AVR223: Digital Filters with AVR

Features

- **Implementations of Simple Digital Filters**
- **Coefficient and Data Scaling**
- **Fast Implementation of 2nd Order FIR Filter**
- **Compact Implementation Of 8th Order FIR Filter**
- **Fast Implementation of 2nd Order IIR Filter**
- **Compact Implementation of 8th Order IIR Filter**

Introduction

Applications involving processing of data from external analog sources (sensors) often require some kind of digital filtering of the data prior to using the data to respond to some external event. In applications where extremely high filter performance is required Digital Signal Processors (DSP) are usually chosen, but in many applications these are too expensive to use. In these cases 8- and 16-bit Microcontrollers (MCU) comes into the picture: They are inexpensive, efficient and have all the required I/O features and communication modules that the DSP often does not have.

The Atmel AVR microcontrollers are excellent for signal processing applications due to their powerful architecture, strong instruction set and built-in multi-channel 10-bit Analog to Digital Converter (ADC). The megaAVR® series further have a hardware multiplier, which is important in signal processing applications.

This document focus on the use of the AVR hardware multiplier, the use of the general purpose registers for accumulator functionality, how to scale coefficients when implementing algorithms on fixed point architectures, the actual implementation examples and finally possible ways to optimize/modify the implementations suggested.

If the reader has a need for understanding digital filters, transfer functions, frequency response, and other topics that are briefly mentioned here, most literature on the topic of digital signal processing can provide this information. A literature list is enclosed last in this document.

General Digital Filters

Two filter types are referred to when discussing digital filters, the Finite Impulse Response (FIR) filters and Infinite Impulse Response (IIR) filters. The output of FIR filters is based on filter input only and the filter output is therefore calculated using a non-recursive algorithm (feed-forward). The output of the IIR filter is based on both the input and a feedback of the filter output. The IIR filter output is thus calculated using a recursive algorithm.

The description of digital filters, covering both FIR and IIR filters, can be generalized to the discrete differential equation seen in Equation 1.

$$\sum_{m=0}^{M} a_m \cdot y[n-m] = \sum_{k=0}^{N} b_k \cdot x[n-k]$$

<div align="right">Equation 1</div>

The filter coefficients related to the feedback and the feedforward part of the generalized digital filter are respectively denoted by a_m and b_k. The filter input is x[n], where n refers to the number of a given sample – similarly y[n] is the output corresponding to sample n. The ranges of the summations M and N are referred to as the filter order or the filter length.

The right side of the equation describes the feedforward in the filter function and the left side the feedback in the filter function. A filter function can consist of either or both sides of the differential equation. If only the right side of the differential equation is used, the filter is a FIR filter (N 0 and M = 0). If both sides are used the filter is an IIR filter (N ≠ 0 and M ≠ 0). A special case is when only the left side of the differential equation is used (N = 0 and M ≠ 0). In that case the filter is called an "all-pole" filter. The all-pole filter is actually also an IIR type filter since all filters that involve feedback are having an infinite impulse response. In this document the IIR filter is however defined to be where ranges of the two summations are equal (M = N). The all-pole IIR filter implementation will therefore not be described separately in this context.

Often digital filters are described in the Z-domain. The Z-transform of Equation 1 is seen from Equation 2.

$$\sum_{m=0}^{M} a_m \cdot y[n-m] = \sum_{k=0}^{N} b_k \cdot x[n-k] \quad \xleftrightarrow{Z} \quad Y(z) \cdot \sum_{m=0}^{M} a_m \cdot z^{-m} = X(z) \cdot \sum_{k=0}^{N} b_k \cdot z^{-k}$$

<div align="right">Equation 2</div>

The topic of Z-transforms is left for the reader to study further, recommended reading could be [1]. For now it is sufficient to know that Z^X in the Z-domain represent a delay element and that the exponent determines the number of discrete delays generated.

The reason that the Z-transform of the differential equation is provided is that characteristics of the digital filters are described by their transfer function, H(z), in the Z-domain. The transfer function of a filter (or any system) describes the relation between the input and output of the filter (or system). See Figure 1.

Figure 1. System
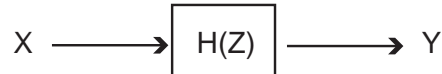
The general form of the relation between the filter input and the filter output, the transfer function of the filter, can be seen in Equation 3.

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{N} b_k \cdot z^{-k}}{\sum_{m=0}^{M} a_m \cdot z^{-m}}$$

<div align="right">Equation 3</div>

The numerator describes the feedforward part of the filter function and the denominator the feedback of the filter function.

Filter Stability

From the transfer function of the filter it is possible to determine if the filter is stable. A filter is said to be stable if a *bounded input generates a bounded output*, meaning that the output of the filter will die out when the filter input ceases. The criterion of stability is not that the output becomes zero if the input becomes zero, but that the magnitude of the output will decrease continuously toward zero (or becomes zero) if the input to the filter becomes zero. Without elaborating on the topic of filter stability, it should be known that the roots of the denominator polynomial, the poles that is, should be within the unit-circle for the filter to be stable. Further information on the topic of filter stability can be found in e.g. [1] and [2].

FIR Filters

FIR filters are characterized by being feed-forward filters, and there will thus only be an output from the filter as long a data are fed into the filter input tap. The duration from the input to the filter have ended to the output is "silent" (zero) depends on the order of the filter, since the order of the filter determines the number of delay elements that would be found in the filter tap delay line. An important feature of the FIR filter is that it is always stable, since a bounded input generates a bounded output.

The FIR filter is the special case of the general digital filter, where M = 0 and N ≠ 0. The mathematical description of the FIR filter can be seen in Equation 4.

$$\sum_{m=0}^{M} a_m \cdot y[n-m] = \sum_{k=0}^{N} b_k \cdot x[n-k] \quad ; \text{for} M = 0 \ \text{and} \ N \neq 0$$

$$\Updownarrow$$

$$y[n] = \sum_{k=0}^{N} b_k \cdot x[n-k]$$

Equation 4

The transfer function of the FIR in the Z-domain is seen in Equation 5.

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{k=0}^{N} b_k \cdot z^{-k}}{1} = \sum_{k=0}^{N} b_k \cdot z^{-k}$$

Equation 5

From both Equation 4 and the FIR transfer function (Equation 5) it can be seen that the filter output, y[n], sole rely on the input samples, x[n-k]. To make this even more obvious one can study the illustration of the FIR filter structure, seen in Figure 2. This filter structure is referred to as "Direct Form I".

Figure 2. FIR Filter, Direct Form I Structure
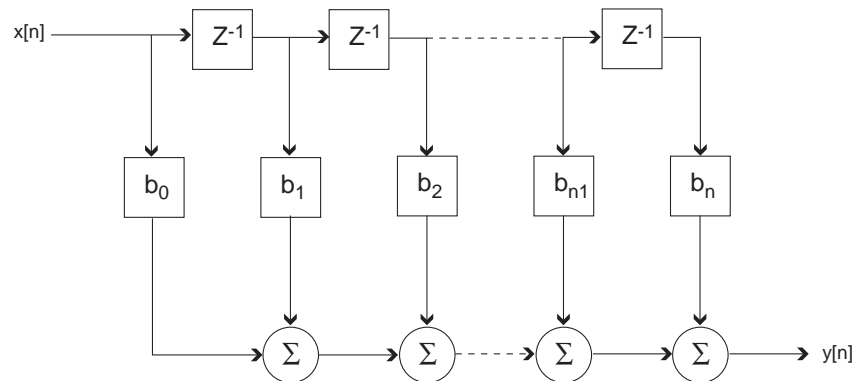
The arrows in the filter structure seen in Figure 2 represent the data flow in the filter. Boxes are functions, such as multiplications between signal and coefficients or discrete signal delays.

The FIR filter can be used for both High Pass (HP) and Low Pass (LP) filter – this is a matter of choosing the coefficient to match the desired frequency response. However, the transfer function is only containing "zeros" – root of the numerator polynomial, and since the zeros cause attenuation of the amplitude of signal filtered, the FIR is efficient to attenuate selected frequencies. The FIR filters are good at removing undesired frequency components of a signal. In Figure 3 a FIR filter magnitude response is illustrated. From this figure it is obvious that this specific filter is attenuating the normalized frequency[1] 0.5 by approximately 50 dB (by a factor 316). These filter coefficients thus generates a notch filter.

Figure 3. Magnitude Response for FIR Filter, b_K = {0.25, 0.25, 0.25, 0.25}
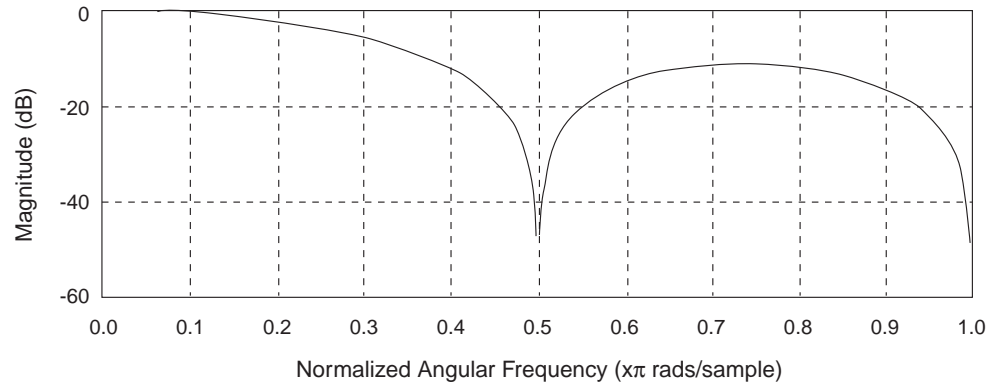
FIR filters are often used to smoothen signals and minimizing fluctuations in the signal due to background noise. This can be done e.g., by using an averaging filter, where all the coefficients equals 1/(N+1), where N is the filter order. By doing this the filter will produce the average of the last N+1 samples feed into the filter.

Note: 1. When working with discrete signals, frequencies are always normalized. When sampling a signal it is not possible to get valid information about the frequencies above half the sampling frequency, fs. Refer to Shannon's sampling theorem for more information on this. The normalization is thus done relative to fs/2.

IIR Filters

The IIR filter is a combination of both feedforward and feedback. The IIR filter is more efficient than the FIR filter, but have due to the feedback the disadvantage that the filter can be unstable. Following common guidelines for designing IIR filters this is not in general a problem however. If considering computation complexity with respect to the filter order the IIR filter is more computational heavy than the FIR filter. An IIR filter is approximately twice as complex as a FIR of the same order. The IIR filter is defined as the general form of the digital filter, with the assumption N = M ≠ 0.

Often the scaling of the coefficients is made so that the a_0 coefficient equals 1. The filter differential equation (Equation 1) can therefore be rearranged as follows from Equation 6. Presented in this way it is easier to understand what the filter output is generated from.

$$\sum_{m=0}^{M} a_m \cdot y[n-m] = \sum_{k=0}^{N} b_k \cdot x[n-k] \quad ; \text{for } a_0 = 1$$

$$\Updownarrow$$

$$y[n] = \sum_{k=0}^{N} b_k \cdot x[n-k] + \sum_{m=1}^{M} -a_m \cdot y[n-m]$$

Equation 6

The transfer function for the IIR filter in Z-domain can be seen from Equation 7.

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\displaystyle\sum_{k=0}^{N} b_k \cdot z^k}{1 + \displaystyle\sum_{m=1}^{M} a_m \cdot z^m} \quad ; \text{for } a_0 = 1$$

Equation 7

The structure of the IIR filter is partly similar to the structure of the FIR filter – the feedforward part is similar. In addition to the feedforward part of the filter there is also a feedback tap delay line. The structure of the IIR filter is illustrated in Figure 4.

Figure 4. IIR Filter Presented as Direct Form I Structure
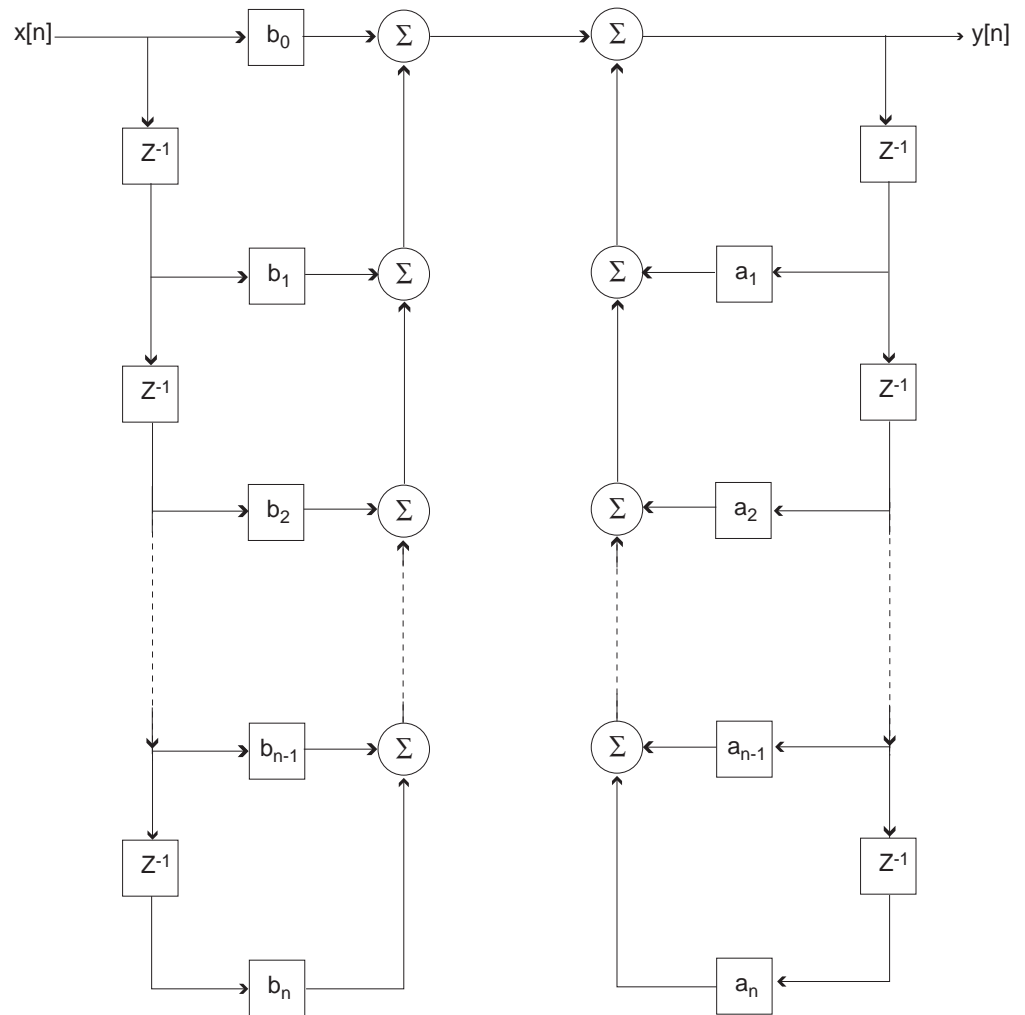
The representation of the IIR filter used in Figure 4 is called a Direct Form I structure, which is fairly efficient in terms of memory usage and ideally seen the fastest calculation method.

The force, and danger, of the IIR is the ability to "boost" frequencies, instead of attenuating. This boosting is due to the "poles" of the filter transfer function. Poles are the roots of the denominator polynomial of the transfer function. The boosting of frequencies due to the poles of the IIR filter can be seen from Figure 5, where the normalized frequency 0.5 are boosted by approximately 50 dB. The filter in Figure 5 is actually an all-pole filter and due to the boosting nature of the magnitude response this filter is called a comb filter.

Figure 5. IIR Filter Frequency Boosting
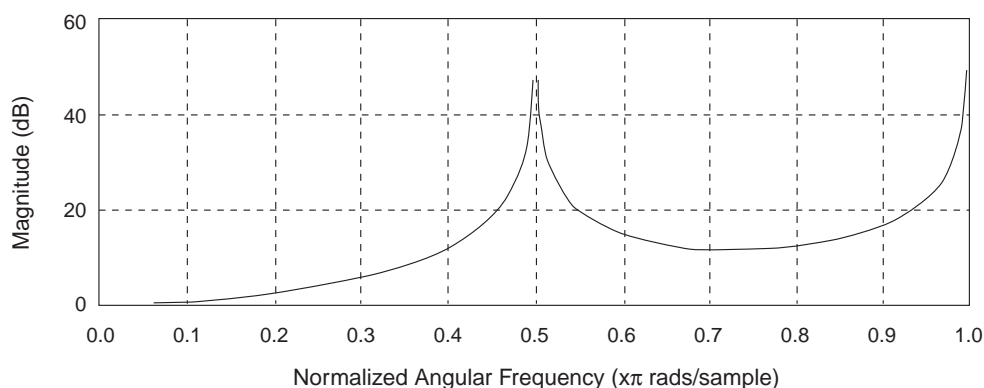
The IIR filter is often used in applications where single frequency or narrow frequency bands are of interest. This could e.g., be when an application where single tone detection is required.

Filter Implementation Considerations

When implementing a digital filter many factors must be considered. Some of the most relevant factors are frequency response characteristic (attenuation, steepness, phase shift and ripple), filter throughput, Signal-to-Noise Ratio (SNR) and data and coefficient scaling. Much of this is not directly related to MCU architecture and will therefore be left out of this document. This applies to e.g., the frequency response characteristic of the filter.

To understand how to make the implementations of digital filters on an AVR, knowledge about especially the hardware multiplier and the 32 8-bit general purpose registers (the Register File) is important.

The AVR Hardware Multiplier

Since the AVR is an 8-bit architecture the hardware multiplier is an 8-bit by 8-bit = 16-bit multiplier. The multiplier is invoked using three different instructions[1]: the MUL, the MULS and the MULSU. Unsigned, signed and signed times unsigned multiplications.

Note: 1.   The AVR also have instructions supporting a fractional multiplication. These are not used in this application note and are therefore not mentioned in this context. These instructions are described in the application note "AVR201: Using the AVR Hardware Multiplier".

In relation to the filters described in this application note the 8 by 8 multiplication is not sufficient. The filters require multiplications of 16-bit values 16 by 16. Multiplications and multiply-and-accumulate implementations are described in application note "AVR201: Using the AVR Hardware Multiplier" and are therefore not described in this document. However, the algorithm's requirements to the multiplication will be briefly highlighted.

The filter algorithm is a sum of products. The first product is calculated using a "simple" multiplication (MUL) between two N-bit values, providing a 2N-bit result. The multiplication is made between one byte from each of the multiplicands at a time. The result is stored in the 2N-bit "accumulator". The next products are calculated and added to the "accumulator", a so-called multiply-and-accumulate operation (MAC).

Four different multiply operations are used to implement the filter (see "AVR201: Using the AVR Hardware Multiplier"):

- muls16x16_32
- mac16x16_32
- muls16x16_24
- mac16x16_24

When and why the different multiply operations are used will become clear subsequently. In the filter implementation examples provided it is assumed that both the data and the coefficients are signed.

The AVR Virtual Accumulator

The AVR does not have a dedicated accumulator – instead the AVR allows any number of GP Registers to form a virtual accumulator. "Virtual accumulator" thus refers to a number of GP Registers that is combined to obtain accumulator functionality. Subsequently the virtual accumulator of the AVR is referred to as an "accumulator", though more flexible than ordinary accumulators used in other architectures, due to the possibility to scale the word length of the virtual accumulator.

If a 24-bit accumulator is required by the AVR to MUL two 12-bit values, three 8-bit GP Registers are combined into a 24-bit accumulator. If a MAC requires a 40-bit accumulator, five 8-bit registers are combined to form the accumulator. Using this flexibility of the accumulator ensures that no parts of the result or sub-result should be moved back and forth during the MAC operating, which would have been required if the accumulator size was fixed to 32-bit or less.

To understand the relevance of the flexibility of the AVR accumulator one should understand the problems related to implementing algorithms using fixed point MCU's: Values can only be described within a limited range and therefore overflow may occur in algorithms implemented on fixed point MCUs if this matter is not understood. Once understood, the problem can be avoided with ease.

Overflow of Fixed Point Values

Overflow can occur two places in the filter algorithm, in the output stage of the filter, i.e., in the final result, and in the sub-results of the algorithm.

To avoid overflow in the output stage, one of the design criteria is to avoid that the gain in the filter exceeds one. If the gain of the filter is less or equal to one, the output of the filter will never be larger than the input and will thus not overflow. Consider that a value represented in 16-bit, using the full range of the resolution, is multiplied by two (representating a gain larger than one); the result will no longer be possible to hold using 16-bit resolution – it requires 17-bit. It is relatively simple to avoid overflow in the output stage: Keep the filter gain low enough to be able to represent the result with the resolution available in the output stage.

The reason that overflow can occur in the results of algorithms is that any multiplication of two N-bit values can produce a 2N-bit result and any addition of two N-bit values can produce an (N+1)-bit result. Consider a fourth order FIR filter described by Equation 8.

$$y[n] = \sum_{k=0}^{4} b_k \cdot x[n-k]$$

Equation 8

The algorithm is a sum of five multiplications. If the data and the coefficients are both using a resolution of 16-bit the algorithm requires a 35-bit accumulator to store the result. The required resolution of the filter algorithm is specified in Equation 9.

$$N = K_1 + K_2 + \log_2(M)$$
$$= 16 + 16 + \log_2(5)$$
$$N = 34.3 = 35$$

Equation 9

N is the required number of bits to represent the result, K_1 and K_2 are the resolution and n is the number of values added.

Considering a fixed point implementation the resolution required when having a filter with unity gain would be somewhat similar to Equation 9. The difference would be that the maximum resolution would not be required by the final result, but by the sub-results. The resolution required for a filter with unity gain is described by Equation 10.

$$N = K_1 + K_2 + \frac{\log_2(M)}{2}$$

Equation 10

Where K_1 and K_2 are the resolution of the two multiplicands.

Consider the fourth order FIR filter of Equation 8 again. Assume that the filter has unity gain and that both the coefficients and the data are using 16-bit resolution. The resolution required to avoid overflow in the sub-results would be as described by:

$$N = 16 + 16 + \frac{\log_2(5)}{2}$$
$$N = 33$$

A fourth order FIR filter thus requires an accumulator word length of 34 bits if the data and the coefficient are both 16-bit.

By reducing the required resolution of the accumulator computations can be saved. To do this and to be able to implement the unity gain of the filter scaling of the coefficients must be understood.

Coefficient Scaling

Scaling of coefficients is the second important issue to understand to be able to implement digital filters and for that matter any algorithm on a fixed point architecture. Understanding the scaling of coefficients will make it possible to implement the filter to be fast and efficient.

Filter coefficient scaling is always an issue when working with digital filters using "Fixed" point Processors. When determining the coefficients from the desired characteristics of the filter frequency response, one will end up with a number of coefficients in floating point representation. The "Fixed" point Registers cannot store float values, so different "tricks" are used to make the coefficients fit into the "Fixed" point representation. The most important thing to keep in mind when scaling coefficients is to always scale the coefficients to use highest resolution possible. This will ensure that the filter is maintaining a correct frequency response.

The scaling of the coefficients is based on the fact of understanding how the coefficients utilize the resolution optimally according to the specific algorithm. In other words the coefficients are multiplied with the highest common value[1] that does not make any of the coefficients overflow the resolution range available. The fractional value 0.9001 can thus be multiplied by 2^{16}, and still be possible to represent by 16-bit resolution. The scaled value is 58988,9536 and will be rounded of to 58989, resulting in a relative rounding error of less than $8 \cdot 10^{-7}$. The higher resolution used for the coefficients the smaller rounding error. The rounding error will result in a change in the filter characteristic, so always double check the characteristics of the filter after rounding the coefficients.

Note: 1. Scaling is always done using scaling factors of 2's compliment, 2^X. The reason for doing this type of scaling is that the scaling then can be achieved using left and right shifts instead of multiplications and divisions. Especially divisions should be avoided, since they are not implemented in hardware and therefore will be "expensive" to use in terms of code size and code execution speed.

If coefficients are signed, which they often are, the sign will take up one bit of the available resolution. The value negative fractional value -0.9001 will not be possible to scale by 2^{16} without overflowing the 16-bit available. In this case the scaling should be 2^{15}. The relative rounding error is then less than $2 \cdot 10^{-5}$, due to the decrease in resolution.

At times it can be desired to decrease the resolution to speed up the algorithms. The reason is that by reducing the resolution of the coefficients and data[1], the required word length of the accumulator can be limited and data processing can be saved. Once again consider the fourth order FIR filter and assume that the data originates from the AVR ADC and therefore are of 10-bit resolution (though 16-bit is required to hold the 10-bit value). The required word length of the accumulator is calculated. See Equation 12.

Note: 1. Reducing the resolution of the data is not desired since this will introduce noise in the in the system and decrease the SNR in general. The effect of coefficient scaling is not introducing noise in the system, but can sometimes make it difficult to match the desired filter characteristic.

$$N = 10 + 16 + \frac{\log_2(5)}{2}$$
$$N = 27.2 = 28$$

Equation 12

The final result can be represented by 26-bit and the highest resolution required during the calculations is 28-bit. A 32-bit accumulator is required to run the algorithm.

Since there would be computational time saved if the resolution of the accumulator is reduced to 24-bit it could be desirable to reduce the resolution of the coefficients: Consider the fourth order FIR filter with 10-bit data input, if the resolution of the coefficients is decreased to 12-bit, the accumulator word length could be decreased to 24-bit. The sub-results of the filter will not exceed 24-bit if the data is 10-bit wide and the coefficients are 12-bit wide. The benefit would be that the MUL and MAC operations used in relation to a 24-bit accumulator would be executed 27% faster than if the algorithm was based on a 32-bit accumulator. The overall performance of the filter would increase by approximately 20%.

Eliminating the Gain Introduced by Coefficient Scaling

Unity gain has until now been mentioned without giving the details on the relation between the filter coefficients and the word length of the final result. The statement has been that if having a unity gain the 16 by 16 multiplication used in the filter algorithm would give a 32-bit result. Now, if there is no gain, how can the result be 32-bit wide when the data is 16-bit wide? The explanation is that if there should be not gain in the filter, the filter coefficients should be represented by fractional values, typically between -1 and 1. Since a fixed point architecture cannot represent fractional values, the filter coefficients are scaled. When scaling the coefficients initially the result has to be "unscaled" to eliminate the gain introduced by the coefficient scaling.

As mentioned in relation to the FIR filter example the coefficient scaling introduces a gain in the filter. The mathematical description of the scaling performed can be seen from Equation 13.

$$2^x \cdot \sum_{m=0}^{M} a_m \cdot y[n-m] = 2^x \cdot \sum_{k=0}^{M} b_k \cdot x[n-k]$$

$$\Updownarrow$$

$$2^x \cdot a_0 \cdot y[n] = \sum_{k=0}^{M} 2^x \cdot b_k \cdot x[n-k] + \sum_{m=0}^{M} 2^x \cdot (-a_m) \cdot y[n-m]$$

Equation 13

From Equation 13, it can be seen that the a_0 coefficient is scaled by the same factor as the rest of the coefficients, this explains why a gain is present and how large the gain is. The scaling or gain should be eliminated for two reasons; first, it will not be possible to compare the energy of the raw input signal to the energy of the filtered signal if a gain is present. If, e.g., it is required to compare the energy in a specific frequency to the total energy of the signal a unity gain is required. Second, if the gain is not removed the "true" value of y[n] will not be possible to use in the feedback and an error will be introduced in the filter. The second reason only relates to IIR filters since the FIR filters does not have feedback.

The gain introduced by the coefficient scaling is quite simple to eliminate in the output stage; Consider the FIR example where 10-bit data and 16-bit coefficients were used. The accumulator had to be 32-bit to be able to hold the sub-results, 10-bit comes in and a 26-bit result comes out of the filter. The 26-bit result is similar to a 10-bit value left shifted 16 times (since the coefficients are scaled with 2^{16}, which equals 16 left shifts of a binary value). To scale the output back to obtain the desired gain the result should be right shifted 16 times (scaled by 2^{-16}). Instead of actually shifting the result right 16 times the upper 16-bits of the 32-bit of the accumulator is used as the output of the filter. This is illustrated in Figure 6.

Figure 6. Un-scaling the Accumulator Value to Give Desired Filter Output
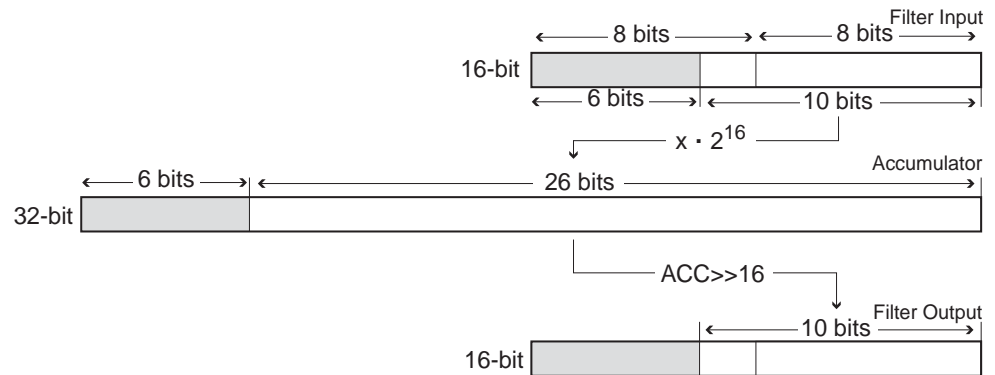
In this case the elimination of the gain is simplified because the coefficients are scaled by 2^{16}. If however the coefficients are scaled by a value that does not allow the down-scaling of the result to be accomplished by just grabbing the 16 MSB of the accumulator, the result stored in the accumulator could require several left or right shifts. If the coefficients are scaled by 2^{14}, the accumulator should be left shifted twice before returning the 16 MSB of the accumulator. This can also be seen from Equation 12, where y is the output, x is the input and the "rest" is scaling.

$$y = \left(\left(x \cdot 2^{14}\right) << 2\right) \cdot 2^{-16}$$

Equation 14

The foundation for implementing the filters should now be in place.

Filter Implementations

All filters described in this document are developed and compiled using the IAR EWAVR compiler version 2.26.

The filter coefficients used in the implementations are calculated using software made for this purpose. Much different software is capable of doing this. The software programs are ranging from costly mathematic programs such as Matlab™ to Java applets on the web. A list of web sites treating the topic of calculating filter coefficients are provided in the literature list enclosed last in the application note. An alternative is to calculate the coefficients the "hard" way: By hand. Methods for calculating the filter coefficients (and investigating stability of these filters) are described in [1] and [2].

Four different filters are implemented; Two FIR filters and two IIR filters. The two FIR filter implementations are: A second order High Pass (HP) filter and an eight order HP filter. The two IIR filter implementations are: A second order Band Pass (BP) filter and a sixth order BP filter.

The filters are implemented in assembly for efficiency reasons. The implementations are made in such a way that the filter function can be called from C. Prior to calling the filter functions it is required that the filter nodes (memory of the delay elements) are initialized – otherwise the startup condition of the filters are unknown. For all the filters, a C code example that initializes and calls the filter function is provided.

All parameters required by the filters are passed at run time and the filter functions can thus be reused to implement more than one filter without additional code space is required. This also allows cascade coupling of filters. Often multiple second order filters are used to form higher order filters by feeding the output of one filter into the input of the next filter, called cascading the filters.

The implementations all focus on fast execution of the filters, since having a high throughput in the filters is of great importance. See "Optimization of the Filter" on page 22 for suggestions of alternative implementations to reduce the code size.

A Second Order FIR Filter

Two High Pass (HP) filters are described. The filter function called is common to both of the filters, only the filter parameters differ. Both are based on the windowing design technique and both are of second order. The window used is the Hamming window. Refer to [1] for information on how to do this. Two different sets of filter coefficients are made to show how second order filters can be combined in cascade. The filter parameters are seen from Table 1.

Table 1. FIR Filter Parameters

Filter	Order	Cutoff	Coefficients {b0, b1, b2}	Scaling	Scaled Coefficients {b0, b1, b2}
No. 1	2	0.4	-0.0373, 0.9253, -0.0373	2^{12}	-153, 3790, -153
No. 2	2	0.6	-0.0540, 0.8920, -0.0540	2^{12}	-222, 3653, -222

The parameters specified in relation to filter No. 1 will result in a filter with a magnitude response as shown in Figure 2. The low order of the filter can be seen by the relative poor attenuation of the low frequencies and by the soft slope of the filter between the pass-band and the stop-band.

Figure 7. Magnitude Response of the Second Order Fir Filter No. 1, where the Cutoff Frequency is 0.4.
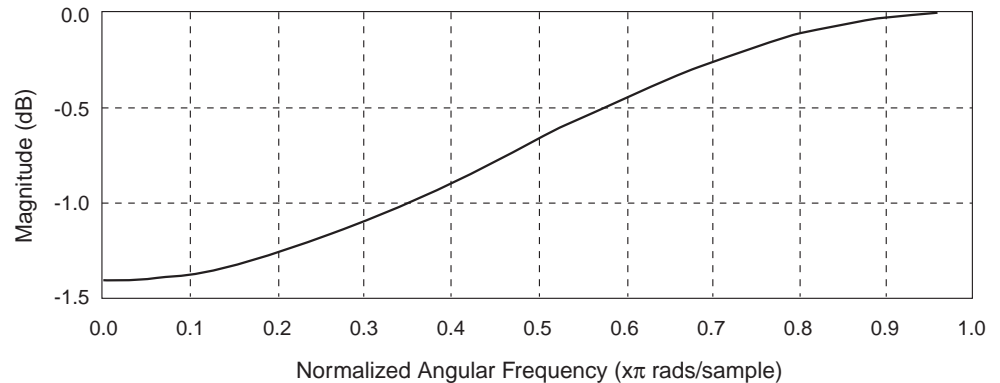
The filter routines are implemented in assembly to make them efficient, however the filter parameters and the filter nodes needs to be initialized prior to calling the filter function. The initialization is done in C. A struct containing the filter coefficients and the filter nodes are defined for each of the filters. The structs are defined as follows:

```
struct FIR_filter{
 int filterNodes [FILTER_ORDER];            //Filter nodes memory
 int filterCoefficients[FILTER_ORDER+1];    //Filter coefficients memory
 } filter04 = {0,0, B10, B11, B12},         //Init filter No. 1
   filter06 = {0,0, B20, B21, B22};         //Init filter No. 2
```

The `filterNodes` array is used as a FIFO buffer, holding previous input samples. The `filterCoefficients` array is used for the feedforward coefficients of the filter. Once the filter is initialized the filter function can be called. The filter function is defined as follows:

```
 int FIR2(struct FIR_filter *myFilter, int newSample );
```

In the assembly function the registers that needs to be preserved are first stored. The pointer to the filter struct is then copied into the Z-register, since this can be used for indirect addressing of data, i.e., for pointer operations. Then the core of the algorithm is ready to be run. The samples (data) are loaded and multiplied (MUL) with the matching coefficients. The products are added in the 24-bit wide accumulator. When all data and coefficients are multiplied-and-accumulated (MAC), the result are scaled down and returned. This can be seen from the flow chart in Figure 7 specific is a general description of the flow in a FIR filter algorithm and is therefore not providing information about the filter order, the coefficient scaling and the result down scaling.

The flow chart in Figure 8 shows the algorithm as if it was implemented using loops, this is done to increase the readability of the flow chart. The algorithm is implemented using straight-line code. Further, the flow chart shows that non-volatile registers are stored and restored respectively in the beginning and in the end of the algorithm. This is not actually done in the second order implementation, since it is not required to use non-volatile registers in that algorithm.

Figure 8. Data Flow in FIR Filter Algorithm
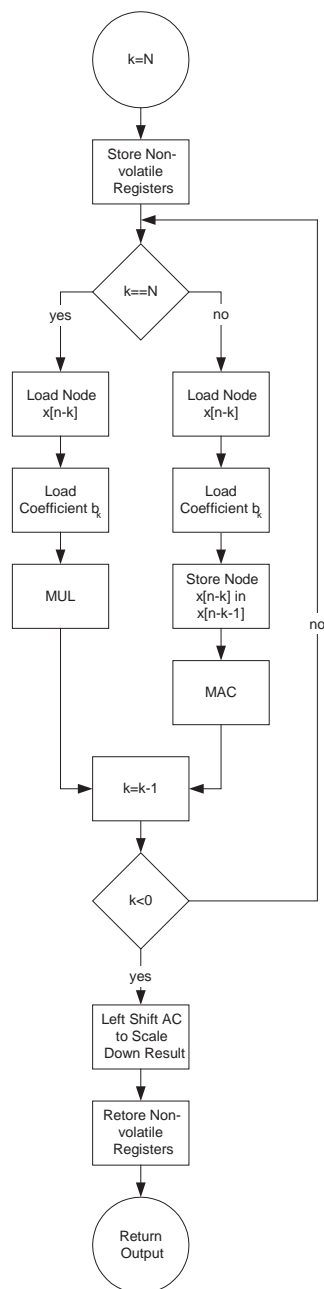
As mentioned the two filters can be placed in cascade. In the C file calling the filter function the cascading is shown. The idea of cascading is that the output from filter No. 1 is feed into the filter No. 2. In this way it is possible to obtain a fourth order filter by combining two second order filters. The principle is illustrated in Figure 9.

Figure 9. Combining Two Second Order FIR Filters to One Fourth Order FIR Filter by Cascading
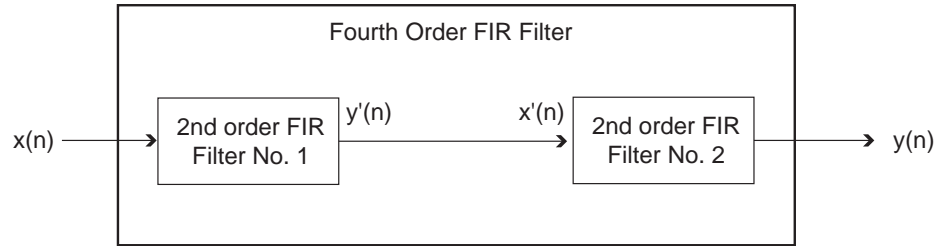
Filter Algorithm Performance

The performance of the algorithm in terms of code size (bytes), execution cycles and filter efficiency are given in Table 2. The numbers of instructions/cycles given in the table are all including calls and returns from functions.

Table 2. Performance of Second Order FIR Filter

Instructions [Filter + init]	Execution Cycles [Filter + init]	Filter Efficiency [Cycles/Order]
122 + 10[(1)]	100 + 135	50

Note: 1. The code required to initialize the filter is a routine that is default included by the compiler to initialize SRAM, the code size for the initialize routine is therefore only given as the additional code required to initialize the filter

From Table 2 it can be estimated that the second order FIR filter can handle up to 160 k samples/second if the AVR is run at 16 MHz core clock.

A Eight Order FIR Filter

The eight order FIR filter is like the second order FIR filter a HP filter. The filter coefficients for the eight order FIR filter are calculated in the same way as those for the second order FIR, using a Hamming window. The filter parameters are seen from Table 3.

Table 3. FIR Filter Parameters

Order	Cutoff	Coefficients {b0, b1, …, b8}	Scaling	Scaled Coefficients {b0, b1, …, b8}
8	0.4	0.0060, 0.0133, -0.0501, -0.2598, 0.5951, -0.2598, -0.0501, 0.0133, 0.0060	2^{12}	49, 108, -411, -2129, 4875, -2129, -411, 108, 49

The parameters specified will result in a filter with a magnitude response as shown in Figure 10. The high order of the filter (compared to the second order FIR filter) can be seen by the more powerful attenuation of the low frequencies, the more steep change between the pass-band and the stop-band and by the minimized attenuation of the frequencies in the pass-band.

Figure 10. Magnitude Response of Eight Order FIR Filter with a Cutoff Frequency at 0.4
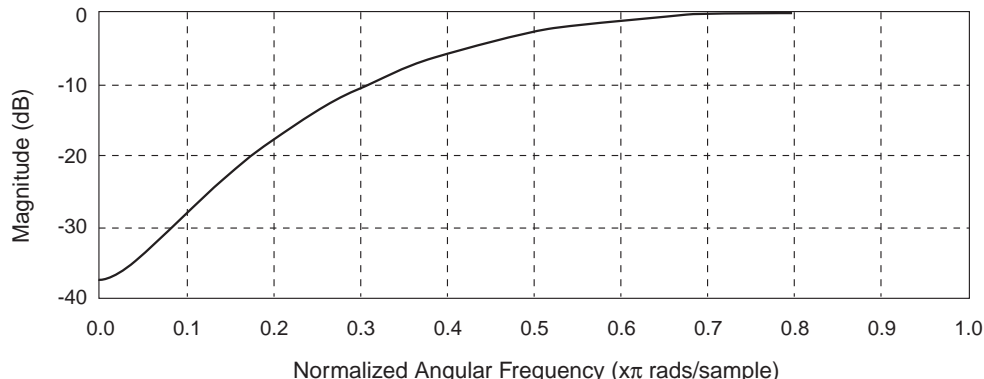
As for the second order FIR filter the eight order FIR filter's parameters and the filter nodes needs to be initialized prior to calling the filter function. The filter parameters are held in a struct, which is defined as follows:

```
struct FIR_filter{
 int filterNodes [FILTER_ORDER];//Filter nodes memory
 int filterCoefficients[FILTER_ORDER+1];//Filter coefficients memory
 } filter04 = {0,0,0,0,0,0,0,0,
             B0,B1,B2,B3,B4,B5,B6,B7,B8}; //Init filter
```

The `filterNodes` array is used as a FIFO buffer, holding previous input samples. The `filterCoefficients` array is used for the feedforward coefficients of the filter. Once the filter is initialized the filter function can be called. The filter function is defined as follows:

```
int FIR8(struct FIR_filter *myFilter, int newSample );
```

The eight order FIR filter function is very similar to the function used for the second order FIR filter. The only significant difference is that the accumulator used is 32-bit wide in the eight order FIR filter. The data flow chart of Figure 7 therefore also illustrated the data flow of the eight order FIR filter.

Filter Algorithm Performance

The performance of the algorithm in terms of code size (bytes), execution cycles and filter efficiency are given in Table 4. The numbers of instructions/cycles given in the table are all including calls and returns from functions.

Table 4. Performance of Eight Order FIR Filter

Instructions [Filter + init]	Execution Cycles [Filter + init]	Filter Efficiency [Cycles/Order]
456 + 20[1]	331 + 399	41

Note: 1. The code required to initialize the filter is a routine that is default included by the compiler to initialize SRAM, the code size for the initialize routine is therefore only given as the additional code required to initialize the filter.

From Table 4 it is seen that the eight order FIR filter is relatively more efficient than the second order filter; Less cycles are used per filter order. It can be estimated that the eight order FIR filter can handle up to 48 k samples/second if the AVR is run at 16 MHz core clock.

**A Second Order IIR Filter**　The IIR filters are a bit more complex to work with than FIR filter, but also more powerful. To illustrate the strength of the IIR filter a band-pass (BP) filter is implemented based on the Butterworth filter type. The filter parameters are seen from Table 5.

Table 5. FIR FIlter Parameters

Order	Cutoff	Coefficients {b0, b1, b2; a0, a1, a2}	Scaling	Scaled coefficients {b0, b1, b2; a0, a1, a2}
2	0.45 - 0.55	0.1367, 0, -0.1367; 1.0000, 0.0000, 0.7265	2^{11}	280, 0, -280; 2048, 0, 1488

The parameters specified will result in a filter with a magnitude response as shown in Figure 11. Compared to the FIR filters it can be seen that even a low order IIR filter has good ability to attenuate frequencies in the stop-band.

Figure 11. Magnitude Response of Second Order Butterworth Band-pass Filter with Cutoff Frequencies at [0.45 0.55]
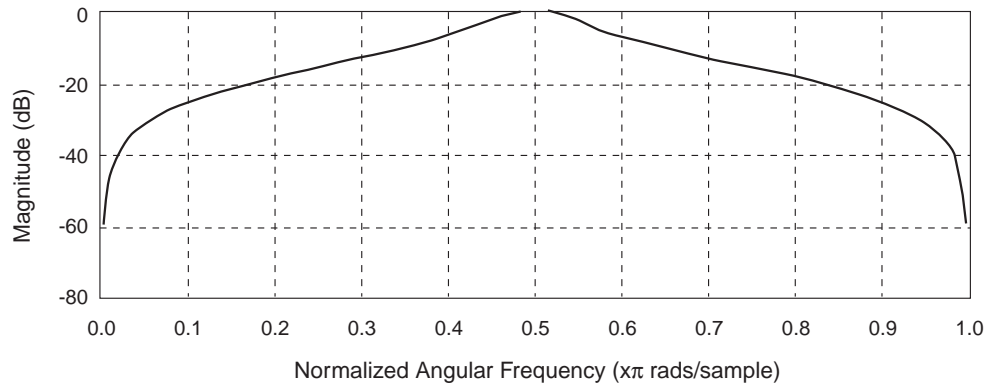
A struct containing the filter coefficients and the filter nodes are defined for the filter. The filter should be initialized before the filter function is called. The struct is defined as follows:

```
struct IIR_filter{
 int filterNodesX[FILTER_ORDER];          //filter nodes, stores x(n-k)
 int filterNodesY[FILTER_ORDER];          //filter nodes, stores y(n-k)
 int filterCoefficientsB[FILTER_ORDER+1]; //filter feedforward coefficients
 int filterCoefficientsA[FILTER_ORDER];   //filter feedback coefficients
 }filter04_06 = {0,0,0,0, B0, B1, B2, A1, A2};    //Init filter
```

The `filterNodesX` and `filterNodesY` arrays are used as FIFO buffers, holding previous input samples and previous output values respectively. The `filterCoefficientsB` and `filterCoefficientsA` arrays are used for the feedforward and the feedback coefficients of the filter respectively. Once the filter is initialized the filter function can be called. The filter function is defined as follows:

```
int IIR2(struct IIR_filter *myFilter, int newSample );
```

In the assembly function the registers that needs to be preserved are first stored. The pointer to the filter struct is then copied into the Z-register, since this can be used for indirect addressing of data, i.e., for pointer operations. Then the core of the algorithm is

ready to be run. The samples (data) are loaded and multiplied with the matching coefficients. The products are added in the 24-bit wide accumulator. When all data and coefficients are Multiplied-and-Accumulated (MAC) and the filter node FIFO buffers are updated. Finally the result are scaled down and returned. Note that y[n] (the result in the accumulator) is scaled down before the updating the y[n-1] FIFO buffer element. A data flow chart is seen in Figure 12.

Figure 12. Data Flow in IIR Filter Algorithm
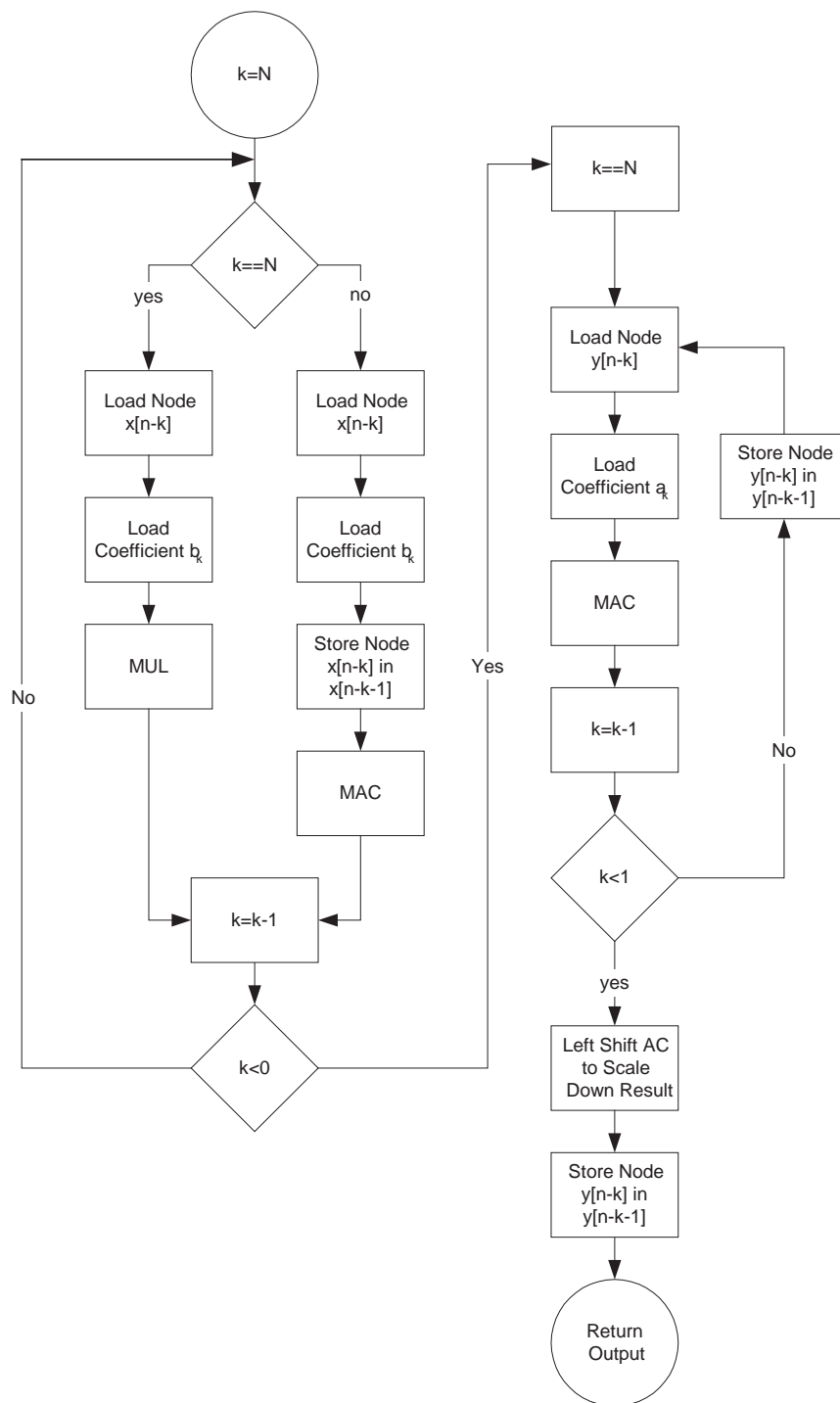
The flow chart in Figure 12 shows the algorithm as if it was implemented using loops, this is only to increase the readability of the flow chart. The algorithm is implemented using straight-line code. Further the flow chart shows that non-volatile registers are stored and restored respectively in the beginning and in the end of the algorithm.

Filter Algorithm Performance

The performance of the algorithm in terms of code size (bytes), execution cycles and filter efficiency are given in Table 6. The numbers of instructions/cycles given in the table are all including calls and returns from functions.

Table 6. Performance of Second Order IIR Filter

Code Size [Filter + init]	Execution Cycles [Filter + init]	Filter Efficiency [Cycles/Order]
194 + 10[1]	155 + 223	78

Note: 1. The code required to initialize the filter is a routine that is default included by the compiler to initialize SRAM, the code size for the initialize routine is therefore only given as the additional code required to initialize the filter.

From Table 6 it can be estimated that the second order IIR filter can handle up to 103 k samples/second if the AVR is run at 16 MHz core clock.

A Sixth Order IIR Filter

A band-pass (BP) filter is implemented based on the Butterworth filter type. The filter parameters are seen from Table 7.

Table 7. IIR Filter Parameters

Order	Cutoff	Coefficients {b0, b1,…, b6; a0, a1,…,a6}	Scaling	Scaled Coefficients {b0, b1,…, b6; a0, a1,…,a6}
6	0.4	0.0029, 0, -0.0087, 0, 0.0087, 0, -0.0029; 1.0000, 0.0000, 2.3741, 0.0000, 1.9294, 0.0000, 0.5321	214	47, 0, -142, 0, 142, 0, -47; 16384, 0, 38897, 0, 31611, 0, 8718

The parameters specified will result in a filter with a magnitude response as shown in Figure 10. The high order of the filter (compared to the second order FIR filter) can be seen by the more powerful attenuation of the low frequencies, the more steep change between the pass-band and the stop-band and by the minimized attenuation of the frequencies in the pass-band.

Figure 13. Magnitude Response of Sixth Order Butterworth Bandpass Filter with Cutoff Frequencies at 0.45 and 0.55
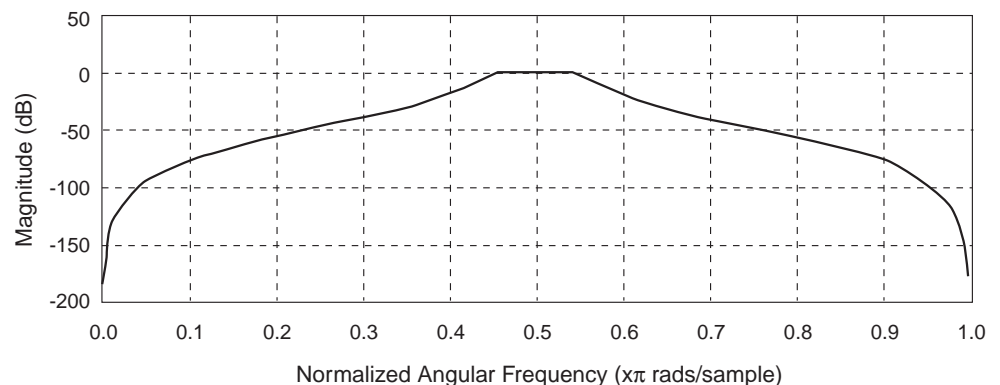
A struct containing the filter coefficients and the filter nodes are defined for the filter. The filter should be initialized before the filter function is called. The struct is defined as follows:

```
struct IIR_filter{
 int filterNodesX[FILTER_ORDER];         //filter nodes, stores x(n-k)
 int filterNodesY[FILTER_ORDER];         //filter nodes, stores y(n-k)
 int filterCoefficientsB[FILTER_ORDER+1]; //filter feedforward coefficients
 int filterCoefficientsA[FILTER_ORDER];   //filter feedback coefficients
}   filter04_06 = {0,0,0,0,0,0,
     0,0,0,0,0,0,
     B0,B1,B2,B3,B4,B5,B6,
     A1,A2,A3,A4,A5,A6};                   //init filter
```

The f$_{ilterNodesX}$ and filterNodesY arrays are used as FIFO buffers, holding respectively previous input samples and previous output values. The filterCoefficientsB and filterCoefficientsA arrays are used for respectively the feedforward and the feedback coefficients of the filter. Once the filter is initialized the filter function can be called. The filter function is defined as follows:

```
int IIR6(struct IIR_filter *myFilter, int newSample );
```

The sixth order IIR filter function is very similar to the function used for the second order IIR filter. The only significant difference is that the accumulator used is 32-bit wide in the sixth order IIR filter. The data flow chart of Figure 12 therefore also illustrated the data flow of the eight order FIR filter.

Filter Algorithm Performance The performance of the algorithm in terms of code size (bytes), execution cycles and filter efficiency are given in Table 2. The numbers of instructions/cycles given in the table are all including calls and returns from functions.

Table 8. Performance of sixth Order IIR Filter

Instructions [Filter + init]	Execution Cycles [Filter + init]	Filter Efficiency [Cycles/Order]
640 + 10[1]	463 + 575[1]	77

Note: 1. The code required to initialize the filter is a routine that is default included by the compiler to initialize SRAM, the code size for the initialize routine is therefore only given as the additional code required to initialize the filter.

From Table 8 It can be seen that the sixth order IIR filter is not significantly more efficient in terms of cycles per filter order. This is mainly caused by the use of a 32-bit accumulator in the sixth order IIR filter (compares to 24-bit in the second order IIR). The benefit of using a high filter order is thus relatively small in terms of performance.

It can be estimated that the sixth order IIR filter can handle up to 34 k samples/second if the AVR is run at 16 MHz core clock.

Optimization of the Filter

Optimization of the filters described in this application note is possible. This could be required to obtain a smaller code size or to increase the throughput of the filter. Suggestions of how to do this are found below.

Optimizing for Smaller Code Size

It is possible to achieve filters implementations faster than those described in this application note. One way of doing this is to ensure that only relevant calculations are actually performed. Due to the general form of the implementations in this application note the algorithms are, e.g., not optimized to leave out multiplications where the filter coefficients are zero. Several of the coefficients of the sixth order IIR band-pass filter (see Table 7) are zero and if the filter should be used in an application it would be evident to remove all MUL and MAC operations on these zero-coefficients since they do not contribute to the result anyway. By removing the MUL and MAC operations on the zero-coefficients the filter would be reduced by 6 of 13 MUL/MAC operations, which would decrease the execution time by approximately 45%. The code size reduction would be equivalent if the filter is using a straight-line implementation.

Optimizing for Increased Throughput

If a code size reduction is further needed, the filters can be implemented using calls to the MAC operation. This would reduce the code size significantly. The cost would be a decrease in performance however. This will have most effect on high order filters, since these have the most macro-calls to the MAC operation. Using the eight order FIR and the sixth order IIR filters as examples it can be seen that the code size can be reduced by respectively 43% and 58% (see Table 9). Another way to reduce the code size of high order filters is to use cascading of low second order filter elements.

Table 9. Comparisons Between Straight-line Implementations and Implementations Using Call to the MAC Operation

	Straight Line Implementation		MAC Call Implementation	
Filter Type	Performance	Code Size	Performance	Code Size
Eight Order FIR	331	456	387	222
Sixth Order IIR	463	636	547	266

Other Optimization

The filters are currently made so that the filter coefficients are placed in SRAM. This means that they are moved from FLASH to SRAM in the start-up code. Since the coefficients are constants they could also be located in FLASH and fetched directly from the FLASH when required. This will potentially save half the SRAM used.

References

[1] "Discrete-Time signal processing", A. V. Oppenheimer & R. W. Schafer. Prentice-Hall International Inc. 1989. ISBN 0-13-216771-9

[2] "Introduction to Signal Processing", S. J. Orfanidis, Prentice Hall International Inc., 1996. ISBN 0-13-240334-X

[3] FIR filter design, http://www.iowegian.com/scopefir.htm

[4] FIR filter design, http://www.dsptutor.freeuk.com/FIRFilterDesign/FIRFiltDes102.html

[5] FIR filter design,
http://www.dsptutor.freeuk.com/KaiserFilterDesign/KaiserFilterDesign.html

[6] IIR filter design, http://www.apogeeddx.com/BQD_Appnote.PDF

[7] IIR filter design, http://moshier.ne.mediaone.net/ellfdoc.html

[8] FIR and IIR filter design, http://www-users.cs.york.ac.uk/~fisher/mkfilter/

[9] FIR and IIR filter design, http://www.nauticom.net/www/jdtaft/papers.htm