Digital Filtering with AT89LP6440

Features
• Implementation of FIR and IIR Digital Filters
• Example 23rd Order FIR Low-Pass Filter
• Example 5th Order IIR Low-Pass Filter
• C and Assembly source code for SDCC and Keil® compilers targeting AT89LP3240 or AT89LP6440 devices
• Also can be adapted for AT89LP51RB2, AT89LP51RC2, AT89LP51RD2, AT89LP51ED2, AT89LP51IC2 or AT89LP51ID2 devices
• Sampling Rates in Audio Range

1. Introduction
Applications involving processing of signals from external analog sources/sensors usually require some kind of digital filtering. For extremely high filter performance, Digital Signal Processors (DSP) are usually chosen, but in many cases these are too expensive to use. In these cases, 8- and 16-bit Microcontrollers (MCU) come into the picture. They are inexpensive, efficient, and have all the required I/O features and communication modules that DSP seldom have.

The Atmel® AT89LP 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). Some devices, such as the AT89LP3240 and AT89LP6440, have a hardware multiply-accumulate (MAC) unit, which is important in signal processing applications. This document focuses on the use of the MAC, the use of the data pointer circular buffer functionality, how to scale coefficients when implementing algorithms on fixed point architectures, and possible ways to optimize a filter implementation. Two example implementations are included.

Although digital filter theory is not the focus of this application note, some basics are covered. A list of suggested, more in-depth literature on digital filter theory is enclosed last in this document.

2. Filter Theory
All digital, linear, time-invariant (LTI) filters can be described by a difference equation on the form shown in Equation 1. The output signal is denoted by \( y[n] \) and the input signal by \( x[n] \).

**Equation 1: General Difference Equation for Digital Filters.**

\[
\sum_{i=0}^{M} a_i \cdot y[n-i] \quad \sum_{j=0}^{N} b_j \cdot x[n-j]
\]

A filter is uniquely defined by its order and coefficients \( a_i \) and \( b_j \). The order of a filter is defined as the largest of \( M \) and \( N \), denoting the longest delay used in the calculations. Note that the coefficients are usually scaled so that \( a_0 \) equals 1. The output of the filter may then be calculated as shown in Equation 2.
Equation 2: Difference Equation for Filter Output.

\[ y[n] = \sum_{j=0}^{N} b_j \cdot x[n-j] + \sum_{i=0}^{M} (-a_i) \cdot y[n-i] \]

If \( x[n] \) is an impulse (1 for \( n = 0 \) and 0 for \( n > 0 \)), the output is called the filter’s impulse response: \( h[n] \).

A filter may be classified as one of two types from the value of \( M \):

- Finite Impulse Response (FIR), for \( M = 0 \)
- Infinite Impulse Response (IIR), for \( M \neq 0 \)

The difference between these two types of filters is the feedback: For IIR filters the output samples are calculated recursively, i.e., from previous output in addition to the input samples. The term Finite/Infinite then describes the length of the filter’s impulse response (disregarding quantization effects in a real implementation). Note that an IIR filter with \( N = 0 \) is a special case of filters, called “all-pole”. For further information on these two classes of filters, refer to the suggested literature list at the end of this document.

Often, digital filters are described in the Z-domain, a complex frequency domain. The Z-transform of Equation 1 is shown in Equation 3.


\[ Y(z) = \sum_{i=0}^{M} a_i \cdot z^{-i} = X(z) \sum_{j=0}^{N} b_j \cdot z^{-j} \]

The transfer function, \( H(z) \), for a filter is usually supplied as it tends to give a compact representation and allows for easy frequency analysis. The transfer function is defined as the ratio between output and input of a filter in the Z-domain, as shown in Equation 4. Note that the transfer function is the Z-transform of the filter’s impulse response, \( h[n] \).

Equation 4: Transfer Function of General Digital Filter.

\[ H(z) = \left( \frac{Y(z)}{X(z)} \right) = \frac{\sum_{j=0}^{N} b_j \cdot z^{-j}}{\sum_{i=0}^{M} a_i \cdot z^{-i}} = \frac{\sum_{j=0}^{N} b_j \cdot z^{-j}}{1 + \sum_{i=1}^{M} a_i \cdot z^{-i}} \]

As can be seen, the numerator describes the feed-forward part and the denominator describes the feedback part of the filter function. For more information on the Z-domain, refer to the suggested literature at the end of this document.

For the purposes of implementing a filter from a given transfer function, it is sufficient to know that \( z \) in the Z-domain represents a delay element, and that the exponent defines the delay length in units of samples. Figure 2-1 illustrates this with a general digital filter in Direct Form 1 representation.
3. Filter Implementation Considerations

When implementing a filter on a given MCU architecture, several issues must be considered. For example:

- The resolution (number of bits) of input and output will affect the maximum allowable filter gain and throughput.
- The resolution of filter coefficients will affect the frequency response and throughput.
- The filter order will affect the throughput.
- Fractional filter coefficients require some thought when used with an integer multiplier.

These and other implementation issues are discussed in this section. Another important consideration is the choice between FIR and IIR. FIRs are easier to implement (smaller code size) and have fewer sources of error than IIRs (scaling of only one set of coefficients and one result, and no feedback). However, without feedback the FIR requires a higher order filter to achieve the same quality of results as an IIR, e.g. the IIR can operate at higher throughput for a given filter response.

In addition, a quick description of the AT89LP multiply-accumulate unit and circular buffer addressing is in order, since knowledge about these is important for understanding the filtering code.
3.1 Multiply–Accumulate Unit (MAC)

The filter algorithm uses a convolution, or sum of products, calculation that is common in digital signal processing. The AT89LP6440 includes a multiply and accumulate (MAC) unit that can significantly speed up this type of mathematical operation. The MAC unit includes a 16-by-16 bit multiplier and a 40-bit adder that can perform integer or fractional multiply-accumulate operations on signed 16-bit input values. The MAC unit also includes a 1-bit arithmetic shifter that will left or right shift the contents of the 40-bit MAC accumulator register (M).

A block diagram of the MAC unit is shown in Figure 3-1. The 16-bit signed operands are provided by the register pairs (AX,ACC) and (BX,B) where AX and BX hold the higher order bytes. The 16-by-16 bit multiplication is computed through partial products using the AT89LP6440’s 8-bit multiplier. The 32-bit signed product is added to the 40-bit M accumulator register. The MAC operation is summarized as follows:

\[
\text{MAC AB: } M \leftarrow M + (AX \cdot ACC) \times (BX \cdot B)
\]

All computation is done in signed two’s complement form.

**Figure 3-1. Multiply–Accumulate Unit**

The MAC operation is performed by executing the MAC AB (A5 A4H) extended instruction. This two-byte instruction requires nine clock cycles to complete. The operand registers are not modified by the instruction and the result is stored in the 40-bit M register.

Three additional extended instructions operate directly on the M register. CLR M (A5 E4H) clears the entire 40-bit register in two clock cycles. LSL M (A5 23H) and ASR (A5 03H) shift M one bit to the left and right respectively. Right shifts are done arithmetically, i.e. the sign is preserved.

The 40-bit M register is accessible 16-bits at a time through a sliding window as shown in Figure 3-2. The MRW1,0 bits in DSPR select which 16-bit segment is currently accessible through the MACL and MACH addresses. For normal fixed point operations the window can be fixed to the rank of interest. For example, multiplying two 1.15 format numbers places a 2.30 format result in the M register. If MRW is set to 10B, a 1.15 value is obtained after performing a single LSL M.
3.1.1 Avoiding Overflow in Sub-Results

The M register is sized to avoid any overflows during normal operation. Multiplying two 16-bit numbers gives a 32-bit result that is always less than the maximum 32-bit value. The extra 8 guard bits of the M register allow up to 256 products to be summed before possibility of overflow. The MAC AB instruction updates the C and OV flags in PSW. C represents the sign of the MAC result and OV is the two’s complement overflow. Note that MAC AB will not clear OV if it was previously set to one. Practically, the OV flag need not be checked unless more than 256 products are summed without clearing the M register or if the register is preloaded with a 33-bit or larger value.

3.2 Circular Buffer Addressing (FIFO)

As noted above, the \( z \) elements in Figure 2-1 represent delay elements holding the previous input samples or output values. These are normally implemented using a first-in/first-out (FIFO) structure in data memory. FIR filters require one FIFO and IIR filters require two. The size of the first FIFO is equal to the number of coefficients, that is the order of the filter plus one, while the size of the second FIFO (if needed) is one less than the first FIFO. The AT89LP6440 provides two 16-bit data pointers: DPTR0 formed by the register pair DPOL and DPOH, and DPTR1 formed by the register pair DP1L and DP1H. The data pointers are used by several instructions to access the program or data memories. The AT89LP6440 supports the following operating modes for the data pointers that can help speed up FIFO access:

- Automatic postincrement or postdecrement of the data pointer to avoid a separate address update
- Circular buffer mode which forces the data pointer contents to remain within a fixed address range by wrapping around in a circular fashion
- Disable index mode of the MOVC instruction
- Allow MOVX or MOVC instructions to use the B register in place of ACC

The CBE0 and CBE1 bits in DSPR can configure DPTR0 and DPTR1, respectively, to operate in circular buffer mode. The AT89LP6440 maps circular buffers into two identically sized regions of EDATA/XDATA. These buffers can speed up memory accesses for convolution computations. The length of the buffers are set by the value of the FIRD register for up to 256 entries. Buffer A is mapped from 0000H to FIRD and Buffer B is mapped from 0100H to 100H+FIRD as shown in Figure 3-3. Both data pointers may operate in either buffer. When circular buffer mode is enabled, updates to a data pointer referencing the buffer region will follow circular addressing rules. If the data pointer is equal to FIRD or 100H+FIRD any increment will cause it to overflow to 0000H or 0100H respectively. If the data pointer is equal to 0000H or 0100H any decrement will cause it to underflow to FIRD or 100H+FIRD respectively. In this mode, updates can be
either an explicit INC DPTR or an automatic update using DPU\textsubscript{n} where the DPD\textsubscript{n} bits control the direction. The data pointer will increment or decrement normally at any other addresses. Therefore, when circular addressing is in use, the data pointers can still operate as regular pointers in the FIRD+1 to 00FFH and greater than 100H+FIRD ranges.

Practically, FIFO values will always be equivalent to two-byte words. Since FIRD encodes the number of bytes in the FIFO plus one, FIRD should be set to twice the number of words minus one.

$$FIRD = (2N) - 1 \quad \text{where} \quad (N = \text{length in words})$$

![Circular Buffer Mode](image-url)

3.2.1 Data Pointer Auto-Update

The Dual Data Pointers on the AT89LP6440 include two features that control how the data pointers are updated. The Data Pointer Configuration Register (DPCF) controls operation of the dual data pointers. The data pointer decrement bits, DPD1 and DPD0 in DPCF, configure the INC DPTR instruction to act as DEC DPTR. The resulting operation will depend on DPS.

The data pointer update bits, DPU1 and DPU0, allow MOVX @DPTR and MOVC @DPTR instructions to update the selected data pointer automatically in a post-increment or post-decrement fashion. The direction of update depends on the DPD1 and DPD0 bits. The data pointer update feature is employed in the main filter loop to speed up execution.

3.2.2 Data Pointer Redirect

The Data Pointer Redirect to B bit, DPRB (DSPR.0), allows MOVX and MOVC instructions to use the B register as the data source/destination when the instruction references DPTR\textsubscript{1}. DPRB can improve the efficiency of routines that must fetch multiple operands from different RAM locations. The filter routine uses this feature to load the coefficient values directly into the B register.

3.2.3 Index Disable

The MOVC Index Disable bit, MVCD (DSPR.1), disables the indexed addressing mode of the MOVC A, @A+DPTR instruction. When MVCD = 1, the MOVC instruction functions as MOVC A, @DPTR with no indexing. MVCD can improve the efficiency of routines that must fetch multiple operands from program memory. The filter routine uses this feature to load the coefficient values without disturbing the accumulator.
Digital Filtering with AT89LP6440

3.3 Scaling of Coefficients

Another important issue is the representation of the filter coefficients on Fixed Point (FP) architectures. FP representation does not necessarily mean that the values must be integers. The normal fixed point representation is for values to be in the range -1.0 to 1.0. For 16-bit precision this means a 1.15 representation, i.e. 15 bits to the right of the binary point and one sign bit to the left. These values are thus equivalent to integers in the range -32768 to 32768. Note that since two’s complement is used, 1.0 or 32768 are not achievable.

Naturally, to most accurately represent a number, one should use as many bits as possible. Therefore the standard practice is to “normalize” the coefficients into the desired range. Practically this means scaling all the coefficients by the largest common factor that does not cause any overflows in their representation. This scaling also applies to the a0 coefficient, so a downsampling of the output is necessary to get the correct value (especially for IIR filters). 16-bit division is not implemented in hardware, so the scaling factor should be of the form 2^k since division and multiplication by factors of 2 may easily be done with bitshifts. The principle of coefficient scaling and subsequent downsampling of the result is shown in Equation 5.

Equation 5: Scaling of Filter Coefficients and Downsampling of Result

\[
2^k \cdot \sum_{i=0}^{M} a_i \cdot y[n-i] = 2^k \cdot \sum_{j=0}^{N} b_j \cdot x[n-j] \\
y[n] = \left( \sum_{j=0}^{N} 2^k \cdot b_j \cdot x[n-j] + \sum_{i=0}^{M} (-2^k \cdot a_i) \cdot y[n-i] \right) >> k
\]

Note that the sign bit must be preserved when downsampling. This is easily done with the ASR instruction (arithmetic shift right). A further consideration on fixed-point systems is that a 1.15 by 1.15 multiplication gives a 2.30 format result, i.e. 2^{15} \times 2^{15} = 2^{30}. Therefore to restore the full 16-bit precision a single left shift (LSL instruction) is required. Practically this usually means just performing one less ASR than necessary when downsampling.

Optimization of the downsampling is possible if the factor k is above a multiple of 8. If this is the case, the program may simply "ignore" bytes of the result. The MRW bits that control the sliding M window (See Figure 3-2) can be used to select which bytes of the result are of interest. The result with still need to be shifted k MODULO 8 times.

Naturally, the coefficient with the largest absolute value will be the one limiting the maximum scaling factor. For IIR filters, if the a and b coefficients are of much different ranges, two separate scaling steps may be required to maintain the highest precision as shown in Equation 6.

Equation 6: Independent Scaling of IIR Coefficients and Downsampling of Results

\[
y[n] = \left( \sum_{j=0}^{N} 2^k \cdot b_j \cdot x[n-j] \right) >> k + \left( \sum_{i=0}^{M} (-2^k \cdot a_i) \cdot y[n-i] \right) >> h
\]

For this application the B \cdot X computation is done first and left in the accumulator. In this case once the result is downscaled, it must be scaled to the same range as the A \cdot Y computation as shown in Equation 7. In this case only the h factor and not the k factor can be optimized by MRW as the subresult in the accumulator after the B \cdot X computation must be scaled fully before starting the A \cdot Y computation.
3.3.1 Effect of Downscaling

One may wonder why one adds bits to avoid overflow in the sub-results, then basically “throws them away” to fit the result into a specified resolution. The explanation is that the bits are needed for precision during calculations, the filter has unity gain, and the output should be interpreted as an integer.

For filters with unity gain, the coefficients will be fractional, i.e., less than one, and thus the multiplications will add bits that actually represent fractional values. The summations will, however, add bits that represent higher significance. But due to the unity gain of the filter, these bits will never be used in the result: The output of the filter will not get an absolute value higher than that of the input, thus allowing the output to be represented with the same integer range as the input.

The downscaling simply removes the fractional part of the result, leaving just the integer part in the wanted resolution. Clearly, this also means that the precision is reduced. This is of consequence to IIR filters, since they have feedback.

3.3.2 Reduced Resolution

Given that the hardware MAC operation is geared toward 16-bit operands, there is nothing to be gained from reducing resolution in terms of throughput or code size unless the values could be represented in 8-bits or less. In practice it is difficult to achieve a viable filter with only 8-bit resolution. Therefore coefficients should always use the full 16-bits available. Input values may have less resolution when they are obtained from the 10-bit ADC for example. In this case the best results are achieved when the value is left-aligned and the LSBs filled with zeroes as this may reduce the number of shifts required to bring the output in range.

4. Filter Implementations

The example filters in this application note were developed and compiled using SDCC version 2.50 and Keil C51 version 9.0.

The filter coefficients were calculated using software made for this purpose. There is a plethora of software that can do this, ranging from costly mathematical programs such as Matlab™, to freely available Java applets on the web. A list of web sites that deal with the topic of calculating filter coefficients are provided in the literature list enclosed last in this 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].

Two filters are implemented: A 23rd order Low Pass (LP) FIR filter, and a 5th order Low Pass (LP) IIR filter. For both implementations, 16-bit signed input samples and 16-bit signed coefficients are used.

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 conditions of the filters are unknown. For both filters, a C code example that initializes and calls the filter function is provided.
Digital Filtering with AT89LP6440

All parameters required for filtering are passed at run time, so the filter function may be reused to implement more than one filter. The implementations focus on fast, programmable execution of the filters, since a high throughput in the filters is of great importance. See Section 5. on page 15 for suggestions on ways to reduce code size and increase throughput further.

There are several items that must be considered when using the example code:

- The compilers treat 16-bit values in different ways. Keil is big-endian and SDCC is little-endian. If a third compiler is used, the starting code base should use the same endianess.
- The interface between the C and Assembly uses the compiler’s default parameter passing scheme. If this scheme is changed or a third compiler is used, the assembly routines must be updated accordingly. Note that Keil and SDCC are different in this regard.
- The SFRs used in the examples apply to AT89LP3240/6440 and may be located at different addresses in other devices.
- At the time of writing, the compilers/assemblers did not support the extended instructions of the Atmel AT89LP family. These instructions are emulated in the assembly code by placing A5H escapes directly in the source. These are wrapped in macros in Keil for better readability.

4.1 23rd Order FIR Filter

For this implementation, a low-pass filter was designed using the Remez application bundled with [10]. The filter parameters are listed in Table 4-1.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Low Pass</td>
</tr>
<tr>
<td>Order</td>
<td>23</td>
</tr>
<tr>
<td>Pass Band</td>
<td>0.0 – 0.06</td>
</tr>
<tr>
<td>Stop Band</td>
<td>0.12 – 1.0</td>
</tr>
<tr>
<td>Scaling</td>
<td>(2^2 (k = 2))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>Value</th>
<th>Scaled</th>
<th>Coefficient</th>
<th>Value</th>
<th>Scaled</th>
</tr>
</thead>
<tbody>
<tr>
<td>b_0</td>
<td>0.0033740915</td>
<td>442</td>
<td>b_12</td>
<td>0.2335460577</td>
<td>30610</td>
</tr>
<tr>
<td>b_1</td>
<td>0.0149382978</td>
<td>1957</td>
<td>b_13</td>
<td>0.1885075162</td>
<td>24707</td>
</tr>
<tr>
<td>b_2</td>
<td>0.0105693581</td>
<td>1385</td>
<td>b_14</td>
<td>0.1154071273</td>
<td>15126</td>
</tr>
<tr>
<td>b_3</td>
<td>0.0025415065</td>
<td>333</td>
<td>b_15</td>
<td>0.0400895415</td>
<td>5254</td>
</tr>
<tr>
<td>b_4</td>
<td>-0.0159299926</td>
<td>-2088</td>
<td>b_16</td>
<td>-0.0146291680</td>
<td>-1918</td>
</tr>
<tr>
<td>b_5</td>
<td>-0.0340853420</td>
<td>-4468</td>
<td>b_17</td>
<td>-0.0381121746</td>
<td>-4996</td>
</tr>
<tr>
<td>b_6</td>
<td>-0.0381121746</td>
<td>-4996</td>
<td>b_18</td>
<td>-0.0340853420</td>
<td>-4468</td>
</tr>
<tr>
<td>b_7</td>
<td>-0.0146291680</td>
<td>-1918</td>
<td>b_19</td>
<td>-0.0159299926</td>
<td>-2088</td>
</tr>
<tr>
<td>b_8</td>
<td>0.0400895415</td>
<td>5254</td>
<td>b_20</td>
<td>0.0025415065</td>
<td>333</td>
</tr>
<tr>
<td>b_9</td>
<td>0.1154071273</td>
<td>15126</td>
<td>b_21</td>
<td>0.0105693581</td>
<td>1385</td>
</tr>
<tr>
<td>b_10</td>
<td>0.1885075162</td>
<td>24707</td>
<td>b_22</td>
<td>0.0149382978</td>
<td>1957</td>
</tr>
<tr>
<td>b_11</td>
<td>0.2335460577</td>
<td>30610</td>
<td>b_23</td>
<td>0.0033740915</td>
<td>442</td>
</tr>
</tbody>
</table>
Figure 4-1. Amplitude Response of FIR Filter

The filter routine is implemented in assembly to make it efficient. However, the filter nodes need to be initialized prior to calling the filtering function. The initialization is done in assembly, but is callable from C. An array containing the filter coefficients (COEFF) and the filter nodes (FIFO) are defined for the filter. COEFF is placed in the code flash memory and FIFO is placed in the internal extra RAM at address 0000H as detailed in Section 3.2 on page 5.

Note the coefficients are placed in the table in reverse order (b_{23} to b_{0}) as the convolution is done in order from oldest to newest value. The final value of the table is the number of shifts required to downscale the result. This value is placed here to avoid passing more than three parameters to the filter function. For big-endian compilers, like Keil, the shift value must be placed in the high byte as this will be the next byte accessed after all the coefficients. (A full 16-bit shift value makes no sense when the accumulator is only 40-bits wide).

The clear_fifo() function must be called first to initialize the FIFO. Then a sample must be placed in the FIFO. The C code uses the get_sample() function to fetch data from a waveform table. In a real application this function would fetch the next sample from the ADC or one of the communication channels like UART or SPI. The store_in_fifo() function is an assembly function that places the sample value into the FIFO and updates the FIFO pointer using the circular buffer. The FIFO pointer (fifoP) always points to the last (oldest) entry in the FIFO.

The filter function is defined as follows:

```c
int compute_fir(int *bufP, int *coeffP, unsigned char n);
```

In the example, the pointers are explicitly located in the XDATA space. The function first copies the pointers into the data pointers. Control bits are set up and then the core of the algorithm is ready to be run. The samples are loaded and multiplied with the corresponding coefficients, then added to the M accumulator. The samples are processed from the oldest to the newest value.

When all samples and coefficients are multiplied-and-accumulated, the result is downscaled. For this example k = 2, but only a single shift is required because of the implicit shift of the binary point due to the multiplication. The truncated result is available in bytes 3 and 4 of the M register.
Digital Filtering with AT89LP6440

(MRW = 10B). An optional rounding step can also be performed by adding a 1 to the digit just to the right of the window (the MSB of byte 2) before returning the result. There is no instruction to add a constant to the M accumulator, so instead a MAC instruction with constant operands is performed to generate the desired offset (in this case 8000H, e.g. 2 x 4000H). These constants are hardcoded in the assembly and would need to be changed for other implementations. It is left as an exercise to the user to either pass them in with the coefficient table or as a separate parameter.

4.1.1 Filter Performance

Table 4-2 shows the performance of the general FIR filter. Byte and cycle counts include all the instructions implemented in the assembly routines, but not the function call overhead from the C code. The difference between Keil and SDCC is due to the different parameter passing schemes and little versus big-endianess. Figure 4-2 shows the input and generated output from the example filter. The performance of the example 23rd order filter is given in Table 4-3.

**Table 4-2.** FIR Performance

<table>
<thead>
<tr>
<th>Function</th>
<th>Keil</th>
<th>SDCC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bytes</td>
<td>Cycles</td>
</tr>
<tr>
<td>clear_fifo()</td>
<td>12</td>
<td>9+11(N+1)</td>
</tr>
<tr>
<td>store_in_fifo()</td>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td>compute_fir()</td>
<td>59</td>
<td>46+5S+29(N+1)</td>
</tr>
<tr>
<td>compute_fir()</td>
<td>72</td>
<td>66+5S+29(N+1)</td>
</tr>
</tbody>
</table>

Notes: 1. No rounding
2. With rounding
3. S = number of shifts; N = filter order

**Figure 4-2.** FIR/IIR Example Input and Output over 64 Samples
The maximum possible sampling rate for a filter of given order can be computed from Equation 8 and the maximum order for a given sampling rate is given by Equation 9. Note that these are theoretical maximums that do not figure in the extra overhead in the C code.

**Equation 8:** Maximum Sampling Rate for Order N (No Rounding)

\[
\text{Keil: } f_S < \frac{f_{CPU}}{71 + 5S + 29(N + 1)} \\
\text{SDCC: } f_S < \frac{f_{CPU}}{62 + 5S + 32(N + 1)}
\]

**Equation 9:** Maximum Order for Sampling Rate \( f_S \) (No Rounding)

\[
\text{Keil: } N < \frac{f_{CPU}}{f_S} \cdot \frac{71 - 5S}{29} - 1 \\
\text{SDCC: } N < \frac{f_{CPU}}{f_S} \cdot \frac{62 - 5S}{32} - 1
\]

### Table 4-3. 23rd Order FIR Performance (N=23; S=1)

<table>
<thead>
<tr>
<th>Function</th>
<th>Cycles</th>
<th>f&lt;sub&gt;CPU&lt;/sub&gt;</th>
<th>Keil</th>
<th>SDCC</th>
<th>f&lt;sub&gt;S&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear_fifo()</td>
<td>273</td>
<td>8 MHz</td>
<td>10 ksps</td>
<td>9.5 ksps</td>
<td></td>
</tr>
<tr>
<td>store_in_fifo()</td>
<td>25</td>
<td>12 MHz</td>
<td>15.5 ksps</td>
<td>14.3 ksps</td>
<td></td>
</tr>
<tr>
<td>compute_fir()</td>
<td>747 (767)</td>
<td>16 MHz</td>
<td>20.7 ksps</td>
<td>19.1 ksps</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>20 MHz</td>
<td>25.9 ksps</td>
<td>23.9 ksps</td>
<td></td>
</tr>
</tbody>
</table>

4.2 Fifth Order IIR Filter

For this implementation, a low pass Elliptical filter was designed using the elf software from [7]. The filter parameters are shown in Table 4-4 and the amplitude response in Figure 4-3.

The filter routine is implemented in assembly to make it efficient. However, the filter nodes need to be initialized prior to calling the filtering function. The initialization is done in assembly, but is callable from C. Two arrays containing the filter coefficients (COEFF_A, COEFF_B) and the filter nodes (FIFO_A, FIFO_B) are defined for the filter. COEFF_A/COEFF_B are placed in the code flash memory and FIFO_A/FIFO_B are placed in the internal extra RAM at address 0000H/0100H as detailed in Section 3.2 on page 5.

Note the coefficients are place in the table in reverse order (\( b_5 \) to \( b_0; a_5 \) to \( a_1 \)) as the convolutions are done in order from oldest to newest value. The final value of the table is the number of shifts required to downscale the result. This value is placed here to avoid passing more than three parameters to the filter functions. For big-endian compilers, like Keil, the shift value must be placed in the high byte as this will be the next byte accessed after all the coefficients. (A full 16-bit shift value makes no sense when the accumulator is only 40-bits wide).

The clear_fifo() function must be called twice first to initialize the FIFO_A and FIFO_B. Then a sample must be placed in FIFO_B. The C code uses the get_sample() function to fetch data.
from a waveform table. In a real application this function would fetch the next sample form the ADC or one of the communication channels like UART or SPI. The store_in_fifo() function is an assembly function that places the sample value into the FIFO_B and updates the FIFO_B pointer using the circular buffer. The FIFO pointers always points to the last (oldest) entry in the FIFO.

The filter functions are defined as follows:

```c
int compute_iir_A(int *bufP, int *coeffP, unsigned char n);
void compute_iir_B(int *bufP, int *coeffP, unsigned char n);
```

In the example, the pointers are explicitly located in the XDATA space. The functions first copy the pointers into the data pointers. Control bits are set up and then the core of the algorithm is ready to be run. The samples are loaded and multiplied with the corresponding coefficients, then added to the M accumulator. The samples are processed from the oldest to the newest value.

The B coefficients and input samples are processed first. When all samples and coefficients are multiplied-and-accumulated, the result is downscaled to the range of A. For this example $k = 9$ and $h = -3$, so $k-h$ gives a total shift of 12. If rounding is enabled, it must occur before the final shift by adding one to the M register.

The A coefficients and output values are processed next without clearing the M register. The final result must be shifted left four times: three because $h = -3$ plus one for the implicit shift of the binary point due to the multiplication. However, the routines do not support left shifting. The same result is done with right shifts by shifting right 4 times, but taking the results from bytes 2 and 3 of the M register (MRW = 01B). An optional rounding step can also be performed by adding a 1 to the digit just to the right of the window (the MSB of byte 1) before returning the result. There is no instruction to add a constant to the M accumulator, so instead a MAC instruction with constant operands is performed to generate the desired offset (in this case 80H, e.g. $1 \times 80H$). These constants are hardcoded in the assembly and would need to be changed for other implementations. It is left as an exercise to the user to either pass them in with the coefficient table or as a separate parameter.

**Table 4-4. IIR Filter Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Low Pass</td>
</tr>
<tr>
<td>Order</td>
<td>5</td>
</tr>
<tr>
<td>Pass Band</td>
<td>0.0 – 0.06</td>
</tr>
<tr>
<td>Stop Band</td>
<td>0.12 – 1.0</td>
</tr>
<tr>
<td>Scaling</td>
<td>$a: 2^{-3}$ ($h = -3$) b: $2^9$ (k = 9)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>Value</th>
<th>Scaled</th>
<th>Coefficient</th>
<th>Value</th>
<th>Scaled</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-a_0$</td>
<td>-1</td>
<td>-4096</td>
<td>$b_0$</td>
<td>0.00094553</td>
<td>15863</td>
</tr>
<tr>
<td>$-a_1$</td>
<td>4.38885</td>
<td>17977</td>
<td>$b_1$</td>
<td>-0.00117032</td>
<td>-19635</td>
</tr>
<tr>
<td>$-a_2$</td>
<td>-7.88315</td>
<td>-32288</td>
<td>$b_2$</td>
<td>0.000875864</td>
<td>14695</td>
</tr>
<tr>
<td>$-a_3$</td>
<td>7.2291</td>
<td>29610</td>
<td>$b_3$</td>
<td>0.000875864</td>
<td>14695</td>
</tr>
<tr>
<td>$-a_4$</td>
<td>-3.3806</td>
<td>-13846</td>
<td>$b_4$</td>
<td>-0.00117032</td>
<td>-19635</td>
</tr>
<tr>
<td>$-a_5$</td>
<td>0.644496</td>
<td>2640</td>
<td>$b_5$</td>
<td>0.00094553</td>
<td>15863</td>
</tr>
</tbody>
</table>
4.2.1 Filter Performance

Table 4-5 shows the performance of the general IIR filter. Byte and cycle counts include all the instructions implemented in the assembly routines, but not the function call overhead from the C code. The difference between Keil and SDCC is due to the different parameter passing schemes and little versus big-endianess. Figure 4-2 shows the input and generated output from the example filter. The performance of the example 5th order filter is given in Table 4-6.

Table 4-5. IIR Performance

<table>
<thead>
<tr>
<th>Function</th>
<th>Keil</th>
<th>SDCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear_fifo()</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>store_in_fifo()</td>
<td>20</td>
<td>14</td>
</tr>
<tr>
<td>compute_iir_A(1)</td>
<td>56</td>
<td>60</td>
</tr>
<tr>
<td>compute_iir_A(2)</td>
<td>69</td>
<td>73</td>
</tr>
<tr>
<td>compute_iir_B(1)</td>
<td>58</td>
<td>60</td>
</tr>
<tr>
<td>compute_iir_B(2)</td>
<td>71</td>
<td>73</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Keil</th>
<th>SDCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear_fifo()</td>
<td>9+11(N+1)</td>
<td>11+11(N+1)</td>
</tr>
<tr>
<td>store_in_fifo()</td>
<td>43+5S+29N</td>
<td>42+5S+32N</td>
</tr>
<tr>
<td>compute_iir_A(1)</td>
<td>63+5S+29N</td>
<td>62+5S+32N</td>
</tr>
<tr>
<td>compute_iir_A(2)</td>
<td>47+5S+29(N+1)</td>
<td>44+5S+32(N+1)</td>
</tr>
<tr>
<td>compute_iir_B(1)</td>
<td>67+5S+29(N+1)</td>
<td>64+5S+32(N+1)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Keil</th>
<th>SDCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear_fifo()</td>
<td>12</td>
<td>14</td>
</tr>
<tr>
<td>store_in_fifo()</td>
<td>20</td>
<td>14</td>
</tr>
<tr>
<td>compute_iir_A(1)</td>
<td>56</td>
<td>60</td>
</tr>
<tr>
<td>compute_iir_A(2)</td>
<td>69</td>
<td>73</td>
</tr>
<tr>
<td>compute_iir_B(1)</td>
<td>58</td>
<td>60</td>
</tr>
<tr>
<td>compute_iir_B(2)</td>
<td>71</td>
<td>73</td>
</tr>
</tbody>
</table>

Notes: 1. No rounding  
2. With rounding  
3. $S =$ number of shifts; $N =$ filter order

The maximum possible sampling rate for a filter of given order can be computed from Equation 10 and the maximum order for a given sampling rate is given by Equation 11. Note that these are theoretical maximums that do not figure in the extra overhead in the C code.
Digital Filtering with AT89LP6440

Equation 10: Maximum Sampling Rate for Order N (No Rounding)

Keil: \[ f_s < \frac{f_{CPU}}{140 + 5S_A + 5S_B + 29(2N + 1)} \]
SDCC: \[ f_s < \frac{f_{CPU}}{124 + 5S_A + 5S_B + 29(2N + 1)} \]

Equation 11: Maximum Order for Sampling Rate \( f_s \) (No Rounding)

Keil: \[ N < \frac{f_{CPU}}{f_s} - 140 - 5S_A - 5S_B \] \[ \frac{58}{5} - 0.5 \]
SDCC: \[ N < \frac{f_{CPU}}{f_s} - 124 - 5S_A - 5S_B \] \[ \frac{64}{6} - 0.5 \]

Table 4-6. 5th Order IIR Performance (N=5; S_A=4; S_B=12)

<table>
<thead>
<tr>
<th>Function</th>
<th>Cycles</th>
<th>( f_{CPU} )</th>
<th>( f_s )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Keil</td>
<td>SDCC</td>
<td>Keil</td>
</tr>
<tr>
<td>clear_fifo()</td>
<td>139</td>
<td>143</td>
<td>8 MHz</td>
</tr>
<tr>
<td>store_in_fifo()</td>
<td>50</td>
<td>38</td>
<td>12 MHz</td>
</tr>
<tr>
<td>compute_iir_A()</td>
<td>208 (228)</td>
<td>222 (242)</td>
<td>16 MHz</td>
</tr>
<tr>
<td>compute_iir_B()</td>
<td>286 (306)</td>
<td>301 (321)</td>
<td>20 MHz</td>
</tr>
</tbody>
</table>

5. Optimization of Filter Implementations

The filters implemented in this application note were made efficient, but still easy to adapt to different filters. Because of this, they are sub-optimal. Below are suggested ways to optimize filters with regards to code size and/or throughput.

5.1 Improving Throughput

One way to improve throughput is to unroll the shift loops and hardcode each shift directly. This removes the loop overhead at the expense of code size and also fixes the filter code to a single filter implementation.

Another method is to place the coefficients in RAM. This removes one cycle from each coefficient byte fetch, but increases the memory usage as the values must be copied from flash into RAM. Be sure to use the internal EDATA (EXRAM = 0) in place of XDATA for all filter memory as external memory incurs additional overhead.

5.2 Reducing Code Size

For FIR filters the coefficients are usually symmetrical. For higher order filters the code can be reduced by only storing half of the coefficient table. However, throughput will be impacted as the filtering routine must detect the half-way point and then scan the table in the opposite direction.

For simplicity the input and output convolution loops for the IIR filter are implemented in separate functions even though they are mostly the same. Code size can be further reduced by combining these functions into a single function.
5.3 Reducing Code Size and Improving Throughput

The easiest way to reduce the code size and improve throughput is to fix the filter implementation. The filter parameters can then be hardcoded directly in the assembly routines without having to pass them in to the function. These include the size of the FIFO and the location of the coefficient table.

In some cases it may also be beneficial to handle the FIFO pointers directly in the assembly routines without interfacing to the C code.

6. References

Disclaimer: The information in this document is provided in connection with Atmel products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Atmel products. EXCEPT AS SET FORTH IN ATMEAL’S TERMS AND CONDITIONS OF SALE LOCATED ON ATMEAL’S WEB SITE, ATMEAL ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL ATMEAL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF PROFITS, BUSINESS INTERRUPTION, OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF ATMEAL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Atmel makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and product descriptions at any time without notice. Atmel does not make any commitment to update the information contained herein. Unless specifically provided otherwise, Atmel products are not suitable for, and shall not be used in, automotive applications. Atmel’s products are not intended, authorized, or warranted for use as components in applications intended to support or sustain life.

© 2011 Atmel Corporation. All rights reserved.

Atmel®, Atmel logo and combinations thereof, and others are registered trademarks or trademarks of Atmel Corporation or its subsidiaries. Other terms and product names may be trademarks of others.

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
Tel: (+1) (408) 441-0311
Fax: (+1) (408) 487-2600
www.atmel.com
8051@atmel.com

Atmel Asia Limited
Unit 01-5 & 16, 19F
BEA Tower, Millennium City 5
418 Kwun Tong Road
Kwun Tong, Kowloon
HONG KONG
Tel: (+852) 2245-6100
Fax: (+852) 2722-1369

Atmel Munich GmbH
Business Campus
Packring 4
D-85748 Garching b. Munich
GERMANY
Tel: (+49) 89-31970-0
Fax: (+49) 89-3194621

Atmel Japan
9F, Tonetsu Shinkawa Bldg.
1-24-8 Shinkawa
Chuo-ku, Tokyo 104-0033
JAPAN
Tel: (+81) (3) 3523-3551
Fax: (+81) (3) 3523-7581

© 2011 Atmel Corporation. All rights reserved.

Atmel®, Atmel logo and combinations thereof, and others are registered trademarks or trademarks of Atmel Corporation or its subsidiaries. Other terms and product names may be trademarks of others.

Disclaimer: The information in this document is provided in connection with Atmel products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Atmel products. EXCEPT AS SET FORTH IN ATMEAL’S TERMS AND CONDITIONS OF SALE LOCATED ON ATMEAL’S WEB SITE, ATMEAL ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL ATMEAL BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF PROFITS, BUSINESS INTERRUPTION, OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF ATMEAL HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Atmel makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and product descriptions at any time without notice. Atmel does not make any commitment to update the information contained herein. Unless specifically provided otherwise, Atmel products are not suitable for, and shall not be used in, automotive applications. Atmel’s products are not intended, authorized, or warranted for use as components in applications intended to support or sustain life.