12.424.

For a filter whose gain is not 1, the user must pay attention to the saturation issue in real-time filtering by using a DSP chip:

- Scale down the input sequence using the scale value given above. If the user has scaled down the input sequence, it is unnecessary to scale down the filter output. It needs to be scaled down only once.
- Set the SAT bit in the DSP56F800 and DSP56800/E core. When saturation is enabled, the IIR function will return the maximum or minimum fractional values if overflow or underflow occurs during the calculation of each output element.

#### 3.3.3.3 Pole Point Calculation

We can get the pole point from this equation:

$$z^2 - 1.560486 \times z + 0.640991 = 0$$

So $p_1 = 0.78 + j0.18$

and $p_2 = 0.78 - j0.18$

#### 3.3.3.4 Establishing Coefficient Array in PE

Firstly, QEDesign Lite designs the filter according to specifications the user defines, and gives a cascade filter of biquad coefficients as in Equation 43.

$$H(z) = \frac{b_{10} + b_{11} \times z^{-1} + b_{12} \times z^{-2}}{1 - a_{11} \times z^{-1} - a_{12} \times z^{-2}} \times \frac{b_{20} + b_{21} \times z^{-1} + b_{22} \times z^{-2}}{1 - a_{21} \times z^{-1} - a_{22} \times z^{-2}} \quad \text{Eqn. 43}$$

For the input vector of biquadratic coefficients, the coefficients can be organized in an array as shown below, where the subscript i designates the i-th biquad polynomial ($0 \le i < n$): const IirCoefs []=

{(a1,0)/2 (a2,0)/2 (b0,0)/2 (b1,0)/2 (b2,0)/2... (a1,i)/2 (a2,i)/2 (b0,i)/2 (b1,i)/2 (b2,i)/2... (a1,n)/2 (a2,n)/2 (b0,n)/2 (b1,n)/2 (b2,n)/2……}

For example, using the coefficients generated in the filter coefficient file, you can declare a coefficient array.

```
Frac16 IirCoefs[] = {  0xD6FA, //{-a1/2 section 1}
  0x63DF, //{-a2/2 section 1}
  0x1001, //{+b0/2 section 1}
  0x2002, //{+b1/2 section 1}
  0x1001}; //{+b2/2 section 1}
```

Because there is only one biquad polynomial, IIR coefficients consist of only five items. If there are two biquad polynomials, the IIR coefficients will consist of ten items.

## 3.4 How to Call the IIR API Function

There are two methods to initialize the IIR structure. One is to use the DFR1_dfr16IIRInit() API function to initialize the IIR filter structure parameters and allocate memory statically. The other is to use the

dfr16IIRCreate() API function to initialize the IIR filter structure parameters and allocate memory dynamically.

### 3.4.1 IIR Filter Implementation Using Static Memory Allocation

This section explains how to call DFR1_dfr16IIRInit() and allocate memory statically.

1. Declare a history array to save temporary variables.

   For an IIR filter, we must allocate memory to save the old values of an output sequence. For a two-order IIR filter, assume the output sequence is y[k]. We must save y[k–1] and y[k–2] in temporary history array variables. The size of the history array must be BIQUAD_NUMBER × FILT_STATES_PER_BIQ.

2. Declare an array to save IIR filter coefficients.

   These constants are declared by the Processor Expert software in dfr16.h:

```
#define FILT_STATES_PER_BIQ 2 //the order of an IIR filter
#define FILT_COEF_PER_BIQ 5
```

   This is the code for a two-order IIR filter.

```
#define BIQUAD_NUMBER  1 // the number of biquad polynomial
#define NUM_SAMPLES_IIR 100 /*Number of incoming samples */
#define HISTORY_BUFFER_SIZE BIQUAD_NUMBER  *FILT_STATES_PER_BIQ
#define IIR_COEF_LENGTH BIQUAD_NUMBER*FILT_COEF_PER_BIQ
Const Frac16 IirCoefs[] = {  0xD6FA, //{-a1/2 section 1}
  0x63DF, //{-a2/2 section 1}
  0x1001, //{+b0/2 section 1}
  0x2002, //{+b1/2 section 1}
  0x1001}; //{+b2/2 section 1}

Frac16 history[ HISTORY_BUFFER_SIZE * sizeof(Frac16)];
Frac16 iirCoefArray[IIR_COEF_LENGTH* sizeof(Frac16)];
/* IIR Structures */
dfr16_tIirStruct Iir;
dfr16_tIirStruct *pIir = &Iir;
Frac16* pIirCoefs;
Frac16 x[NUM_SAMPLES_IIR]; //input data sequence before filter
Frac16 zIIR[NUM_SAMPLES_IIR]; //output data sequence after filter
Result res; //Result data type is alias of int

main()
{
 /*** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!! ***/
  PE_low_level_init();
  /*** End of Processor Expert internal initialization.                    ***/

pIir -> pC=( Frac16 *)&iirCoefArray[0];
pIirCoefs =( Frac16 *)&IirCoefs[0];
pIir -> pHistory =( Frac16 *)&history[0];
/* Initialize the IIR filter */
DFR1_dfr16IIRInit(pIir,pIirCoefs,sizeof(IirCoefs)/(sizeof(Frac16)*FILT_COEF_PER_BIQ));
/*Run the IIR filter (C version) */
res = DFR1_dfr16IIR(pIir, x, zIIR, NUM_SAMPLES_IIR);
}
```

### 3.4.2 IIR Filter Implementation Using Dynamic Memory Allocation

This section explains how to call DFR1_ dfr16IIRCreate() and allocate memory dynamically.

The user must declare a pointer of dfr16_tIirStruct structure type so that the dfr16IIRCreate(Frac16 *pC, UInt16 nbiq) function can initialize the pointer. The PC pointer must point to the array of IIR coefficients; the nbiq parameter is the number of biquad polynomials; the return value of the API function is the dfr16_tIirStruct type pointer.

This application note uses the code given in this section to implement an IIR filter.

The code generates a step input response of the IIR filter, so the input sequence is a constant sequence. The pZ array is used to save the result of the filter.

The constant coeff_IIR_83_gain is the reciprocal of the filter's overall gain and is expressed by the Frac16 data type. (Because the filter overall gain is greater than 1, the reciprocal of the filter overall gain can be expressed by a Frac16 data type.) We scale down the input sequence by the coeff_IIR_83_gain so that the filter output cannot saturate.

Frac16 IirCoefs[] is the coefficient array of the IIR filter.

The pointer dfr16_tIirStruct *pIIR is a struct pointer without initialization, which must be initialized by the dfr16IIRCreate() function.

Frac16 *pC is a pointer, and points to the IIR coefficient array.

UInt16 n is the input data sequence length and nbiq is the number of biquad polynomials.

If memory is allocated dynamically, the user must modify the link command file. In the next section, we will explain how to change the link command file.

```
#define STEP_VALUE0 0.3

Frac16 stepWave[]=
{
        FRAC16(STEP_VALUE0),
        FRAC16(STEP_VALUE0),
        FRAC16(STEP_VALUE0),
…………
}

Frac16 pZ[100];

const Frac16 coeff_IIR_83_gain = 2637; //{overall gain of the filter = 21096/8}

        Frac16 IirCoefs[] = {  0xD6FA, //{-a1/2 section 1}
                                                    0x63DF, //{-a2/2 section 1}
                                                    0x1001, //{+b0/2 section 1}
                                                    0x2002, //{+b1/2 section 1}
                                                    0x1001}; //{+b2/2 section 1}
dfr16_tIirStruct *pIIR;
Frac16 *pC;
Result res;
Frac16 *pX;
UInt16 n, nbiq;
```

```
extern Result dfr16IIRC_Fixed(dfr16_tIirStruct *pIIR, Frac16 *pX, Frac16 *pZ, UInt16 n);

void main(void)
{
         unsigned int i;
  /*** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!! ***/
  PE_low_level_init();
  /*** End of Processor Expert internal initialization.                    ***/


pC=&IirCoefs[0];
nbiq=1;
pX=&stepWave[0];
n=100;
for(i=0; i<n; i++)
{
                  pX[i] =  mult(coeff_IIR_83_gain,pX[i]);
}
pIIR = dfr16IIRCreate (pC, nbiq);
if(pIIR==NULL) asm(debug)
res = dfr16IIRC_Fixed(pIIR, pX, pZ, n);
asm(nop);

asm(nop); //set break point here
dfr16IIRDestroy (pIIR);
return;
}
```

Both the “Result” data type and macro “FRAC16()” are defined in port.h in PE, which is listed here:

```
typedef int             Result;
#define FRAC16(x) ((Frac16)((x) < 1? ((x) >= -1 ? (x)*0x8000 : 0x8000) : 0x7FFF))
```

## 3.5 Modify the Link Command File for Dynamic Memory Allocation

The DSP56F805 linker command file here is generated by CodeWarrior for DSP56800/E version 8.1, then modified. The internal data RAM of the DSP56F805 device is from X:0x0000 to X:0x0800. The .x_Data section memory size is reduced from 0x7C0 to 0x4C0; the .xDynamic section is added by the user manually, and the data memory space from 0x0500 to 0x0800 is allocated for this section; the LENGTH of the section is 0x300 in MEMORY{}.

In Mem1.c file, we use the void MEM1_Init(void) function to initialize the dynamic memory. In this function, two structures, mem_sState and mem_sPartition, have been defined in advance. The variable InitialState with a mem_sState structure type is used to save the dynamic memory characteristic parameters. The variables defined in the link command file are used to initialize the variable InitialState. All variables defined in the linker command file are prefixed by F. For example, the FmemEXbit variable defined in the link command file is actually the memEXbit variable, which can be accessed in any function as a global variable. If the user uses the link command file given below, the variable InitialState with a mem_sState structure data type is initialized using this code in the void MEM1_Init(void) function.

```
InitialState.EXbit = memEXbit;
  InitialState.numExtPartitions = memNumEMpartitions;
  InitialState.numIntPartitions = memNumIMpartitions;
  InitialState.extPartitionList = (mem_sPartition*)&memEMpartitionAddr;
  InitialState.intPartitionList = (mem_sPartition*)&memIMpartitionAddr;
```

After the above code is executed, the InitialState will be:

- InitialState.EXbit = 0
- InitialState.numExtPartitions = 0
- InitialState.numIntPartitions = 1
- InitialState.extPartitionList = 0
- InitialState.intPartitionList = 0x0da; //the pointer pointing to a mem_sPartition data structure that saves the internal dynamic memory address and data length

Therefore:

- InitialState.intPartitionList -> partitionAddr = 0x0500
- InitialState.intPartitionList -> partitionSize = 0x0300

Note that the variables defined in the link command file are saved in flash and copied to RAM by application code. Thus the dynamic memory characteristic parameters become nonvolatile and the DSP can retrieve the dynamic memory address and size in standalone running mode.

```
typedef struct
{
  UInt16                   EXbit;
  UInt16                   numExtPartitions;
  UInt16                   numIntPartitions;
  const mem_sPartition * intPartitionList;
  const mem_sPartition * extPartitionList;
} mem_sState
and
typedef struct
{
  void *    partitionAddr;
  UInt16    partitionSize;
} mem_sPartition

mem_sState InitialState;

void MEM1_Init(void)
{
  mem_sState InitialState;
                                        /* These variables are defined in linker.cmd */
  extern UInt16 memEXbit;
  extern UInt16 memNumEMpartitions;
  extern UInt16 memNumIMpartitions;
  extern char * memIMpartitionAddr;
  extern char * memEMpartitionAddr;

  InitialState.EXbit = memEXbit;
  InitialState.numExtPartitions = memNumEMpartitions;
  InitialState.numIntPartitions = memNumIMpartitions;
  InitialState.extPartitionList = (mem_sPartition*)&memEMpartitionAddr;
  InitialState.intPartitionList = (mem_sPartition*)&memIMpartitionAddr;
  memInitialize(&InitialState);
}
```

In this application, the link command file is modified as shown below. The lines that are bold/italic must be added by the programmer.

```
MEMORY {
 # I/O registers area for on-chip peripherals
 .x_Peripherals (RW)   : ORIGIN = 0x0C00, LENGTH = 0
        .x_CoreRegs    (RW)   : ORIGIN = 0xFF80, LENGTH = 0
        # List of all sections specified in the "Build options" tab
        #Internal vector boot area.
        .p_Interruptsboot  (RWX) : ORIGIN = 0x00008000, LENGTH = 0x0004
        .p_Interrupts  (RWX) : ORIGIN = 0x00000000, LENGTH = 0x00000080
        .p_Code  (RWX) : ORIGIN = 0x00000080, LENGTH = 0x00007D80
        .x_Data  (RW) : ORIGIN = 0x00000040, LENGTH = 0x000004C0
        .xDynamic(RW) :ORIGIN = 0x00000500, LENGTH = 0x00000300

        .x_CWRegisters  (RW) : ORIGIN = 0x00000030, LENGTH = 0x00000010

        #Other memory segments
        .x_internal_ROM  (RW)  : ORIGIN = 0x00001000, LENGTH = 0x1000
        .p_internal_RAM  (RWX) : ORIGIN = 0x00007E00, LENGTH = 0x0200
}
KEEP_SECTION { interrupt_vectorsboot.text }
KEEP_SECTION { interrupt_vectors.text }
SECTIONS {
        .interrupt_vectorsboot :
        {
          F_vector_addr = .;
          # interrupt vectors boot area
          * (interrupt_vectorsboot.text)
        } > .p_Interruptsboot

        .interrupt_vectors :
        {
          # interrupt vectors
          * (interrupt_vectors.text)
        } > .p_Interrupts
        .Data_xROM :
        {
              * (.rodata)
              # save address for ROM data we will copy to RAM
              __xROM_data_start =.;
        } > .x_internal_ROM
        .ApplicationCode
        {
              F_Pcode_start_addr = .;
              # .text sections
             OBJECT (F_EntryPoint, Cpu.c) # The function _EntryPoint have to be placed at the
beginning of the code
                                  # section for proper functionality of the serial bootloader.
              * (.text)
              * (rtlib.text)
              * (startup.text)
              * (fp_engine.text)
              * (user.text)

              F_Pbss_start_addr = .;
              _P_BSS_ADDR = .;
              F_Pbss_length = . - _P_BSS_ADDR;
              F_Pcode_end_addr = .;
        } > .p_Code
```

```
.ApplicationData :  AT(__xROM_data_start)
{
      __xRAM_data_start =.;
      * (fp_state.data)
      * (rtlib.data)
      * (.data.char)        # used if "Emit Separate Char Data Section" enabled
      * (.data)
      _EX_BIT = 0;
      # Internal Memory Partitions (for mem.h partitions)
      _NUM_IM_PARTITIONS = 1;
      # External Memory Partition (for mem.h partitions)
      _NUM_EM_PARTITIONS = 0;
      FmemEXbit = .;
      WRITEH(_EX_BIT);
      FmemNumIMpartitions = .;
      WRITEH(_NUM_IM_PARTITIONS);
      FmemNumEMpartitions = .;
      WRITEH(_NUM_EM_PARTITIONS);
      FmemIMpartitionAddr = .;
      WRITEH(ADDR(.xDynamic));
      FmemIMpartitionSize = .;
      WRITEH(SIZEOF(.xDynamic) / 2);
      FmemEMpartitionAddr = 0;
      WRITEH(0);
      FmemEMpartitionSize = .;
      WRITEH(0);
      __xRAM_data_end =.;
      __data_size = __xRAM_data_end - __xRAM_data_start;
      # .bss sections
      * (rtlib.bss.lo)
      * (rtlib.bss)
      . = ALIGN(4);
      F_Xbss_start_addr = .;
      _START_BSS = .;
      * (.bss.char)         # used if "Emit Separate Char Data Section" enabled
      * (.bss)
      _END_BSS   = .;
      F_Xbss_length = _END_BSS - _START_BSS;
      /* Setup the HEAP address */
      . = ALIGN(4);
      _HEAP_ADDR = .;
      _HEAP_SIZE = 0x00000100;
      _HEAP_END = _HEAP_ADDR + _HEAP_SIZE;
      . = _HEAP_END;
      /* SETUP the STACK address */
      _min_stack_size = 0x00000200;
      _stack_addr = _HEAP_END;
      _stack_end  = _stack_addr + _min_stack_size;
      . = _stack_end;
      /* EXPORT HEAP and STACK runtime to libraries */
      F_heap_addr   = _HEAP_ADDR;
      F_heap_end    = _HEAP_END;
      F_Lstack_addr = _HEAP_END;
      F_StackAddr = _HEAP_END;
      F_StackEndAddr = _stack_end - 1;
      # runtime code __init_sections uses these globals:
      F_Ldata_size     = __data_size;
```

```
            F_Ldata_RAM_addr = __xRAM_data_start;
            F_Ldata_ROM_addr = __xROM_data_start;
            F_xROM_to_xRAM   = 0x0001;
            F_pROM_to_xRAM   = 0x0000;
            F_start_bss    = _START_BSS;
            F_end_bss      = _END_BSS;
            __DATA_END=.;
        } > .x_Data

        # peripheral registers
        FCoreIO = ADDR(.x_CoreRegs);   /* Core registers */
}
}
```

## 3.6 How to Call the FIR API Function

After getting FIR coefficients from QEDesign Lite, users can call the FIR API function to implement the filter provided by Processor Expert.

$$y(n) = \sum_{k=0}^{N-1} h(k) \times x(n-k)$$

According to the convolution formula, to get one set of output data y(n), we need the h(n) series and the history of input x(n).

$h(0), h(1), h(2), h(3), \ldots\ldots\ldots\ldots h(N-1)$

$x(m), x(m-1), x(m-2), x(m-3), \ldots\ldots\ldots\ldots x(m-N+1)$

$y(n) = h(0) \times x(n) + h(1) \times x(n-1) + h(2) \times x(n-2) + h(3) \times x(n-3) + \ldots + h(N-1) \times x(n-N+1)$

To compute y(n), Processor Expert defines the structure dfr16_tFirStruct as:

```
typedef struct dfr16_sFirStruct {
 Frac16    * pC;                   /* Coefficients for the filter */
 Frac16    * pHistory;             /* Memory for the filter history buffer */
 UWord16     Private[6];
} dfr16_tFirStruct
```

In the above structure, the PC pointer points to the FIR coefficient array. The pHistory pointer points to a temporary array to save the history of the input data sequence. The private array is private data.

### 3.6.1 FIR Filter Implementation Using Static Memory Allocation

We must declare a variable with the dfr16_tFirStruct data type before calling the API function to implement the FIR filter. There are two methods to initialize and allocate memory for the dfr16_tFirStruct structure type variable. One is to statically allocate memory for the variable with the dfr16_tFirStruct

structure data type; the other is to use it to dynamically allocate memory for the variable. From a functional perspective, they operate in the same way to allocate memory for a variable with a structure data type.

Here is how the user will statically allocate memory for the dfr16_tFirStruct structure type.

1. Define a variable with the dfr16_tFirStruct type as dfr16_tFirStruct fir.
2. Statically allocate two arrays to save the coefficients and the input series x(n) history.
   — Frac16 firCoefArray[FIR_COEF_LENGTH * sizeof(Frac16)]
   — Frac16 history[FIR_COEF_LENGTH * sizeof(Frac16)] /*unaligned */
3. Define a Frac16 type pointer, Frac16 *pFirCoefs, so that the start address of the coefficient array can be given to the pointer variable.
4. Call the DFR1_dfr16FIRInit() API function to initialize the variable with the dfr16_tFirStruct structure data type.
5. Call DFR1_dfr16FIR() to implement the digital filter.

The user can filter either one sample or a block of samples at a time.

In real-time filtering, this is the general scenario: the system consists of an ADC, a DSP, and a DAC. After the ADC samples new data which can generate an interrupt, the DSP can enter the ADC interrupt service routine (ISR), then read the sample and implement the digital filter. Then it can output the digital filter output to the DAC.

This is an FIR example for either block data filtering or single-sample data filtering, developed under Processor Expert and CodeWarrior for DSP56800/E version 7.3.

In the example, the on-chip ADC is used to sample an external analog signal. In the ISR of the end-of-scan routine of the on-chip ADC of DSP56F80x, a boolean data type flag, variable adc_flag, is set to indicate that a new sample is available. The DSP polls the flag in the main function. When it detects that the flag value represents “true,” the DSP will clear the flag and implement the FIR digital filter.

```
#define FIR_COEF_LENGTH 10 /*Number of filter taps */
#define NUM_SAMPLES 150 /*Number of incoming samples */
#define CEF(x) x*32767
//#define BLOCK
const int FirCoefs[]=
{
-1953,-1435,1006,4646,7904,9207,7904,4646,1006,-1435,-1953
};

dfr16_tFirStruct fir;
Frac16 *pFirCoefs;
Frac16 firCoefArray[FIR_COEF_LENGTH * sizeof(Frac16)];
Frac16 history[FIR_COEF_LENGTH * sizeof(Frac16)]; /*unaligned */
//#ifndef BLOCK
bool adc_flag=FALSE;
Frac16 sample0;
Frac16 filter_result0;
//#endif

void main(void)
{

  /*** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!! ***/
```

```
  PE_low_level_init();
  /*** End of Processor Expert internal initialization.                    ***/
         AD1_EnableIntTrigger();

  /* Write your code here */
    /* Initialize FIR filter */
         fir.pC =(Frac16 *)&firCoefArray[0];
         fir.pHistory =(Frac16 *)&history[0];
         pFirCoefs =(Frac16 *)&FirCoefs[0];
         DFR1_dfr16FIRInit((dfr16_tFirStruct*)&fir,pFirCoefs,FIR_COEF_LENGTH);
//sizeof(FirCoefs)/sizeof(Frac16));
//       dfr16FIRHistory(pFir,&x[0]); //fill the pipeline,
//       DFR1_dfr16FIR((dfr16_tFirStruct*)&fir, &x[0], &z1[0], NUM_SAMPLES);
         asm(nop); //set break here
         asm(nop);
  while(1)
  {
  #ifndef BLOCK
                  if(adc_flag)
                  {
                           DFR1_dfr16FIR((dfr16_tFirStruct*)&fir, &sample0, &filter_result0,
1);
                           adc_flag=FALSE;
                  }
  #endif
  }

}
```

### 3.6.2 FIR Filter Implementation Using Dynamic Memory Allocation

In dynamic memory allocation, the variable with the struct dfr16_tFirStruct data type is allocated memory space during run-time rather than before the code is executed.

The firCreat() API function dynamically initializes and allocates memory to the variable with the dfr16_tFirStruct data structure, which is used by FIR to preserve the filter's state values. To optimize the performance of the FIR, firCreate() allocates the dfr16_tFirStruct.pC buffer to hold the FIR filter coefficients.

Next, firCreate() allocates the buffer dfr16_tFirStruct.pHistory to save the past history of input x(n) data elements required by the FIR filter computation. Its length is the tap number of the FIR filter–1. The firCreate() function tries to allocate this history buffer aligned on a kbit boundary, where $k = \log_{10}2(N)$ and N is the FIR tap number, so that FIR can achieve maximum efficiency using modulo addressing of pFIR's history buffer.

After the dfr16_tFirStruct structure and its component buffers have been allocated, the user can call the FIR filter, dfr16FIRC().

The dfr16_tFirStruct is defined in dfr16.h as:

```
typedef struct dfr16_sFirStruct {
 Frac16    * pC;                  /* Coefficients for the filter */
 Frac16    * pHistory;            /* Memory for the filter history buffer */
 UWord16     Private[6];
} dfr16_tFirStruct
```

Use this procedure to implement the FIR filter using dynamic memory allocation:

1. Design the FIR filter using QEDesign Lite and get the FIR coefficients according to the steps described in Section 3.1, “FIR Filter.”
2. Create a project and add the beans DSP_Func_DFR by selecting *SW -> DSP Function&Math Library -> Digital Signal Processing Library -> DSP_Func_DFR* in the Processor Expert menu.
3. Add the DSP_MEM bean by selecting *SW -> DSP Function&Math Library -> Memory Manager -> DSP_MEM*.
4. Select *Processor Expert -> Generate code*… All the files will be generated automatically.
5. In the main function, define a constant array as a global array to save the FIR coefficients — for example, const int FirCoefs[]={ –1953,–1435,……}. The array will be saved in flash because of the prefix “constant,” and a copy is created in internal or external RAM.
6. Define a dfr16_tFirStruct type pointer, such as dfr16_tFirStruct* pFir, so that the user can initialize the structure using the dfr16FIRCreate() API function.
7. Define a Frac16 type pointer to point to the FIR coefficient array, such as Frac16 *pFirCoefs=(Frac16 *)&FirCoefs[0].
8. Initialize the dfr16_tFirStruct type pointer using the dfr16FIRCreate() API function in main, such as pFir = dfr16FIRCreate (pFirCoefs, FIR_COEF_LENGTH).
9. To perform block filtering, use this function:

   dfr16FIRC((dfr16_tFirStruct*)pFir, x, z11,NUM_SAMPLES) — x and z11 are arrays with Frac16 data type. To use a single-sample data filter, use dfr16FIRC((dfr16_tFirStruct*)pFir,&sig_x,&sig_z11,1). Here, sig_x and sig_z11 are Frac16 data type single variables.
10. Modify the linker command file to allocate the memory for the above structure. The dynamic memory start address and size must be saved into flash for the DSP to allocate memory in standalone mode.

```
#define FIR_COEF_LENGTH 11 /*Number of filter taps */
#define NUM_SAMPLES 150 /*Number of incoming samples */
#define CEF(x) x*32767
#define BLOCK

Frac16 x[NUM_SAMPLES]=
{
        CEF(0.5),
        CEF(0.5),
        CEF(0.5),
        CEF(0.5),
        CEF(0.5),
        CEF(0.5),
        CEF(0.5),
        CEF(0.5),
        CEF(0.5),
        CEF(0.5),
        CEF(0.5),
………….
}

Frac16 z11[NUM_SAMPLES];
```

```
const int FirCoefs[]=
-1953,-1435,1006,4646,7904,9207,7904,4646,1006,-1435,-1953
};

dfr16_tFirStruct* pFir;
Frac16 *pFirCoefs=(Frac16 *)&FirCoefs[0];
//Frac16 firCoefArray[FIR_COEF_LENGTH * sizeof(Frac16)];
//Frac16 history[FIR_COEF_LENGTH * sizeof(Frac16)]; /*unaligned */
//#ifndef BLOCK
extern void  dfr16FIRC (dfr16_tFirStruct *pFIR, Frac16 *pX, Frac16 *pZ, UInt16 n);
bool adc_flag=FALSE;
Frac16 sample0;
Frac16 filter_result0;
void main(void)
{
  /*** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!! ***/
  PE_low_level_init();
  /*** End of Processor Expert internal initialization.                    ***/

  /* Write your code here */
         pFir = dfr16FIRCreate (pFirCoefs, FIR_COEF_LENGTH);
         if(pFir==NULL) asm(debug);
         dfr16FIRC((dfr16_tFirStruct*)pFir,x,z11,NUM_SAMPLES);
         asm(nop);
         asm(nop);


  for(;;) {}
}
```

# 4 FFT

## 4.1 FFT Implementation

This section explains how to implement a fast Fourier transformation (FFT) under the Processor Expert platform in the CodeWarrior tool.

An FFT can transform signals from the time domain to the frequency domain. Assume that the ADC samples an external analog signal at a fixed frequency Fs, and gets *N* points samples x(n).

$$z(k) = scale \times \sum_{i=0}^{N-1} x[i]\left(\cos\left(2\pi \times \frac{k}{N}\right) - j\sin\left(2\pi i \times \frac{k}{N}\right)\right);\ 0 \le k \le \frac{N}{2} \qquad \text{Eqn. 44}$$

The value of z(k) is the frequency component of discrete samples x[i]. In the frequency domain, z(k) corresponds to the actual frequency Fs × k/N and the frequency resolution is Fs/N. For example, suppose that Fs is 1000 Hz and the number of sample points N is 1024. Then the frequency resolution is Fs/N = 1000 Hz/1024 = 0.9765 Hz, and z(k) corresponds to the frequency 1000k/1024. For example, z(3) corresponds to the frequency 2.93 Hz. Because z(k) and z(N–k) are conjugate complex numbers, the FFT API function only gives z(k). The value of k ranges from 0 to N/2.

This is the structure which is used to save the FFT output result:

```
typedef struct {
Frac16 z0; /* z[0] (real 16-bit fractional) */
Frac16 zNDiv2; /* z[n/2] (real 16-bit fractional) */
CFrac16 cz[1]; /* z[1] .. z[n/2 - 1] (complex 16-bit fractional)*/
} dfr16_sInplaceCRFFT;
```

**NOTE**

The remaining (N/2 – 2) locations for the cz[] buffer must be allocated by the user.

According to Equation 44, when we compute the DC component, k is equal to zero. In this case, cos(2πi×0/N)=1 and sin(2πi×0/N)=0.

$$z[0] = scale \times \sum_{i=0}^{N-1} x[i]$$

Because the imaginary part of the complex z(0) is zero, z(0) is a real number.

Let’s compute the last item of z(k), z(N/2). In this case,

k = N/2

cos(2πi×(N/2)/N) = cos(πi) = –1 when i is an odd number

cos(2πi×(N/2)/N) = cos(πi) = +1 when i is an even number

sin(2πi×(N/2)/N) = 0 when i is any integer

so

$$z\left[\frac{N}{2}\right] = scale \times \sum_{i=0}^{N-1} x[i] \times \cos(\pi i)$$

Therefore z(0) and z(N/2) are real numbers instead of complex numbers, because their imaginary parts are zero.

To save memory, we declare z[0] as Frac16 z0 and z[N/2] as Frac16 zNDiv2. We declare other FFT results from z[1] to z[N/2–1] as complex CFrac16 cz[].

This is an example of computing FFT for 16 points:

```
#define NUMBER_SAMPLES 16
Frac16 pX_input[NUMBER_SAMPLES] = {
0x52ef,0x6d71,0x3a3e,0x4158,0x2a29,0x6bef,0x5133,0x54c0,\
0x783e,0x7bcb,0x1f49,0x22d5,0x4654,0x49e0,0x6d5e,0x70eb};

//CFrac16 Actual_op_16[NUMBER_SAMPLES]={20938, -1874,209, -230, 3136, 2138, -1084, 587, 569,
-1875,\
//-1358, 1461, -234, -1095, -156, -1159};
```

```
Frac16 pX[NUMBER_SAMPLES];
Frac16 pZZ[NUMBER_SAMPLES];

void main(void)
{
    Int16 res, flag = 1;  /* 1 = Pass, 0 = Fail */
        UInt16 n=NUMBER_SAMPLES, options = FFT_SCALE_RESULTS_BY_N;
        Int16 i, j;
//        const CFrac16 *Actual_op;
        dfr16_sInplaceCRFFT *pZ;
        dfr16_tRFFTStruct *pRFFT;

  /*** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!! ***/
  PE_low_level_init();
  /*** End of Processor Expert internal initialization.                    ***/

  /* Write your code here */

    pZ=(dfr16_sInplaceCRFFT*)pZZ;


                          for (i=0; i < n; i++)
                          {
                            pX[i] = pX_input[i];
                          }


                          /* Call FFT Create function */
                          pRFFT = DFR1_dfr16RFFTCreate(NUMBER_SAMPLES, options);
                          if (pRFFT == NULL)
                          {
                          printf("pRFFT is NULL,memory allocation fail\n");
                                                                    asm(debug);
                          }
                          else
                          {
                          printf("dynamic memory allocation success\n");
                                                  asm(nop);
                          }

                res = DFR1_dfr16RFFT(pRFFT, &pX[0], pZ);
                asm(nop);
                asm(nop);
}
```

After running the code, the result of the FFT pZ={20938, –1874, 209, –230, 3136, 2138, –1084, 587, 569, –1875, –1358, 1461, –234, –1095, –156, –1159}.

Because the DSP56800/E is a fixed processor, we suggest selecting the option variable as FFT_SCALE_RESULTS_BY_N. Thus, the FFT result will be divided by the length of input samples N; the FFT result will not be greater than 1; and it can be represented by the Frac16 data type.

To compare the FFT result computed by DSP with the result by the general FFT tools such as matlab, users must multiply the FFT result by N.

## 4.2 FFT Results

The DSP56800/E is a fixed point processor, so it can only recognize decimal fraction numbers. Users can define a macro as shown below to transfer between the decimal fraction format and the Frac16 data format. Table 1 is the Frac16 data type format saved in memory.

```
#define CFF(x) (Frac16)(x * 32767.0)
```

**Table 1. Frac16 Data Type Representation**

| Hexadecimal Representation | Integer | | Fraction | |
|---|---|---|---|---|
| | Binary | Decimal | Binary | Decimal |
| 0x7FFF | 0111 1111 1111 1111. | 32767 | 0.111 1111 1111 1111 | 0.9997 |
| 0x7000 | 0111 0000 0000 0000. | 28672 | 0.111 0000 0000 0000 | 0.875 |
| 0x4000 | 0100 0000 0000 0000. | 16384 | 0.100 0000 0000 0000 | 0.5 |
| 0x2000 | 0010 0000 0000 0000. | 8192 | 0.010 0000 0000 0000 | 0.25 |
| 0x1000 | 0001 0000 0000 0000. | 4096 | 0.001 0000 0000 0000 | 0.125 |
| 0x0000 | 0000 0000 0000 0000. | 0 | 0.000 0000 0000 0000 | 0.0 |
| 0xC000 | 1100 0000 0000 0000. | –16384 | 1.100 0000 0000 0000 | –0.5 |
| 0xE000 | 1110 0000 0000 0000. | –8192 | 1.110 0000 0000 0000 | –0.25 |
| 0xF000 | 1111 0000 0000 0000. | –4096 | 1.111 0000 0000 0000 | –0.125 |
| 0x9000 | 1001 0000 0000 0000. | –28672 | 1.001 0000 0000 0000 | –0.875 |
| 0x8000 | 1000 0000 0000 0000. | –32768 | 1.000 0000 0000 0000 | –1.0 |

For example, if Frac16 var = CEF(–0.25), then 0xE000 will be saved in memory for the variable var. In other words, var is expressed in two's complement representation.

For the input sample array in the time domain:

Frac16 pX_input[NUMBER_SAMPLES] = {0x52EF, 0x6D71, 0x3A3E, 0x4158, 0x2A29, 0x6BEF, 0x5133, 0x54C0, 0x783E, 0x7BCB, 0x1F49, 0x22D5, 0x4654, 0x49E0, 0x6D5E, 0x70EB}

the input array can be converted into decimal fraction format. For example, the first element is 0x52EF/32768= 0.648.

So the pX_input array can be expressed as:

pX_input[NUMBER_SAMPLES]= {0.648, 0.855, 0.455, 0.510, 0.329, 0.843, 0.634, 0.662, 0.939, 0.967, 0.244, 0.272, 0.549, 0.577, 0.854, 0.882}.

The DC component in the frequency domain will be the sum of all the elements in the input array, which is 10.223. If the sum is divided by the input sample number 16, we get the value of 0.6389 in decimal fraction format. If you represent the value in the Frac16 data type, the value is 20938 or 0x51CA.

For the FFT result array:

pZ={20938, –1874, 209, –230, 3136, 2138, –1084, 587, 569, –1875, –1358, 1461, –234, –1095, –156, –1159}

the fundamental component in the frequency domain is (209, –230), and it is (209/32768, –230/32768) = (0.00637,0.00702) as represented in the decimal fraction. Taking into consideration the scaling by N (here 16), the actual value is (0.1019,0.1123).

## 4.3 General Code to Compute DFT for Comparison

This is the code used to compute the DFT. The code can be compiled and run on a PC.

```
//
//this source code is aimed at result comparison between DSP and PC
//calling Processor Expert API function, the result will be


#define NM 16
#define PI 3.1416


typedef struct
{
         double real;
         double image;
}complex;
double test;
float input[NM] =
{0.648,0.855,0.455,0.510,0.329,0.843,0.634,0.662,0.939,0.967,0.244,0.272,0.549,0.577,0.854,0.
882};
complex output[NM];

int main()
{
         //do FFT
         int k,n;
         test=sin(PI/6);
         for(k=0; k<NM; k++)
         {
                  output[k].real=0;
                  output[k].image=0;
                  for(n=0; n<NM;n++)
                  {
                           output[k].real=input[n]*cos((float)2*PI*k*n/NM)+output[k].real;

output[k].image=-1.0*input[n]*sin((float)2*PI*k*n/NM)+output[k].image;
                  }
                  printf("real=%lf,image=%lf\n",output[k].real,output[k].image);
         }

         printf("finish!\n");
         return 0;
```

# 5 DAC

## 5.1 DAC of the MC56F802x Family

The MC56F8000 family has a 12-bit DAC module. Its conversion rate can reach up to 0.5 million samples per second or 500 kHz, and the output signal ranges from GND to $V_{DDA}$. The DAC can work in automatic mode or normal mode.

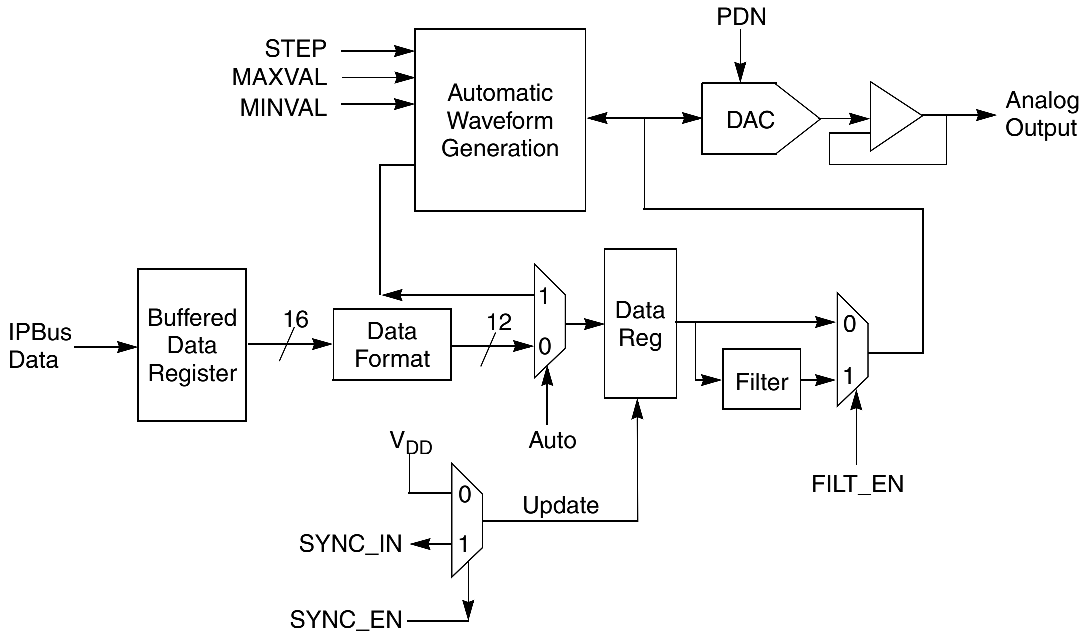



**Figure 10. DAC Diagram of MC56F803x or MC56F802x**

In automatic mode, for each external trigger, the DAC will add to or subtract a constant value saved in the STEP register automatically without DSP intervention. So the DAC can only output a linear signal, such as triangle, sawtooth, or square wave waveforms. If the user wants to generate a nonlinear signal such as a sine wave signal, the DAC must be set as in normal mode and the DAC output must be updated with DSP core intervention.

The DAC trigger signals can be from a timer, a PIT, or a PWM reload synchronization signal, depending on the device — refer to the system integration module (SIM) chapter of the data sheet or reference manual for each device. Figure 11 shows the trigger source register for the MC56F8036.

| Base+$19 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Read | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | IPS1_DSYNC1 | | | 0 | IPS1_DSYNC0 | | |
| Write | | | | | | | | | | | | | | | | |
| Reset | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Figure 11. DAC Trigger Source Register**

The field IPS1_DSYNC1 is used to select the alternate input source signal to the DAC1 SYNC input.

- 000 = PIT0 (internal) — Use programmable interval timer 0 output as DAC SYNC input (default)

- 001 = PIT1 (internal) — Use programmable interval timer 1 output as DAC SYNC input
- 010 = PIT2 (internal) — Use programmable interval timer 2 output as DAC SYNC input
- 011 = PWM SYNC (internal) — Use PWM reload synchronization signal as DAC SYNC input
- 100 = TA0 (internal) — Use timer A0 output as DAC SYNC input
- 101 = TA1 (internal) — Use timer A1 output as DAC SYNC input
- 11x = Reserved

The example below demonstrates how to use the on-chip DAC of the MC56F8037 to output a sine signal. In the interrupt service routine of the PIT, the DSP core updates the DAC data register.

The code outputs a sine table synchronized by the PIT.

```
/* Including used modules for compiling procedure */
#include "Cpu.h"
#include "PIT3.h"
#include "DAC2.h"
/* Include shared modules, which are used for whole project */
#include "PE_Types.h"
#include "PE_Error.h"
#include "PE_Const.h"
#include "IO_Map.h"
#define SINE_TABLE_LENGTH 20
Unsigned int sine_table[SINE_TABLE_LENGTH]={0x4001,0x53c6,0x659a,0x73bd,0x7ccc,0x7fe3,
0x7cb4,0x738f,0x655a,0x537a,0x3fb0,0x2bed,0x1a22,0xc0e,0x313,0x12,0x357,0xc90,0x1ad5,0x2cbf
};
unsigned int temp;
void main(void)
{
  /* Write your local variable definition here */

  /*** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!! ***/
  PE_low_level_init();
  /*** End of Processor Expert internal initialization.                    ***/
  setRegBit(DAC0_CTRL,FORMAT); //
  setReg(INTC_IPR5,0x3000);
  setRegBit(SIM_PCE1,PIT0); //enable PIT0 clock
  setReg(SIM_PCE0,0x1000);
  setReg(SIM_IPS1,0x0000);
         __EI0();
         //set
  /* Write your code here */

  for(;;) {}
}
Interrupt service routine of PIT:
/* END PIT_8037_PE */
/*
** ###################################################################
**
**     This file was created by UNIS Processor Expert 2.98 [03.79]
**     for the Freescale MC56800 series of microcontrollers.
**
** ###################################################################
*/
#pragma interrupt saveall
```

```
void PIT_ISR()
{
         static index=0;
         temp=sine_table[index];
         index++;
         if(index>(SINE_TABLE_LENGTH-1)) index=0;
         asm(nop);
         asm(nop);
         //writting a sine table to DAC
         setReg(DAC0_DATA,temp);
         getReg(PIT0_CTRL);
         clrRegBit(PIT0_CTRL,PRF);
}
```

## 5.2 Class D Amplifier

It is possible for the PWM module to function as a DAC, so the PWM is widely used in many applications such as motor control, switch mode power supply, and audio power stage output.

In the DSP56F800 and MC56F8300 families, most chips have an embedded PWM module.

To save power, the power amplifier must operate completely in switching mode — class D. Normally, the amplifier has efficiency higher than 90 percent, and over 90 percent of the power is delivered to the load. In fact, a Class D amplifier works as a power DAC and can be used to realize high output power. The Class D amplifier is based on an analog technique for pulse width modulation (PWM).

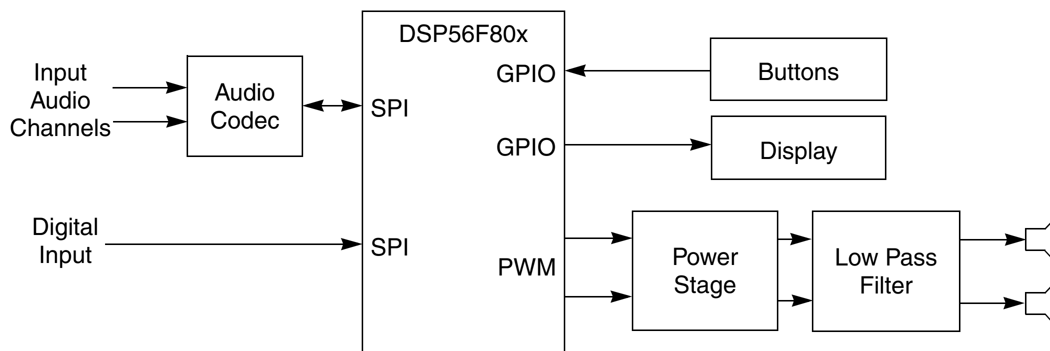



**Figure 12. Class D Amplifier Block Diagram**

In Figure 12, the power stage consists of a transistor, MOSFET or IGBT, which works in cutoff or saturation mode. The power devices consume less power in these modes than in amplifier mode, so the power efficiency increases. The low-pass filter is a passive filter and cuts frequency components that exceed the maximum frequency of the input signal.

When we use the PWM as a DAC, the two equivalent specifications — resolution and sample cycle — are implicit. Here, we discuss how to compute the two specifications, resolution and sample cycle. Figure 13 shows the signals of the PWM module in the DSP56800/E family.

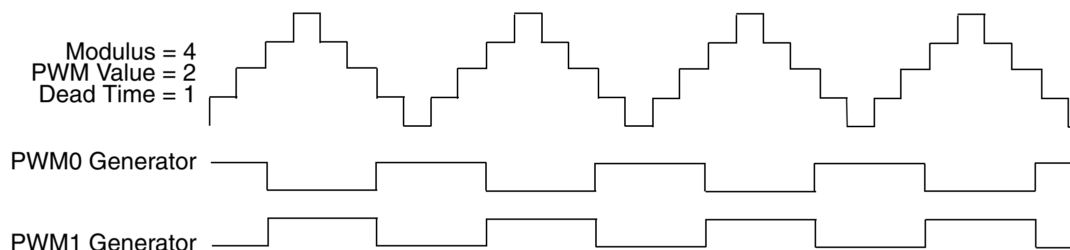



**Figure 13. PWM Waveform**

The PWM module is driven by the IP_BUS clock.

In Figure 13, the PWM cycle can be viewed as a DAC sample cycle. When the PWM is treated as a DAC, the PWM duty cycle denotes the power of the PWM, which ranges from zero to the value saved in the PWM modulo register.

The PWM output frequency is (IP_BUS clock frequency) / (2 × PWM modulo register) in center-aligned mode.

Regarding the PWM resolution, we can express it in equivalent bits to the DAC resolution. The PWM resolution can be expressed as resolution bit = $\log_2$(PWM Counter Modulo Register Value).

The PWM output frequency and resolution are not independent. The relationship is:

$2^{\text{Resolution Bits}}$ × (PWM output frequency) = IP_BUS clock frequency / 2 in center-alignment mode.

# 6 Conclusion

This application note introduces the features of the DSP56800/E devices and their on-chip ADC, lists the ADC specification, and provides the circuit of the voltage reference for the on-chip ADC. It also demonstrates how to design FIR and IIR filters using the QEDesign Lite software, and how to implement those filters using the function which Processor Expert provides after the user enters the coefficients. It also gives examples for implementing the FIR/IIR filter and the FFT. Finally, it illustrates how to compute the PWM frequency and resolution, and gives an example for the DAC of the MC56F803x family.

# Appendix A IIR Filter Code

```
Result  dfr16IIRC_Fixed(dfr16_tIirStruct *pIIR, Frac16 *pX, Frac16 *pZ, UInt16 n)
{
      UInt16   i, j, nbiq;
      Frac16 * pCoefs;
      Frac16   w0,w1,w2;
      Frac16      xn;
      Frac32   Acc,long_xn;

      if (n > PORT_MAX_VECTOR_LEN)
      {
          return FAIL;
      }
      for(i = 0; i < n; i++)                                          /* Sample loop */
      {
                    /* initialize total and setup pointer to walk through coefficients */
          pCoefs = pIIR -> pC;
          Acc = 0;

          xn = shr(*pX++,1);                                       /* x'[n] = x[n]/2 */

          pIIR -> pNextHistoryAddr = pIIR ->pHistory;

          for (j = 0; j < pIIR->nbiq; j++)                            /* States Loop */
          {


              w1 = *(pIIR -> pNextHistoryAddr++);
              w2 = *(pIIR -> pNextHistoryAddr--);

              Acc = L_mult(*pCoefs++,w1);                                   /* a1*w1 */
              Acc = L_mac(Acc,*pCoefs++,w2);                                /* a2*w2 */

              long_xn = L_deposit_h(xn);
              Acc = L_add(long_xn,Acc);              /* Acc = x[n] + a1*w[n-1] + a2*w[n-2] */
              Acc = L_shl(Acc,1);                                  /* w'[n] = 2*Acc */
              w0 = round(Acc);

              Acc = L_mult(*pCoefs++,w0);                                 /* b0 * w0 */
              Acc = L_mac(Acc,*pCoefs++,w1);                              /* b1 * w1 */
              Acc = L_mac(Acc,*pCoefs++,w2);                              /* b2 * w2 */

              *(pIIR -> pNextHistoryAddr++) = w0;
              *(pIIR -> pNextHistoryAddr++) = w1;

              xn = round(Acc); /* = y'[n] (=y[n]/2), becomes x'[n] for next biquad */


          }


          *pZ++ = L_shl(xn,1);                                     /* y[n] = 2*y'[n] */
      }
```

```
        pIIR->pNextHistoryAddr = pIIR -> pHistory;

        return PASS;
}
```

# Appendix B References

- *56F8300 Peripheral User Manual* — MC56F8300UM
- *56F80xx/56F81xx Technical Data Sheet* — specific to the device you're implementing
  (The order number for the document is exactly the same as the device number, for example, MC56F8037.)
- *MC56F800E Reference Manual* — MC56F800ERM
- QEDesign Lite online help
- Processor Expert online help

**How to Reach Us:**

**Home Page:**
www.freescale.com

**Web Support:**
http://www.freescale.com/support

**USA/Europe or Locations Not Listed:**
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

**Europe, Middle East, and Africa:**
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

**Japan:**
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

**Asia/Pacific:**
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3599
Rev.0
07/2008

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals”, must be validated for each customer application by customer’s technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

RoHS-compliant and/or Pb-free versions of Freescale products have the functionality and electrical characteristics as their non-RoHS-compliant and/or non-Pb-free counterparts. For further information, see http://www.freescale.com or contact your Freescale sales representative.

For information on Freescale’s Environmental Products program, go to http://www.freescale.com/epp.

Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.
© Freescale Semiconductor, Inc. 2008. All rights reserved.

freescale™
semiconductor