Audio Filtering with the MAXQ2000

The combination of a multiply-accumulate unit (MAC) and a single-cycle core make the MAXQ2000 a versatile microcontroller (µC). The MAXQ2000 has the performance and I/O peripherals ideal for many applications: alarm clocks, handheld medical devices, digital readouts—any application that requires low power, high performance, and numerous I/Os. With the integrated MAC, the MAXQ2000 ventures into the territory of the DSP (µC).

Just how much performance can the MAXQ2000 get out of its MAC? This application note explores that question with an audio-filtering example, and gives some quantitative guidelines on the performance supported by the MAXQ2000.

Software and Hardware Requirements

This application note features a simple demonstration of an audio filter. The audio data is a prerecorded message of the author saying "The pipe began to rust while new." This text was not selected randomly—it provides a decent mix of frequency components, accentuating the audible effects of simple filters (http://www.cs.columbia.edu/~hgs/audio/harvard.html). The audio recording can be replaced by any 8kHz recording of appropriate length, but it is not required.

The hardware required for this application note includes the MAXQ2000 evaluation kit and a small circuit to interface to a computer speaker.

The MAXQ2000 evaluation kit which is available, is a good tool for exploring the MAXQ2000's capabilities. It includes an LCD panel, LED bank, and access to all the I/O pins of the MAXQ2000 µC. It also includes a MAX1407 ADC/DAC, which can be used for audio output.

The second piece of hardware required can be easily breadboarded. The circuit that was used for this demonstration is shown in **Figure 1**. It calls for a 1 x 8 female header to connect to the MAXQ2000 evaluation kit board at J7, and another connection to any ground (TP1 on the MAXQ2000 evaluation kit is a good choice). The speaker connector can be of any type—shown is a 3.5mm stereo jack, which makes it simple to connect to typical computer speakers. Note that the two input channels are tied together, as our demonstration application is only showing one audio channel (mono sound).

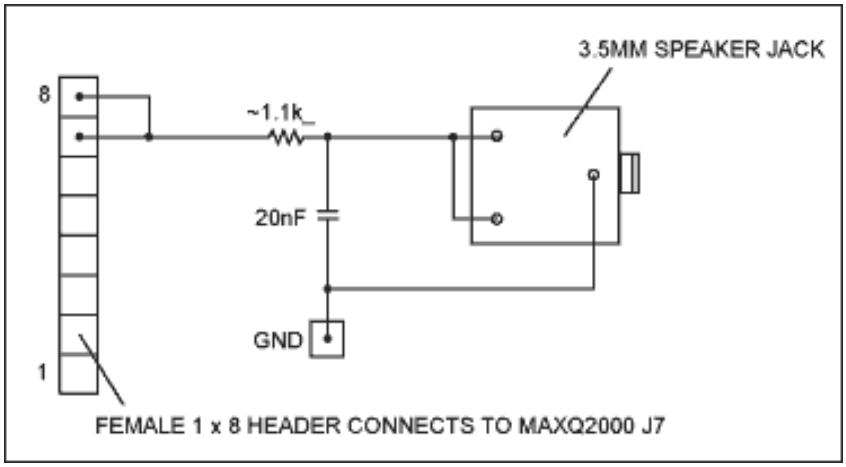

Figure 1. Additional Hardware Required for Audio Playback

The software required to run this demonstration was built and debugged using the IAR embedded workbench. It provides a good debugging environment, making use of the hardware debugging support of the MAXQ2000. You can set breakpoints, set and read registers and memory, and see your call stack while running on the real hardware.

Running the Demo Application

The pushbuttons on the MAXQ2000 evaluation kit board are used to select a filter, and to play the audio sample passed through that filter. Use button SW4 to select a filter—the name of the filter will be displayed on the LCD screen (HI for high pass, LO for low pass, BP for band pass, ALL for all pass). Use the SW5 button to begin playing the audio through the selected filter. The filter can be changed during the middle of playback.

Designing a Simple FIR Filter

I developed a Java™applet that would let me easily create new filters. Rather than use standard windowing techniques given filter parameters, I chose to 'design' my filter grossly by placing zeroes in a pole-zero plot, as shown in **Figure 2**. The applet allows placing zeroes anywhere in the coordinate plane, and continually updates the FIR filter coefficients required by the demonstration application. Note, however, that the demonstration only supports all-zero filters. Supporting IIR filters would not be too difficult—more is explained in the *Supporting IIR Filters* section.

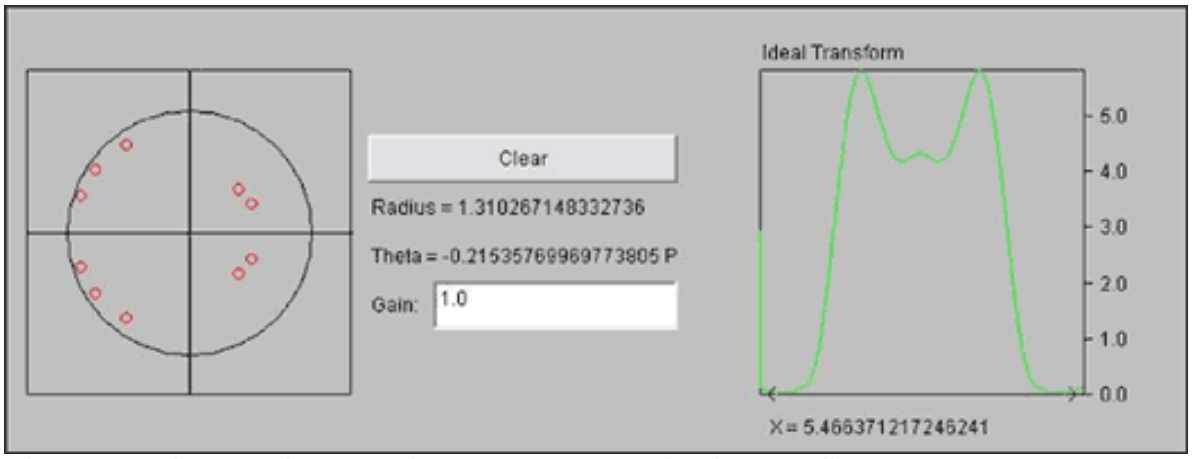

Figure 2. Using a Pole-Zero Plot to Generate a Simple FIR Filter

A generic filter takes the form of a linear equation:

$$y(n) + \Sigma b_K y(k) = \Sigma a_J x(j)$$

where k represents the order of the feedback portion of the filter and j represents the order of the feed-forward portion of the filter.

An example IIR filter could be something as simple as:

$$y(n) = 0.5y(n-1) + x(n) - 0.8x(n-1)$$

Some filters are classified as FIR filters. They do not contain the feedback portion. In other words, there is no y portion in the characteristic filter equation:

$$y(n) = \Sigma a_J x(j)$$
$$y(n) = x(n) - 0.2x(n - 1) + 0.035x(n - 3)$$

In either case, a filter boils down to a characteristic equation that is essentially a weighted average of some number of the past input and output values. The job of filter design is to produce those A_j and B_k values. In order to compute the output of a filter efficiently, we need hardware support that can multiply and sum signed numbers quickly. Enter the MAXQ2000's multiply-accumulate unit.

Implementing a Filter with a Multiply-Accumulate (MAC) Unit

The applet from the previous section works by computing the filter coefficients given the zero coordinates in the plot. However, the coefficients computed are floating-point numbers, while our MAC works with purely 16-bit integer math. In order to correct the problem, the demo application uses a fixed-point number system, where between 0 and 15 bits of the coefficients are to the right of the decimal point (the 16th bit represents the sign magnitude). Once the operation is over, the 48-bit result in the MAC's accumulator is shifted enough places to remove any fraction.

This solution is a tradeoff of accuracy for speed. In many cases, the error from this approach is negligible. For diagnostic purposes, the applet shows three plots of the calculated filter. The first plot shows the ideal filter behavior, using 64-bit floating-point numbers. This plot, labeled "Ideal Transform", is shown in Figure 2.

Figure 3 illustrates the remaining plots produced by the applet. The first plot in Figure 3 shows the effective filter using 16-bit, fixed-point numbers. In many cases, the error is not apparent, so the last plot is an error indicator, showing the ideal behavior divided by the actual frequency response. Ideally, this is a straight line at Y = 1.

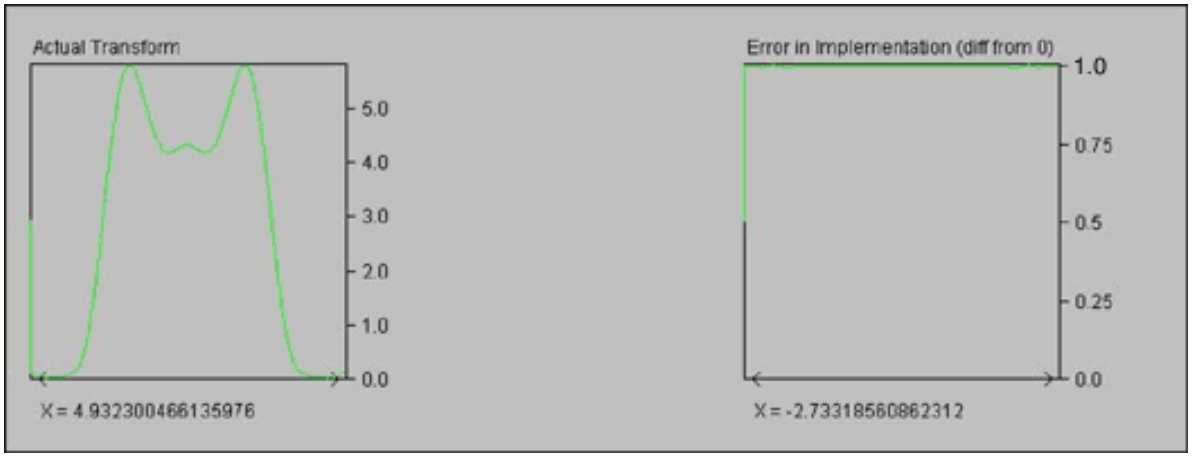

Figure 3. Actual Transform and Round-Off Error (with Virtually No Error) for a 16-Bit Implementation of the Filter

For simplicity, the applet produces the floating-point coefficients required by the MAXQ application, so that new filters can simply be cut-and-pasted into the source for the filter application (into the file data.asm). The applet also produces two other values—the order of the filter (the number of coefficients) and the shift count, so the application can shift the final result appropriately. This data appears in the text box at the bottom of the applet, and might look like this:

```
Zeroes:
    dc16
    dc16 12, 11, 0x1000, 0x26d3, 0x1e42, 0xf9a3, 0xecde, 0xff31, 0xa94,
         0x2ae, 0xfd0c, 0xff42, 0xde
Shift amount: 12
```

Implementing a Filter in MAXQ Assembly Language

In order to get maximum performance and perform an accurate performance analysis, the actual filter will be implemented in assembly language. This will allow us to accurately count the number of cycles required to produce one output value, and thus estimate the performance for other data sets.

The MAX1407 has a 12-bit ADC. However, the input data is 16-bits wide, and our filter produces a 16-bit result. So, while those 4 least significant bits (LSBs) are wasted for this application, we can safely analyze our performance as if working on and producing 16-bit values (CD-quality audio is 16 bits).

In this example, the filter coefficients are stored in code space in a table. When a filter is selected, the application finds the appropriate filter, reads the shift amount and number of taps, and is then ready to start filtering data. The following code applies the filter coefficients:

```
    move  MCNT, #22h          ; signed, mult-accum, clear regs first

zeroes_filterloop:
    move  A[0], DP[0]         ; let's see if we are out of data
    cmp   #W:rawaudiodata     ; compare to the start of the audio data
    lcall UROM_MOVEDP1INC     ; get next filter coefficient
    move  MA, GR              ; multiply filter coefficient...
    lcall UROM_MOVEDP0DEC     ; get next filter data
    move  MB, GR              ; multiply audio sample...
    jump  e, zeroes_outofdata ; stop if at the start of the audio data
    djnz  LC[0], zeroes_filterloop

zeroes_outofdata:
    move  A[2], MC2           ; get MAC result HIGH
    move  A[1], MC1           ; get MAC result MID
    move  A[0], MC0           ; get MAC result LOW
```

Before this code is executed, LC[0] is set up with the number of taps for the filter, DP[0] is set up with the address of the current input byte to the filter, and DP[1] points to the start of the filter coefficients. Therefore, DP[1] works through the filter coefficients in an incrementing manner, and DP[0] works through the input data in a decrementing manner (processing the most recent input first).

Since the MAC works in a single cycle, there is not a lot of code here to handle it. MCNT is set to 22h to denote the use of signed integers. In the main loop, the consecutive writes to MA, then MB triggers the multiply-accumulate operation—the result is ready in the next clock cycle. Since our accumulator is 48 bits (and our multiply result is 32 bits), we need not worry about any overflow (unless we have 64,000 taps in our filter!)

Performance

The sample application works on mono 16-bit audio data being output at 8kHz—not nearly enough to tire out the µC. Because we wrote the filter in assembly language, we can easily count the cycles being used to come up with an expression for the amount of time a FIR filter of length N takes to compute. We can then use this expression to find the maximum filtering rate using the algorithm previously listed.

We can split the function we use to produce an audio sample into three portions: initialization, filter computation loop, and result fixing. In our released example, the initialization takes 38 cycles, the filter computation loop takes 17 cycles per filter coefficient, and the result fixing takes 9 + (6 x S) cycles, where S is the shift amount. Normally, the shift amount is around 12, so we can estimate result fixing at 81 cycles. Therefore, it takes 119 + (17 x N) cycles to produce one filtered output value. At 20MHz, a MAXQ2000 could run a 100-tap filter at close to 11kHz, which is good enough quality for voice data.

Let us go back and reanalyze our application to see where we could tighten it up. We will concentrate in the filter loop, since this is where most of our cycles are happening on any but the most trivial of filters.

There are a few critical improvements we can make to the loop code to improve efficiency. Remember that we are

using prerecorded audio samples that are stored in code space. Lookups to code space require more time than lookups in data space because of the MAXQ's Harvard architecture. The functions called UROM_MOVEDP1INC and UROM_MOVEDP0DEC take 5 cycles each (2 cycles for the LCALL, then 3 cycles inside the function). These could each be replaced by two cycles each if we stored our filter in the RAM (one cycle to select the pointer, one to read from it), and if we were serving real-time input data that was stored in the RAM. If we were willing to donate 256 words of RAM to the filter, we could implement a circular buffer using BP[Offs] to store the input data. These changes reduce the loop time to 11 cycles instead of 17. Our filter loop now looks like this (cycle counts listed first in the comments):

```
zeroes_filterloop:
    move  A[0], DP[0]          ; 1, let's see if we are out of data
    cmp   #W:rawaudiodata      ; 2, compare to the start of the audio data
    move  DP[1], DP[1]         ; 1, select DP[1] as our active pointer
    move  GR, @DP[1]++         ; 1, get next filter coefficient
    move  MA, GR               ; 1, multiply filter coefficient...
    move  BP, BP               ; 1, select BP[Offs] as our active pointer
    move  GR, @BP[Offs--]      ; 1, get next filter data
    move  MB, GR               ; 1, multiply audio sample...
    jump  e, zeroes_outofdata  ; 1, stop if at the start of the audio data
    djnz  LC[0], zeroes_filterloop  ; 1
```

Once we have our filter and input data in the RAM, we can use another trick of the MAXQ architecture. The MAXQ instruction set is highly orthogonal—there are very few restrictions on what can be used as a source in any operation. Therefore, rather than reading the filter data and the input data into GR, we can write it directly to the MAC registers. This brings the loop down to 9 cycles.

```
zeroes_filterloop:
    move  A[0], DP[0]          ; 1, let's see if we are out of data
    cmp   #W:rawaudiodata      ; 2, compare to the start of the audio data
    move  DP[1], DP[1]         ; 1, select DP[1] as our active pointer
    move  MA, @DP[1]++         ; 1, multiply next filter coefficient
    move  BP, BP               ; 1, select BP[Offs] as our active pointer
    move  MB, @BP[Offs--]      ; 1, multiply next filter data
    jump  e, zeroes_outofdata  ; 1, stop if at the start of the audio data
    djnz  LC[0], zeroes_filterloop  ; 1
```

One final improvement can make this code really fly. Every time through the loop, we compare our current data pointer to the start of the audio-input data to see if we are going out of bounds (the MOVE A[0], DP[0] statement, the CMP compare statement, and the JUMP E statement). If we set the initial audio data (which we are now reading with the circular buffer pointed to by BP[Offs]) to all zeroes, we can simply remove these checks. The cost of initializing the RAM to 0 is negligible, as compared to the 4-cycle savings over the next several thousand samples. Our new loop code is a slim 5 cycles.

```
zeroes_filterloop:
    move  DP[1], DP[1]         ; 1, select DP[1] as our active pointer
    move  MA, @DP[1]++         ; 1, multiply next filter coefficient
    move  BP, BP               ; 1, select BP[Offs] as our active pointer
    move  MB, @BP[Offs--]      ; 1, multiply next filter data
    djnz  LC[0], zeroes_filterloop  ; 1
```

Before going back to our performance equation, let us look at the result computation. Our current way of shifting the 48-bit result down seems wasteful.

```
    move  A[2], MC2           ; get MAC result HIGH
    move  A[1], MC1           ; get MAC result MID
    move  A[0], MC0           ; get MAC result LOW
    move  APC, #0C2h          ; clear AP, roll modulo 4, auto-dec AP

shift_loop:
    ;
    ; Because we use fixed point precision, we need to shift to get a real
    ; sample value.  This is not as efficient as it could be.  If we had a
    ; dedicated filter, we might make use of the shift-by-2 and shift-by-4
    ; instructions available on MAXQ.
    ;
    move  AP, #2              ; select HIGH MAC result
    move  c, #0               ; clear carry
    rrc                       ; shift HIGH MAC result
    rrc                       ; shift MID MAC result
    rrc                       ; shift LOW MAC result
    djnz  LC[1], shift_loop   ; shift to get result in A[0]
    move APC, #0              ; restore accumulator normalcy
    move AP, #0               ; use accumulator 0
```

One possible solution is to use our MAC once again. Rather than shifting right by 12 (or any value between 0 and 16), we can shift left by 16 minus that amount (i.e. shift left by 4). This will put our result in the middle 16-bit word of the MAC's registers. Note that our shift left will actually be accomplished with a multiply-by-2 to some power (2^4, in the case where our original shift right was supposed to be 12).

```
    ;
    ; don't care about high word, since we shift left and take the
    ; middle word.
    ;
    move  A[1], MC1           ; 1, get MAC result MID
    move  A[0], MC0           ; 1, get MAC result LOW
    move  MCNT, #20h          ; 1, clear the MAC, multiply mode only
    move  AP, #0              ; 1, use accumulator 0
    and   #0F000h             ; 2, only want the top 4 bits
    move  MA, A[0]            ; 1, lower word first
    move  MB, #10h            ; 1, multiply by 2^4
    move  A[0], MC1R          ; 1, get the high word, only lowest 4 bits significant
    move  MA, A[1]            ; 1, now the upper word, we want lowest 12 bits
    move  MB, #10h            ; 1, multiply by 2^4
    or    MC1R                ; 1, combine the previous result and this one
    ;
    ; result is in A[0]
    ;
```

This will let us take our result computation to 12 cycles, rather than 9 + (6 x S) cycles.

Now let us plug back into our earlier equation. Our new equation uses a conservative estimate of 40 cycles of overhead and 5 cycles per loop iteration. Using the same example as earlier of a 100-tap filter, the MAXQ2000 could process 16-bit, mono audio data at 37kHz, as seen in **Table 1**.

Table 1. Maximum FIR Filter Sample Rates (20MHz MAXQ2000, Looping)

Filter Length (Taps)	Max Rate (Hz)
50	68965.51724
100	37037.03704
150	25316.4557
200	19230.76923
250	15503.87597
300	12987.01299
350	11173.18436

There is one more performance improvement that we can implement for applications in which a higher sampling rate is required and code space can be sacrificed. We can 'in-line' the filter coefficients, which eliminates both the need to select active pointers and the need to loop (this technique is also known as loop unrolling). The price of this change is increased code space—previously, our 100-point filter took 100 words to store; now it will take 300 words to store (2 words for each coefficient move, 1 word for each data value move). In a 16 kilo-word device, this may be an insignificant price for the performance benefit. The new code might look something like this:

```
    move  BP, BP              ; select BP[Offs] as our active pointer
zeroes_filtertop:
    move  MA, #FILTERCOEFF_0  ; 2, multiply next filter coefficient
    move  MB, @BP[Offs--]     ; 1, multiply next filter data
    move  MA, #FILTERCOEFF_1  ; 2, multiply next filter coefficient
    move  MB, @BP[Offs--]     ; 1, multiply next filter data
    move  MA, #FILTERCOEFF_2  ; 2, multiply next filter coefficient
    move  MB, @BP[Offs--]     ; 1, multiply next filter data
    . . .
    move  MA, #FILTERCOEFF_N  ; 2, multiply next filter coefficient
    move  MB, @BP[Offs--]     ; 1, multiply next filter data
    ;
    ; filter calculation complete
    ;
```

To calculate the performance benefit of this change, we again assume 40 cycles of overhead, but now there are 3 cycles per loop iteration, though we have truly eliminated the loop. The 100-tap performance limit now sits at 58kHz (see **Table 2**).

Table 2. Maximum FIR Filter Sample Rates (20MHz MAXQ2000, Unrolled Loop)

Filter Length (Taps)	Max Rate (Hz)
50	105263.1579
100	58823.52941
150	40816.32653
200	31250
250	25316.4557
300	31250
350	27027.02703

Supporting IIR Filters

Though this application note does not demonstrate the use of IIR filters, there is no reason the MAXQ2000 cannot

support them. The changes involved would be to:

- Dedicate a segment of RAM to store the latest output samples (this would be most efficiently implemented as a circular buffer, using the BP[Offs] register in a manner similar to that described previously)
- Include the characteristic filter coefficients for the feedback (the 'y' portion) of the filter
- Add another loop that continues to accumulate products that are the result of the feedback portion of the filter

While adding another loop may sound like a performance hit, it might not necessarily be. Although it will require more time to compute one output of the filter, IIR filters often require fewer taps (a smaller value for N) to compute an output value.

Conclusion

The MAXQ2000's performance and peripherals make it a great, general-purpose µC. It can be used anywhere a fast, versatile µC is required, especially in applications requiring user interaction. The effective use of the MAC gives the MAXQ2000 some digital-filtering capability, making the MAXQ2000 one of the most versatile µCs around.

Relevant Links

- MAXQ Home Page
- MAXQ Family User's Guide
- MAXQ2000 User's Guide Supplement
- Dallas Semiconductor Microcontroller Support Forum
- Source code and support applications used in this application note

More Information

MAX1407:	QuickView	-- Full (PDF) Data Sheet	-- Free Samples
MAXQ2000:	QuickView	-- Full (PDF) Data Sheet	-- Free Samples
MAXQ2000-KIT:	QuickView	-- Full (PDF) Data Sheet	