Freescale Semiconductor
Application Note

Document Number: AN4464
Rev. 0, 10/2012

Digital Filtering with MMA955xL

by: Maureen Helm

This application note introduces digital filtering on the Freescale Xtrinsic intelligent sensing platforms. It is divided into two major sections: the first discusses basic digital filtering theory and general practices, while the second applies the theory and discusses specific filtering enablement features available on the MMA955xL platform.

This document is intended for customers who need to take advantage of the MMA955xL platform's digital filtering enablement features but have little to no experience in digital signal processing.

Contents

1 Digital filter primer 1
1.1 Introduction 1
1.2 Sampling theory and aliasing 2
1.3 Transfer function and difference equation 2
1.4 Digital frequency 3
1.5 Filter realization 3
1.6 Fixed-point issues 4
1.7 Hardware acceleration 5
2 MMA955xL Filter Usage 6
2.1 Introduction 6
2.2 Nth order IIR filter 6
2.3 Configurable cutoff IIR filter 6
2.4 Properties 8
2.5 Use tips 8
2.6 Examples 8
2.7 Frontend anti-aliasing filter 10
3 Goertzel Frequency Analysis 11
3.1 Introduction 11
3.2 Algorithm 12
3.3 Example 12
4 Appendix 12
4.1 Nth order IIR filter 13
4.2 Configurable cutoff IIR filter 16
4.3 Goertzel algorithm 16
5 Other Resources 18
5.1 CFDSPLIB: *Complimentary ColdFire Digital Signal Processing Library* 18
6 Definitions and Acronyms 19
7 Related documentation 20
8 Revision History 20

1 Digital filter primer

1.1 Introduction

Digital filters are often used to process sampled continuous-time signals in discrete-time. Unlike analog filters that typically consist of capacitors and resistors, digital filters comprise of registers and arithmetic operators that can be implemented in custom digital logic or software. MMA9550L, for example, is an intelligent

© 2012 Freescale Semiconductor, Inc. All rights reserved.

freescale
Xtrinsic

sensing platform that samples linear acceleration, an inherently analog continuous-time signal, into a discrete-time sequence of quantized digital samples. Once sampled, these digital signals can be processed with software on an embedded ColdFire microcontroller. This section provides a basic introduction to digital filter theory to enable MMA955xL users to understand the capabilities and limitations of the available filtering tools provided by Freescale, and to use these tools effectively.

1.2 Sampling theory and aliasing

When sampling a signal with an ADC, two important but distinct operations occur: a continuous-time signal is converted to a discrete-time sequence, and an analog signal is quantized into a digital signal. Sampling theory addresses the first of these operations. The rate at which an ADC samples in time is called the sample rate or sample frequency, and half that value defines the Nyquist frequency. The Nyquist frequency is a fundamental concept in sampling theory because it indicates the point at which frequencies begin to alias. Frequencies above Nyquist cannot be distinguished from those below and therefore alias to lower frequencies after sampling. For example, a 60 Hz signal sampled at 100 Hz will alias to 40 Hz. Once aliasing occurs it cannot be undone, therefore it is important to bandlimit a signal in the analog domain before sampling. The Analog Front End (AFE) in the MMA955x devices includes a low pass filter with a bandwidth that adjusts to approximately one quarter the selected sample rate to perform this bandlimiting. This rule also holds true for any noise above the Nyquist frequency or when downsampling a discrete signal to a lower rate.

1.3 Transfer function and difference equation

A digital filter is a linear time-invariant system that can be characterized by a discrete transfer function and a constant coefficient difference equation. The transfer function describes the input-output relationship of the system in the Z-domain and the difference equation provides a discrete time-domain method to compute the output of the system given a particular input. A simple relationship exists between the transfer function and the difference equation in that both use the same set of constant coefficients. A transfer function can easily be converted into a difference equation by replacing $b_k z^{-k}$ with $b_k x[n-k]$ and $a_k z^{-k}$ with $a_k y[n-k]$. Assuming the transfer function's region of convergence includes the unit circle, it can also be converted into a Discrete Fourier Transform (DFT) by replacing z with $e^{j\omega}$.

Transfer function

$$H(z) = \frac{\sum_{k=0}^{N} b_k z^{-k}}{1 + \sum_{k=0}^{N} a_k z^{-k}} \qquad \textbf{Eqn. 1}$$

Difference equation

$$y[n] = \sum_{k=0}^{N} b_k x[n-k] - \sum_{k=1}^{N} a_k y[n-k] \qquad \textbf{Eqn. 2}$$

The difference equation and transfer function can be broken down into several individual terms:

- b_k are constant *numerator coefficients* because they contribute to the numerator of the transfer function.
- $x[n\text{-}k]$ are the current and previous input samples to the filter, whereas k indicates the number of time delays.
- a_k are constant *denominator coefficients* because they contribute to the denominator of the transfer function.
- $y[n\text{-}k]$ are the current and previous output samples of the filter, whereas k indicates the number of time delays.

Put together, the difference equation consists of a sum of products:

- $b_k x[n\text{-}k]$ are the feed-forward terms, each of which is the product of a constant coefficient and a current or previous input to the filter.
- $a_k y[n\text{-}k]$ are the feedback terms, each of which is the product of a constant coefficient and a previous output of the filter. Notice that $a_0=1$, which is the coefficient for the current output. Filters that contain feedback terms are recursive and have an infinite impulse response (IIR), while filters that lack feedback terms have a finite impulse response (FIR).
- N is the order of the filter.

1.4 Digital frequency

Because digital filters operate in the discrete-time domain, frequency response is analyzed using digital, or angular, frequency. When used in a sampled system such as a sensor with an ADC, digital frequency may also be called normalized frequency since it can be normalized to the Nyquist frequency. Several unit conventions exist, but the most common are $[0,1)\ \pi$ *radians/sample* for normalized frequency, and $[0,\pi)$ *radians* for angular frequency. This application note follows the normalized frequency convention also used in Matlab.

1.5 Filter realization

Digital filters may be realized in a variety of ways, meaning that the difference equation can be computed differently. The most straightforward implementation of an IIR filter is called Direct Form I, in which the difference equation is evaluated exactly as written without any intermediate transformations. For an Nth order IIR filter, a Direct Form I realization requires $2N$ memory elements (one for each z^{-1} term), $2N+1$ multiplies, and $2N+1$ additions.

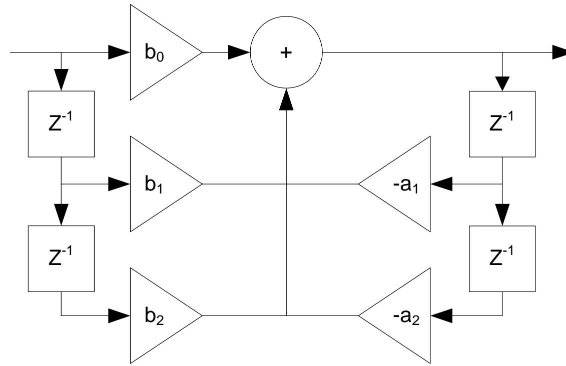

Figure 1. Second Order IIR Direct Form I Realization

Another common implementation of an IIR filter is called the Direct Form II, in which the difference equation is rearranged to minimize the number of memory elements. For an Nth order IIR filter, a Direct Form II realization requires N memory elements, $2N+1$ multiplies, and $2N+1$ additions. Although the order of arithmetic operations is different, both realizations use the same constant coefficients as the difference equation. By rearranging arithmetic operations, however, different filter realizations suffer from different fixed-point effects. Many other realizations exist in addition to the two direct forms discussed here.

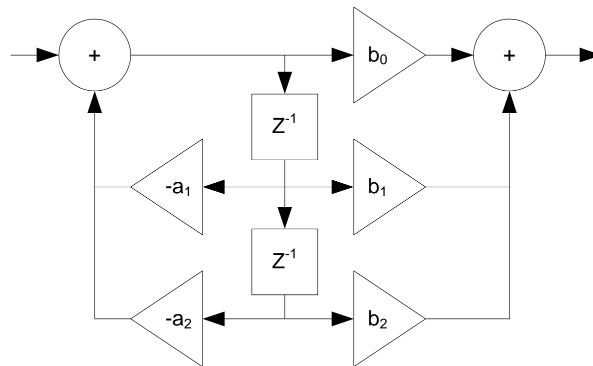

Figure 2. Second Order IIR Direct Form II Realization

1.6 Fixed-point issues

Converting an ideal digital filter with infinite precision coefficients and arithmetic into a realizable filter with fixed-point coefficients and arithmetic can be a complex trial-and-error process. Typically, coefficients are quantized into a fixed-point representation and then simulated with the desired filter realization to analyze the resulting frequency response. Ideally, the fixed-point frequency response should match the infinite-precision frequency response, but coefficient and arithmetic roundoff errors often cause the responses to differ significantly, sometimes even causing the filter to become unstable. There are several rules of thumb to consider:

- Lower order filters tend to perform better in fixed-point than higher order filters.
- High order filters can be broken up into multiple smaller order filters. Cascaded second-order sections, or *biquads*, are often used.
- Extreme frequency cutoffs (<0.1 or $>0.9\ \pi$ radians/sample) often do not work in 16-bit fixed-point representation. One method to achieve an extremely small frequency cutoff is to downsample first

and then apply a filter with a less extreme cutoff. Don't forget that aliasing concerns require low pass filtering of the data before downsampling.

- Saturate and round when typecasting down from wider formats (i.e., 32-bit down to 16-bit) to avoid overflow errors and truncation offsets.
- Limit the range of coefficients. Large ranges (i.e., one very small coefficient and one very large coefficient) can suffer from significant quantization error.
- Store intermediate results in a wider data format than the coefficients and input/output data. For example, keep all 32-bits of product when multiplying 16-bit coefficients with 16-bit data. Avoid casting back down to the input/output format until the entire sum of products has been calculated.

1.7 Hardware acceleration

In general, all realizations of digital filter difference equations evaluate sums of products and therefore benefit significantly from multiply-accumulate (MAC) acceleration. The ColdFire ISA, for example, has specialized supplemental instructions and registers that reduce each of the following C snippets into a single low-latency instruction:

```
acc += bk*xk; bk = *bk_ptr++; /* multiply accumulate with load */
acc -= ak*yk;                 /* multiply subtract */
```

Each of these instructions may be used with 16- or 32-bit signed or unsigned operands to update a 32-bit running sum. The hardware logic consists of a 3-stage execution pipeline optimized for 16-bit operands that can issue up to one instruction per cycle, as well as an added architectural accumulator register to store the running sum. Bare multiply instructions also benefit with lower execution latencies since they can utilize the accelerated multiplier hardware even if they do not need to update the accumulator.

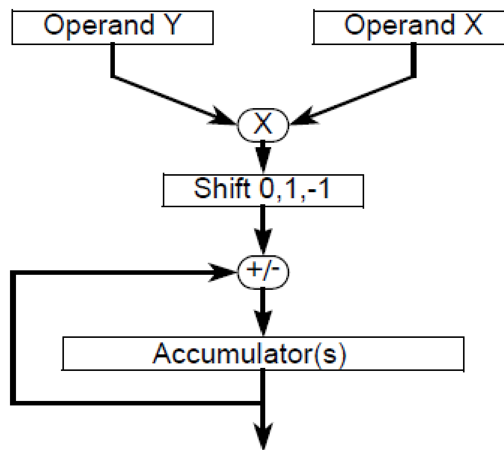

Figure 3. Multiply accumulate functional diagram

2 MMA955xL Filter Usage

2.1 Introduction

The standard firmware loaded onto all MMA955xL variants contains a configurable IIR filter function used to execute frontend digital anti-aliasing filters and all application-specific filters. It also includes a

configurable cutoff IIR filter function that does not require a user to design custom filter coefficients. Both functions are made available through the Freescale API to allow customer access.

2.2 Nth order IIR filter

The callable filter function implements an IIR Direct Form I realization through efficient use of the ColdFire MAC. It accepts a variable length array of coefficients as an input argument to enable any filter of any order. In addition, the function requires a pointer argument to store previous inputs and outputs. Remember that an Nth order IIR filter in Direct Form I uses N previous inputs and N previous outputs to compute the current output, therefore the lengths of the buffer and coefficient arrays vary proportionally with the order of the filter. Since the CodeWarrior IDE C compiler does not use MAC instructions, the filter function is implemented in assembly for optimal code size and execution speed but maintains a C-callable interface. It uses a signed 16-bit fixed-point representation for coefficients and input/output data with 32-bit intermediate accumulator results. Numerator and denominator coefficients share a common but configurable fixed-point scale factor.

2.3 Configurable cutoff IIR filter

A second callable filter function implements a wrapper around the Nth order IIR filter to create a simpler interface that does not require a user to design his/her own filter coefficients. This function computes a temporary set of first order high-pass or low-pass filter coefficients with a configurable frequency cutoff and then calls the generic filter function.

Configurable lowpass filter transfer function

$$H_{LPF}(z) = \frac{2^{-k}}{1 + (2^{-k} - 1)z^{-1}} \qquad \textit{Eqn. 3}$$

Configurable highpass filter transfer function

$$H_{LPF}(z) = \frac{1 - z^{-1}}{1 + (2^{-k} - 1)z^{-1}} \qquad \textit{Eqn. 4}$$

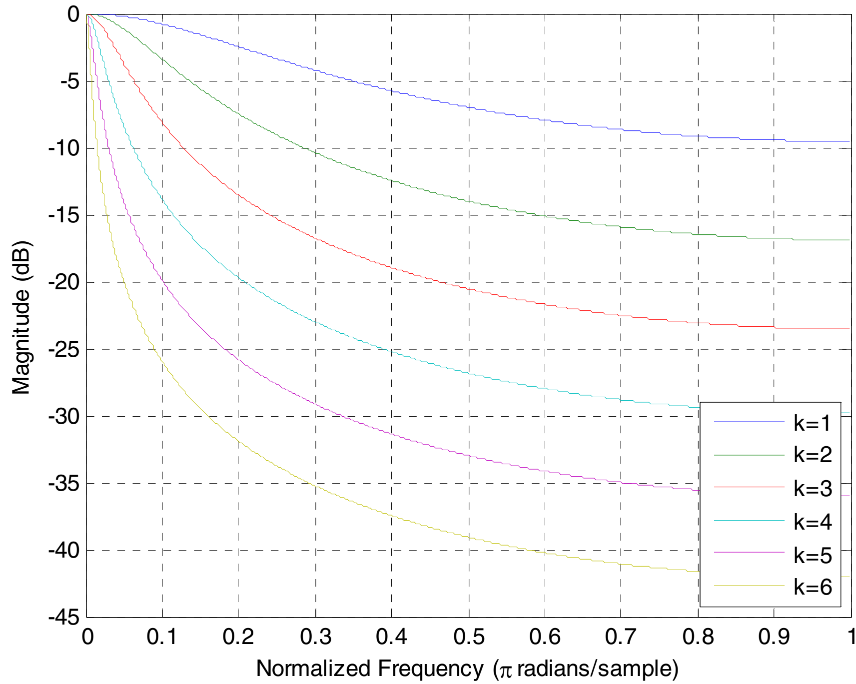

Figure 4. Configurable low-pass filter frequency response

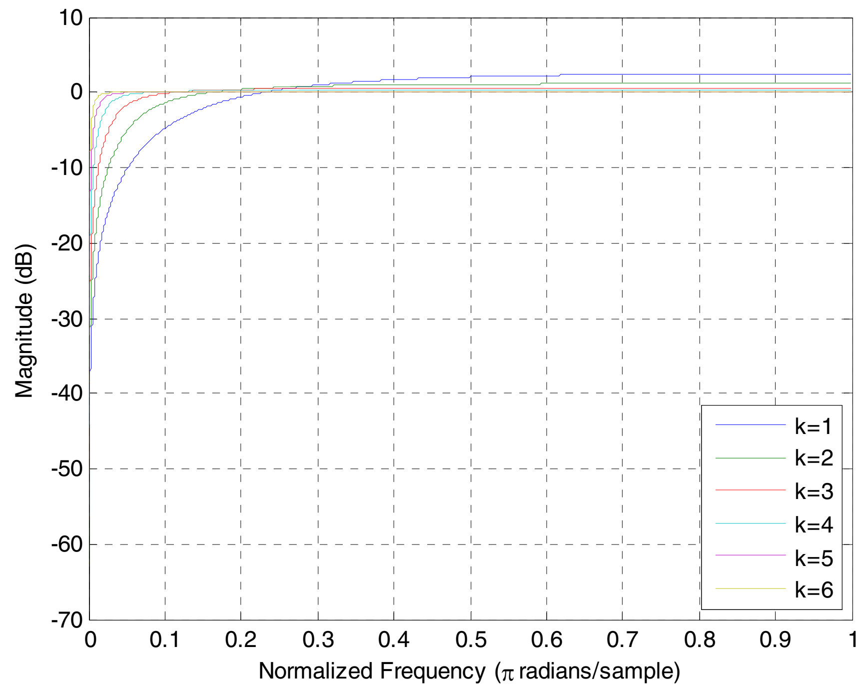

Figure 5. Configurable high-pass filter frequency response

2.4 Properties

Table 1 provides the filter properties for the Nth order IIR filter and the configurable cutoff IIR filter.

Table 1. MMA955xL filter properties

	Nth Order IIR Filter	Configurable Cutoff IIR Filter
Execution Time (core cycles)	65+10N	144
Code Size (Bytes)	112	82
Buffer Size (Bytes)	4N	4
Coefficients Structure Size (Bytes)	4N+8	12 (stored on stack)
Stack Usage (Bytes)	20	48

2.5 Use tips

Coefficients can be shared among multiple instantiations of the filter function because they are constant inputs. For example, the same set of coefficients can be used to apply a common filter to X, Y, and Z acceleration.

Coefficients can be declared as constants to direct the linker to place them in flash memory. If they are not declared as constants, then two copies of the coefficients will exist, one in flash and one in RAM.

Each instantiation of the filter function requires its own static RAM data buffer. Even if the same coefficients are used for X, Y, and Z acceleration, each axis requires its own data buffer to store previous inputs and outputs.

Align coefficients and buffer arrays to longword (4-byte) boundaries to minimize filter execution time. Although these arrays contain word (2-byte) elements, the filter function accesses them by longword.

2.6 Examples

This section provides examples of the analytical filter equation and susequent frequency response graph.

2.6.1 Analytical filter equations

Table 2 provides the equations for the analytical filter.

Table 2. Analytical filter equations

	Derivative	Leaky Integrator	4-Point Average
Difference Equation	$y[n] = x[n] - x[n-1]$	$y[n] = cy[n-1] + x[n]$ $0 < c < 1$	$y[n] = \frac{1}{4}\sum_{k=0}^{3} x[n-k]$
Transfer Function	$H(z) = 1 - z^{-1}$	$H(z) = \frac{1}{1 - cz^{-1}}$	$H(z) = \frac{1}{4}\sum_{k=0}^{3} z^{-k}$
Impulse Response Type	Finite (FIR)	Infinite (IIR)	Finite (FIR)
Order	1	1	3
Numerator Coefficients	[1, -1]	[1, 0]	$\left[\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}\right]$
Denominator Coefficients	[1, 0]	[1, -*c*]	[1, 0, 0, 0]
Discrete Fourier Transform	$H(e^{j\omega}) = 1 - e^{-j\omega}$	$H(e^{j\omega}) = \frac{1}{1 - e^{-j\omega}}$	$H(e^{j\omega}) = \frac{1}{4}\sum_{k=0}^{3} e^{-j\omega k}$
Frequency Response Magnitude	$\Vert H(e^{j\omega})\Vert^2 = 2 - 2\cos\omega$	$\Vert H(e^{j\omega})\Vert^2 = \frac{1}{1 + c^2 - 2c\cos\omega}$	$\Vert H(e^{j\omega})\Vert^2 = \frac{1}{16}(4 + 6\cos\omega + 4\cos 2\omega + 2\cos 3\omega)$
Frequency Response Type	High-pass	Low-pass	Low-pass

Figure 6 illustrates the analytical filter frequency response.

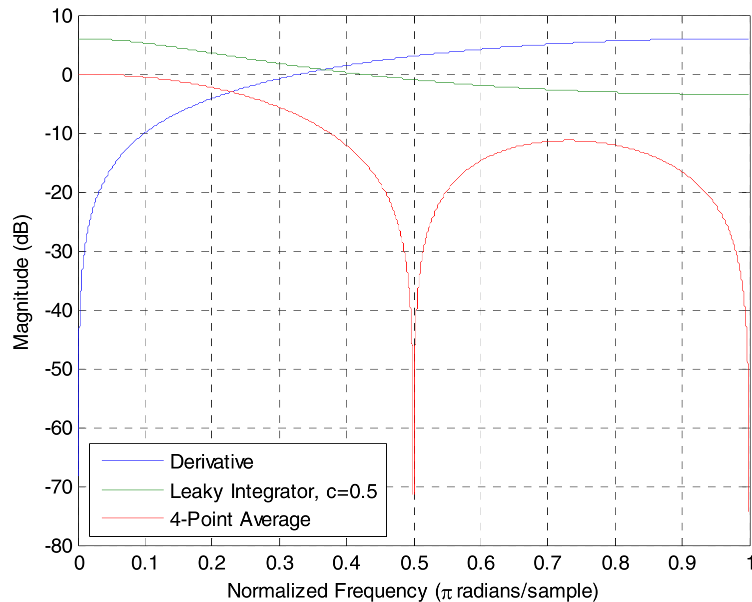

Figure 6. Analytical filter frequency response

2.7 Frontend anti-aliasing filter

The frontend module within the standard MMA955xL firmware reads, trims, and filters raw ADC results before making them available externally or to other internal firmware modules. This digital filter is called the *frontend anti-aliasing filter* because it significantly attenuates high frequencies such that downsampling can occur without aliasing. This is useful for applications such as portrait-landscape and tilt angle detection that execute at lower rates than the native ADC sample rate. The filter uses a sixth order Chebyshev Type II design featuring a monotonic passband and equiripple stopband, and the high order enables a fast rolloff.

The following Matlab commands generate floating-point coefficients, quantize the coefficients into fixed-point, and construct a data structure string that can be copied into C code and used with the Nth Order IIR filter function. The first command is built into the Matlab Signal Processing Toolbox and the second calls a custom function that can be found in Sample coefficients data structures.

```
[b,a] = cheby2(6,60,.6); % generate floating-point coefficients
[haa,saa] = nth_order_iir_filter(b,a); % create fixed-point model
```

Compare the frequency response of the Chebyshev design to a simple 4-point average filter: the Chebyshev is flatter in the passband and attenuates more aggressively in the stopband.

Figure 7 illustrates the frontend anti-aliasing filter frequency response.

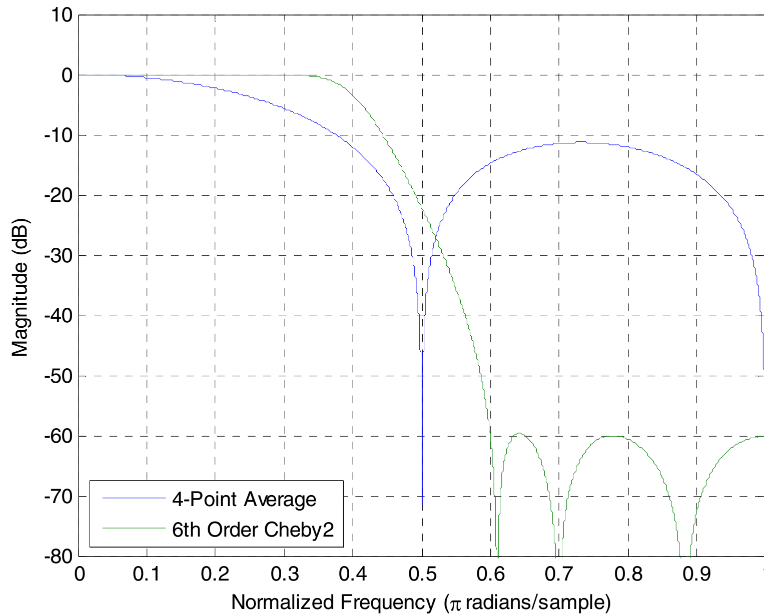

Figure 7. Frontend anti-aliasing filter frequency response

Because the MMA955xL firmware supports multiple simultaneous application rates (488, 244, 122 Hz), two instances of the same anti-aliasing filter occur in the frontend module for each accelerometer axis. First, ADC data sampled at 488 Hz is filtered to limit the bandwidth to 100 Hz. Then, the output of the filter is downsampled by two to achieve a 244 Hz sample rate. Finally, the downsampled data is filtered a second time with the same set of coefficients to limit the bandwidth to 50 Hz. This demonstrates the application of several important ideas introduced in previous sections:

1. The frequency response of a digital filter scales with sample rate.

$$f_c = \frac{100}{488/2} = \frac{50}{244/2} = 0.4\pi \text{ radians/sample} \qquad \textbf{Eqn. 5}$$

2. One copy of the constant coefficients array can be stored exclusively in flash and reused for multiple instances of a filter. This example uses the same coefficients for $(2\ \textit{sample rates}) \times (3\ \textit{axes}) = 6$ filter instances.

3 Goertzel Frequency Analysis

3.1 Introduction

In addition to filtering, DSP applications often also require frequency domain analysis, typically achieved via a Discrete Fourier Transform (DFT) or Fast Fourier Transform (FFT). A third method, called the Goertzel algorithm, can also be used. The FFT evaluates all frequency components at once, and requires a significant amount of memory and computational cycles. The DFT and Goertzel algorithm, on the other hand, can be used to selectively evaluate only the frequency components of interest in a point-by-point fashion, saving both memory and cycles. Their magnitude outputs are mathematically equivalent to one

another, but the Goertzel algorithm requires less memory than the DFT and can make better use of the ColdFire MAC.

3.2 Algorithm

The Goertzel algorithm consists of two parts: an inner loop that computes one output for every input sample, and an outer loop that computes one output for every N input samples. The parameters k and N select the frequency component, and their ratio equals the digital frequency (normalized by the sampling frequency). N corresponds to the length of the DFT, while k corresponds to the bin number and ranges from zero to N/2-1. Unlike the FFT, N does not need to be a power of two.

The inner loop is a second order IIR filter, but several coefficients are zero so it can be optimized from a generic implementation. The filter buffer size can also be reduced since x[n-1] and x[n-2] need not be saved.

$$y[n] = x[n] + 2cos\left(\frac{2\pi k}{N}\right)y[n-1] - y[n-2] \quad \textbf{Eqn. 6}$$

The outer loop computes the magnitude of the DFT bin after the Nth output of the inner loop:

$$\left\|X\left(\frac{2\pi k}{N}\right)\right\|^2 = (y[N-2])^2 + (y[N-1])^2 - 2cos\left(\frac{2\pi k}{N}\right)y[N-2]y[N-1] \quad \textbf{Eqn. 7}$$

3.3 Example

Consider an engine vibration application that samples an accelerometer at $f_s=2\ kHz$ and seeks to measure the magnitude of the fundamental frequency $f_0=244\ Hz$ over 512 samples.

Set N=512 and, $k = \frac{Nf_0}{f_s} = 62$, then compute the value $coef = 2cos\left(\frac{2\pi k}{N}\right)$ and scale to fixed-point.

```
goerzel_struct_t goertzel_struct;
N       = 512;
coef    = 23603; // 2cos(2*pi*k/N) * 2^sf
sf      = 14;
```

For each input sample, invoke the function:

```
output = goertzel(N,coef,sf,input,&goertzel_struct);
```

4 Appendix

This appendix provides the example codes for both the Nth order IIR filter and the configuragble cutoff IIR filter. The example code is provided both in the C source code and the Matlab model.

4.1 Nth order IIR filter

Example 1. Nth order IIR filter—C/assembly source code

```
typedef struct coef_t {
    uint16 order;      // only need uint8 but make uint16 so coef_ary is longword aligned
    uint16 shift;      // only need uint8 but make uint16 so coef_ary is longword aligned
```

```
    int16  coef_ary[]; // empty array must be last struct member
} coef_t;

int16 asm  __declspec(register_abi) iir(int16 input, const coef_t *coef, void *buffer) {
    // initialize registers
    lea.l   -12(A7),A7;                    // move stack pointer
    movem.l D4/A2-A3,(A7);                 // save registers on stack
    mvz.w   D0,D0;                         // load input data and clear upper word {y0,x0}
    lea.l   struct(coef_t.coef_ary)(A0),A2; // load address to coefficient array
    move.l  A1,A3;                         // copy address to data buffer
    move.l  %0x0080,MACSR;                 // configure MAC for signed integer with saturate
    move.l  %0,ACC;                        // clear accumulator
    mvz.w   struct(coef_t.order)(A0),D4;   // initialize inner loop counter

    // iterate i=0:order-1
  loop:
    move.l  (A2)+,D1;                      // load packed coefficients {a0,b0}
    msac.w  D0.u,D1.u;                     // acc -= a0*y0
    mac.w   D0.l,D1.l,(A3),D2;             // acc += b0*x0, load {y1,x1}
    move.l  D0,(A3)+;                      // shift buffer by writing {y0,x0} to {y1,x1}
    move.l  D2,D0;                         // copy {y1,x1} data for next iteration
    subq.l  %1,D4;                         // decrement inner loop counter
    bgt     loop;

    // one more iteration for i=order, but this time do not modify data buffer
    move.l  (A2)+,D1;                      // load packed coefficients {a0,b0}
    msac.w  D0.u,D1.u;                     // acc -= a0*y0
    mac.w   D0.l,D1.l;                     // acc += b0*x0, load {y1,x1}

    // read and shift accumulator
    move.l  ACC,D0;                        // read accumulator
    mvz.w   struct(coef_t.shift)(A0),D1;   // number of fractional bits in coefficients
    asr.l   D1,D0;                         // shift result
    jsr     saturate                       // saturate word

    // finish up
    move.w  D0,(A1);                       // write output to buffer
    movem.l (A7),D4/A2-A3;                 // restore registers from stack
    lea.l   12(A7),A7;                     // restore stack pointer
    rts;                                   // return
}

int16 saturate(int32 input) {
    if (input > (int32) 0x00007fff)
        return((int16) 0x7fff);
    if (input < (int32) 0xffff8000)
        return((int16) 0x8000);
    return((int16) input);
}
```

4.1.1 Sample coefficients data structures

Example 2. Nth order IIR filter—sample coefficients data structures

```
const coef_t lowpass1  = {1,14,{16384,8192,-8192,0}};
const coef_t lowpass2  = {1,14,{16384,4096,-12288,0}};
const coef_t lowpass3  = {1,14,{16384,2048,-14336,0}};
const coef_t lowpass4  = {1,14,{16384,1024,-15360,0}};
const coef_t lowpass5  = {1,14,{16384,512,-15872,0}};
```

```
const coef_t lowpass6  = {1,14,{16384,256,-16128,0}};
const coef_t highpass1 = {1,14,{16384,16384,-8192,-16384}};
const coef_t highpass2 = {1,14,{16384,16384,-12288,-16384}};
const coef_t highpass3 = {1,14,{16384,16384,-14336,-16384}};
const coef_t highpass4 = {1,14,{16384,16384,-15360,-16384}};
const coef_t highpass5 = {1,14,{16384,16384,-15872,-16384}};
const coef_t highpass6 = {1,14,{16384,16384,-16128,-16384}};
const coef_t deriv     = {1,14,{16384,16384,0,-16384}};
const coef_t leakyint  = {1,14,{16384,16384,-8192,0}};
const coef_t avg4      = {3,14,{16384,4096,0,4096,0,4096,0,4096}};
const coef_t lp_cheby2 =
{6,14,{16384,441,-16546,1637,20052,3191,-8837,3925,3957,3191,-624,1637,78,441}};
```

4.1.2 Matlab model

Example 3. Nth order IIR filter—Matlab model

```
function [hd,s] = mma955xL_nth_order_iir_filter(b,a);
% Fixed-point model of MMA955xL's Nth order IIR filter
%
% Usage:
% HD = nth_order_iir_filter(B,A) returns a discrete-time filter object with
% numerator and denominator coefficients in vectors B and A respectively.
%
% [HD,S] = nth_order_iir_filter(B,A) also returns a data structure string that
% can be copied directly into C code
%
% Notes:
% 1. Requires Matlab Filter Design and Fixed-Point Toolboxes
% 2. Saturation is not modeled exactly as implemented in ColdFire hardware. The
% hardware accumulator saturation is sticky, meaning that it must be explicitly
% cleared before becoming unsaturated. Matlab does not model this type of
% behavior.
%
% Example:
% [b,a] = butter(2,.5);
% hd    = nth_order_iir_filter(b,a);
% y     = filter(hd,x);
% freqz(hd);

% create the model
hd = dfilt.df1(b,a); % IIR direct form I realization
f = get(fi([b a]),'FractionLength'); % common fixed-point scale factor
set(hd,...
    'Arithmetic'       ,'fixed',...
    'CoeffAutoScale'   ,0,...
    'NumFracLength'    ,f,...
    'DenFracLength'    ,f,...
    'InputFracLength'  ,0,...
    'OutputFracLength' ,0,...
    'AccumWordLength'  ,32,...
```

```
    'AccumMode'         ,'KeepLSB',...
    'RoundMode'         ,'floor',...
    'OverflowMode'      ,'saturate');

% print a data structure string that can be copied directly into C code
x = [hd.Denominator/2^-hd.DenFracLength; hd.Numerator/2^-hd.NumFracLength];
x = reshape(x,1,numel(x));
s = sprintf('const coef_t coef =
{%g,%g,{%s}};',length(b)-1,f,regexprep(int2str(x),'\s*',','));
```

4.1.3 Matlab Model—MMA9559L (rounding)

Example 4. Nth order IIR filter—Matlab model: MMA9559L (rounding)

```
function [hd,s] = mma9559L_nth_order_iir_filter(b,a);
% Fixed-point model of MMA9559L's Nth order IIR filter
%
% Usage:
% HD = nth_order_iir_filter(B,A) returns a discrete-time filter object with
% numerator and denominator coefficients in vectors B and A respectively.
%
% [HD,S] = nth_order_iir_filter(B,A) also returns a data structure string that
% can be copied directly into C code
%
% Notes:
% 1. Requires Matlab Filter Design and Fixed-Point Toolboxes
% 2. Saturation is not modeled exactly as implemented in ColdFire hardware. The
% hardware accumulator saturation is sticky, meaning that it must be explicitly
% cleared before becoming unsaturated. Matlab does not model this type of
% behavior.
%
% Example:
% [b,a] = butter(2,.5);
% hd    = nth_order_iir_filter(b,a);
% y     = filter(hd,x);
% freqz(hd);

% create the model
hd = dfilt.df1(b,a); % IIR direct form I realization
f = get(fi([b a]),'FractionLength'); % common fixed-point scale factor
set(hd,...
    'Arithmetic'        ,'fixed',...
    'CoeffAutoScale'    ,0,...
    'NumFracLength'     ,f,...
    'DenFracLength'     ,f,...
    'InputFracLength'   ,0,...
    'OutputFracLength'  ,0,...
    'AccumWordLength'   ,32,...
    'AccumMode'         ,'KeepLSB',...
    'RoundMode'         ,'nearest',...
    'OverflowMode'      ,'saturate');
```

```
% print a data structure string that can be copied directly into C code
x = [hd.Denominator/2^-hd.DenFracLength; hd.Numerator/2^-hd.NumFracLength];
x = reshape(x,1,numel(x));
s = sprintf('const coef_t coef =
{%g,%g,{%s}};',length(b)-1,f,regexprep(int2str(x),'\s*',','));
```

4.2 Configurable cutoff IIR filter

4.2.1 C prototypes

```
typedef enum filter_type_tag {
    LOWPASS  = 0,
    HIGHPASS = -32768 /* this is the value of the coefficient for x[n-1] */
} filter_type_t
int16 config_cutoff_filter(int16 input, uint32 k, void *buffer, filter_type_t filter_type);
```

4.3 Goertzel algorithm

4.3.1 C/assembly source code

Example 5. Configurable cutoff IIR filter—C/assembly source code

```
/* This is a second-order IIR filter that evaluates the equation:
 *
 * y[n] = x[n] + coef*y[n-1] - y[n-2]
 *
 * Unlike a general IIR filter implementation, this function only accepts only
 * one coefficient as an input argument. The other coefficients of the filter
 * are fixed. To compute bin k of an N-point DFT, coef = 2*cos(2*pi*k/N). The
 * sf input argument is the number of fixed-point scaling bits of coef. Another
 * difference from a general IIR filter implementation is that this function
 * only requires a 2-entry buffer since the coefficients to x[n-1] and x[n-2]
 * are zero. It only needs to store y[n-1] and y[n-2]. */
inline asm  __declspec(register_abi) void goertzel_iir(int16 x, const int16 coef, const uint8 sf, int16
buffer[2]) {
    move.l  d3,-(sp);               // save register to stack
    moveq.l %1,d3;                  // one
    asl.l   d2,d3;                  // one, scaled
    move.l  %0,acc;                 // clear accumulator
    mac.w   d3.l,d0.l,(a0),d0;      // acc += 1*x[n]
    mac.w   d1.l,d0.u;              // acc += coef*y[n-1]
    msac.w  d3.l,d0.l;              // acc -= 1*y[n-2]
    swap.w  d0;                     // swap y[n-1] into LSByte
    move.w  d0,2(a0);               // save y[n-1] into buffer
    move.l  acc,d0;                 // read accumulator
    clr.l   d1;                     // clear register for rounding
    asr.l   d2,d0;                  // shift result
    addx.l  d1,d0;                  // round to nearest
    jsr     saturate;               // saturate word
    move.w  d0,(a0);                // save y[n] into buffer
    move.l  (sp)+,d3;               // restore register from stack
}

/* Use the buffer from the second-order IIR filter to compute the magnitude
```

```
 * squared of the DFT bin. This function evaluates the equation:
 *
 * mag_squared = y[n-2]*y[n-2] + y[n-1]*y[n-1] - coef*y[n-1]*y[n-2]
 */
inline asm  __declspec(register_abi) int32 dft_magnitude_squared(const int16 coef, const uint8 sf, int16
buffer[2]) {
    move.l  (a0),d2;                // load {y[n-1], y[n-2]}
    move.l  %0,acc;                 // clear accumulator
    mac.w   d2.l,d2.l;              // acc += y[n-2]*y[n-2]
    mac.w   d2.u,d2.u;              // acc += y[n-1]*y[n-1]
    muls.w  d2,d0;                  // y[n-2]*coef
    clr.l   d2;                     // clear register for rounding
    asr.l   d1,d0;                  // shift result
    addx.l  d2,d0;                  // round to nearest
    jsr     saturate;               // saturate word
    move.l  (a0),d2;                // reload {y[n-1], y[n-2]}, since saturate() may not preserve d2
    msac.w  d2.u,d0.l;              // acc -= y[n-1]*(y[n-2]*coef)
    move.l  acc,d0;                 // read accumulator
}

/* Top-level function to execute the goertzel algorithm. The output is updated
 * once every N samples. */
int32 goertzel(const uint32 N, const int16 coef, const uint8 sf, int16 x, goertzel_t * g) {
    asm{move.l %0x0080,MACSR;}      // configure MAC for signed integer with saturate
    goertzel_iir(x,coef,sf,g->buffer);
    if (++g->n >= N) {
        g->output = dft_magnitude_squared(coef,sf,g->buffer);
        g->n      = 0;
    }
    return(g->output);
}
```

4.3.2 Matlab model

Example 6. Configurable cutoff IIR filter—Matlab model

```
function [output,coef,y] = goertzel(k,N,x);
% Fixed-point model of MMA955xL Goertzel algorithm
%
% Usage:
% [OUTPUT,COEF,Y] = goertzel(K,N,X) returns the magnitude squared of the Kth
% bin of the N-point DFT for input vector X. COEF contains the quantized
% coefficient 2*cos(2*pi*k/N), and Y contains the buffer of the intermediate IIR
% filter.

% fixed-point math properties
F =
fimath('ProductMode','KeepLSB','MaxProductWordLength',32,'SumMode','KeepLSB','MaxSum
WordLength',32);

% create the IIR model and filter the data
coef           = 2*cos(2*pi*k/N);
[hd,s]         = mma955xL_nth_order_iir_filter([1 0 0],[1 -coef 1]);
y              = filter(hd,x);
```

```
% compute the magnitude squared of the DFT bin, updating the output once every N
samples
y.fimath      = F;
coef          = fi(coef,numerictype(1,16,hd.denfraclength),F);
T             = numerictype(y(1)*y(1));
output        = fi(zeros(size(x)),T);
for n=N:N:length(x)
    output(n:min(length(x),n+N-1)) = y(n-1)*y(n-1) + y(n)*y(n) -
y(n)*fi(y(n-1)*coef,y.numerictype);
end
```

5 Other Resources

5.1 CFDSPLIB: *Complimentary ColdFire Digital Signal Processing Library*

Freescale provides a free library of filter coefficients available for download from the website:

http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=CFDSPLIB

The web interface allows a user to visually select the type of frequency response desired – lowpass, highpass, or bandpass—as well as the filter order and normalized frequency cutoff. All of the filters present in the library have been tested in hardware. Once the desired filters are selected, a user can download a CodeWarrior project that contains source code for the filter coefficients and implementation. It is important to note that this filter function interface and implementation differs slightly from the Nth order IIR filter described in Nth order IIR filter, therefore the coefficients and data structures in are not directly compatible. The main difference between the two implementations is that the Nth order IIR filter uses a common fixed-point scale factor for numerator and denominator coefficients, while the CFDSPLIB implementation allows for different scale factors between the numerator and denominator coefficients. However, all the filters in CFDSPLIB use a Butterworth design, so they can be regenerated in Matlab with the built-in `butter()` command and requantized for use with the Nth order IIR filter. Alternately, the CFDSPLIB filter implementation can be installed onto the MMA955xL platform as user code.

If desired, coefficients from the CFDSPLIB can be converted into the MMA955xL format by rescaling and rearranging. First, compare the difference equations evaluated by the CFDSPLIB and MMA955xL IIR filter implementations:

$$y_{MMA955xL}[n] = 2^{-sf}\left(\sum_{k=0}^{N} b_k x[n-k] - \sum_{k=1}^{N} a_k y[n-k]\right) \quad \textbf{Eqn. 8}$$

$$y_{CFDSPLIB}[n] = 2^{-num_sf}\sum_{k=0}^{N} d_k x[n-k] + 2^{-den_sf}\sum_{k=1}^{N} c_k y[n-k] \quad \textbf{Eqn. 9}$$

These equations can be rearranged to show how to transform coefficients from the CFDSPLIB to the MMA955xL implementation:

$$b_k = 2^{sf - num_sf} d_k$$

$$a_k = -2^{sf - den_sf} c_k$$

Setting *sf=den_sf* reduces these transformations further:

$$b_k = 2^{den_sf - num_sf} d_k$$

$$a_k = -c_k$$

As a result, the numerator coefficients must be shifted by the difference in scale factors, while the denominator coefficients must be inverted.

Finally, the order of the coefficients array is different. The CFDSPLIB coefficients are arranged with the numerator coefficients in decreasing order first, followed by the denominator coefficients in decreasing order. The MMA955xL coefficients, on the other hand, are interleaved and in increasing order.

$$coef_ary_{CFDSPLIB}\,[\,] = \{d_N, \ldots, d_0, c_N, \ldots, c_0\}$$

$$coef_ary_{MMA955xL}[\,] = \{a_0, b_0, \ldots a_N, b_N\}$$

As an example, a fourth order bandpass filter from the CFDSPLIB:

Example 7. Fourth order bandpass filter

```
/* Butterworth Bandpass Filter, order=4, cutoff frequency=[0.50 0.55]*f_nyquist */
int16 butter4_bp_0_50_0_55_coef[10] = { 11624,    0,-23248,    0,11624,  // num
coefficients
                                 -13120,-4359,-29504,-4872,    0}; // den coefficients
uint8 butter4_bp_0_50_0_55_num_sf   = 21; // numerator fixed point scale factor
uint8 butter4_bp_0_50_0_55_den_sf   = 14; // denominator fixed point scale factor
uint8 butter4_bp_0_50_0_55_order    =  4; // filter order
```

This filter converted into MMA955xL format:

```
const coef_t coef = {4,14,{16384,91,4872,0,29504,-182,4359,0,13120,91}};
```

The conversion results in loss of precision for the numerator coefficients, however, and can adversely affect the frequency response of the filter.

6 Definitions and Acronyms

Table 3 provides the terms and their descriptions that are used within this document.

Table 3. Terms and definitions

Term	Description
FIR	Finite Impulse Response
IIR	Infinite Impulse Response
DFT	Discrete Fourier Transform
ADC	Analog to Digital Converter
MAC	Multiply Accumulator
ISA	Instruction Set Architecture
IDE	Integrated Development Environment
AFE	Analog Front End
FFT	Fast Fourier Transform

7 Related documentation

The MMA955xL platform features and operations are described in a variety of reference manuals, user's guides, and application notes. To find the most-current versions of these documents:

1. Go to the Freescale homepage at:

   http://www.freescale.com/

2. In the *Keyword* search box at the top of the page, enter the device number *MMA955xL*.
3. In the *Refine Your Result* pane on the left, click on the *Documentation* link.

Table 4 provides the documentation referenced within this document.

Table 4. Reference documentation

Description
Discrete-Time Signal Processing, second edition, by A. Oppenheim and R. Schafer (Upper Saddle River, NJ: Prentice Hall, 1998)
MMA955xL Intelligent Motion-Sensing Platform Data Sheet, Freescale Semiconductor Inc. (Document number: MMA955XL)
MMA955xL Intelligent Motion-Sensing Platform Software Reference Manual, Freescale Semiconductor Inc. (Document number: MMA955XLSWRM)
ColdFire DSP Library Reference Manual, Freescale Semiconductor Inc. (Document number: CFDSPLIBRARYRM)
MCF52223 ColdFire Integrated Microcontroller Reference Manual, Freescale Semiconductor Inc. (Document number: MCF52223RM)

8 Revision History

Revision 0 is the initial release of this document.

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

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

Freescale reserves the right to make changes without further notice to any products herein. Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale 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 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 does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/salestermsandconditions.

Freescale, the Freescale logo, AltiVec, C-5, CodeTest, CodeWarrior, ColdFire, C-Ware, Energy Efficient Solutions logo, Kinetis, mobileGT, PowerQUICC, Processor Expert, QorIQ, Qorivva, StarCore, Symphony, and VortiQa are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. Airfast, BeeKit, BeeStack, ColdFire+, CoreNet, Flexis, MagniV, MXC, Platform in a Package, QorIQ Qonverge, QUICC Engine, Ready Play, SafeAssure, SMARTMOS, TurboLink, Vybrid, and Xtrinsic are trademarks of Freescale Semiconductor, Inc. All other product or service names are the property of their respective owners.
© 2012 Freescale Semiconductor, Inc.

AN4464
Rev. 0
10/2012