Iowa Hills Software, LLC
www.IowaHills.com
May 1, 2016

The Filter Code Kit consists of the following.

FFTCode.cpp	The FFT and other freq analysis algorithms.
FilterKitMain.cpp	Contains sample calls for the various functions in the kit.
FIRFilterCode.cpp	The FIR filter code.
FreqSamplingCode.cpp	FIR filters by frequency sampling.
IIRFilterCode.cpp	The IIR filter code.
LowPassPrototypes.cpp	Low pass prototype filter coefficients.
LowPassRoots.cpp	Calculates the roots for Butterworth, Cheby, etc.
NewParksMcClellan.cpp	The Parks McClellan FIR algorithm for FIR filters.
PFiftyOneRevC.cpp	A 100th order root finder.
QuadRootsCode.cpp	A Quartic root solver, called by P51.
CplxDMath.hpp	A complex math library (see the notes below).

With a bit of luck, you should be able to start a project, include these files in the project, compile, and run. The code uses the C++ Builder ShowMessage function to display warnings and errors in about 30 places. Since this isn't a standard C function, they have been commented out. Be sure to replace this function with something appropriate. If you are using C++ Builder, include vcl.h for ShowMessage.

If you get compilation errors because of undefined constants, such as M_PI and M_SQRT2 (Pi and sqrt(2), we included these definitions as comments in the FilterKitMain.h file, but we believe most compilers include these in the math.h file.

Visual C
If you are using Visual C, we have been told that this statement is needed in some of the files for the math constants.
```
#define _USE_MATH_DEFINES
```

Also, the FIRFilterCode file contains a Sinc function that may need to be removed to avoid confusion with VC's Sinc function.

Complex Math
Before the C11 standard was released, there wasn't a standard method for handling complex numbers in C. As a result, all the compilers handled them differently and used a wide range of syntax for complex numbers.

To avoid this problem, previous versions of this kit didn't use complex math, but the inclusion of the low pass filter prototype code necessitated the use of complex variables and inclusion of our complex math library.

The library CplxDMath.hpp, was written by Iowa Hills Software, who are electrical engineers, not software professionals. CplxDMath should be good enough to get you started, but it should be replaced with your compiler's complex math library, which has been more properly vetted (at least in theory).

Attributions
Most of the algorithms used here are essentially textbook algorithms. For example, the equations for the rectangular window FIR filters are developed in many textbooks. The code used here however, is our own work, except as noted.

The Parks McClellan algorithm is obviously not our work, but we did this particular C translation.

The quartic roots code is based on the code by Terence R.F.Nonweiler that was downloaded from http://www.netlib.org/toms Algorithm 326.

The P51 root finder is loosely based on the Jenkins Traub algorithm found at http://www.netlib.org/toms/ Algorithm 493.

The elliptic filter roots code was described in "Elliptic Functions for Filter Design" by H J Orchard and Alan N Willson. IEE Trans on Circuits and Systems April 97.

We derived the IIR Bilinear Transform algorithm used here. Most authors describe a more complicated method that requires finding the roots of a frequency scaled Nth order polynomial. Our method relies on normalized second order s plane filter polynomials. One of the advantages of this method, is that a single set of filter coefficients can be used to generate any type of filter desired. For example, the normalized polynomial coefficients for a 4th order Butterworth can generate a 4th order Butterworth low pass, high pass, band pass, or notch filter, at any center frequency or bandwidth. See the IIREquationDerivations.pdf for details.

This code is guaranteed to have errors, somewhere. If you find a problem, or would like to make a suggestion, please leave us a note at:
http://www.iowahills.com/feedbackcomments.html

Sampling Frequency
We get many questions regarding how the variables OmegaC and BW are related to the sampling frequency.

In the discrete time domain, there is no frequency greater than Pi, so all frequencies are given as a percentage of Pi. Or said a bit differently, all the frequency values are given as a percentage of Nyquist. This is consistent with the terminology used in most discrete time textbooks and is consistent with the frequency terminology used in a DFT or FFT.

For example, if the sampling freq = 20 kHz, then Nyquist = 10 kHz. For a band pass filter with a center frequency of 2 kHz and a 3 dB bandwidth of 500 Hz, set OmegaC = 0.2 and set BW = 0.05, which means 0.2 Pi and 0.05 Pi.

Clearly, we could have incorporated a sampling frequency into the code, but this is pure filter design code. In our view, the sampling frequency belongs at a higher level, in the user interface code. This filter design code doesn't use the sampling frequency.

Denormal Numbers
Just in case you have never heard of denormal numbers, I would like to suggest that you take some time to learn about them. When working with this type of code, we occasionally have problems with extremely slow execution times. For example, we might have trouble with the FFT requiring 100 X more time to execute than normal. The problem is that very small math values can slow execution times to a crawl.

In brief, when the processor executes a math instruction where the variables, and or the result, are less than DBL_MIN, it starts treating them as denormal, per the IEEE standard, and this takes a significant amount of time. You can read about it on Wikipedia or the Stack Overflow site.

This is not a problem unique to this code. It is just that denormal math affects these types of algorithms a great deal. i.e. Algorithms that do millions of math operations, and you need to be aware of it. Per MSDN and Wikipedia, the processors math control registers can be modified to stop denormal math. This is supposed to allow the processor to flush small values to zero, as they say, but I haven't been able to make their instructions work.

The simple solution to this problem is to not allow math values to approach DBL_MIN (2.23E-308), which usually means not using zero as an input value to an IIR filter or FFT. Use something like 1.0E-100 instead.

Low Pass Prototype Filters

Previous versions of this kit contained a large table of filter coefficients for the various filter polynomials, such as the Butterworth and Elliptic.

This kit contains the code to calculate the pole zero locations for the various filter polynomials and generates the filter's second order polynomials, which is what we typically need for filter design. These prototype filters are frequency scaled to ensure their 3 dB corner frequency is at 1 rad/sec.

The code also orders the poles and zeros properly for use in opamp and IIR filters. In other words, from a mathematical point of view, the poles and zeros can be matched and ordered in any manner, but for filter design, we need the poles and zeros mated and ordered in a manner that generates the smallest peak voltages for opamp filters and the smallest math values for IIR filters employing fixed point math.