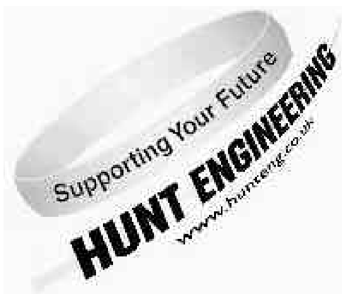


HUNT ENGINEERING
Chestnut Court, Burton Row,
Brent Knoll, Somerset, TA9 4BP, UK
Tel: (+44) (0)1278 760188,
Fax: (+44) (0)1278 760199,
Email: sales@hunteng.demon.co.uk
URL: http://www.hunteng.co.uk

Digital Filters Using the TMS320C6000

Abstract

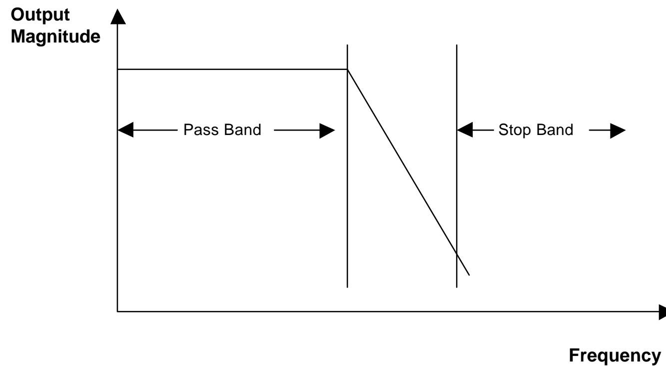
Filtering is one of the basic building blocks of signal processing. It is used everywhere; we use many of the concepts intuitively in daily life, while every electrical appliance we ever use employs filtering - perhaps to remove noise from the power supply of a washing machine, to select the station on a radio, or to detect the DTMF tones of a telephone.

This document describes the basic technology behind digital filters, and shows how we can build filters on the TMS320C6000.

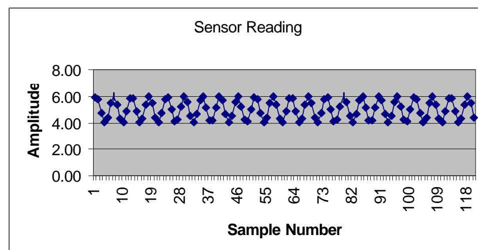
The FIR Filter

Concepts of the FIR

A filter allows us to remove frequencies within a signal in which we are not interested; or to boost ones in which we are interested. Their performance is often drawn as "gain versus frequency", as shown below:

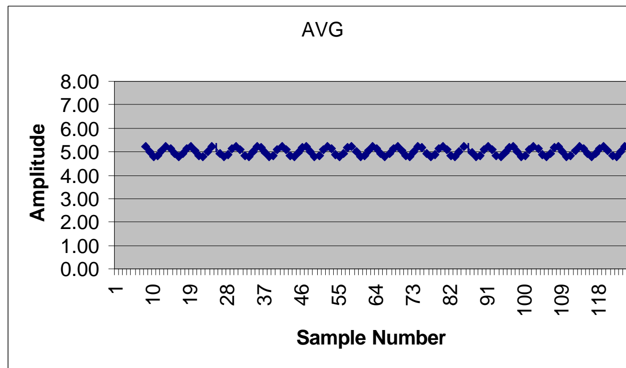


As an example of the use of filters, you may be interested in the level of fuel in your car's fuel tank. The actual level changes slowly; but as you drive, the fuel moves from side to side, and the sensor sees "waves":



The real signal we are interested in is the level, which changes slowly; the waves represent "noise" on that signal. To make any meaningful measurement of the fuel level you have to remove those waves.

Let us assume we sample the level of the fuel tank once per second; and that we are only interested in changes in the fuel level which are sustained for several seconds. If we took several readings a second apart and averaged them we would reduce the noise:



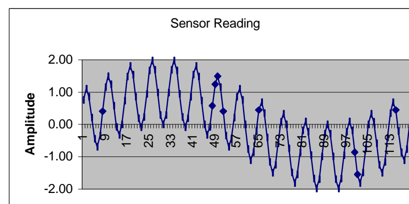
Tank Level	=	0.125 *Reading (n)
	+	0.125 *Reading (n-1)
	+	0.125 * Reading (n-2)
	+	0.125 * Reading (n-3)
	+	0.125 * Reading (n-4)
	+	0.125 * Reading (n-5)
	+	0.125 * Reading (n-6)
	+	0.125 * Reading (n-7)

This is the basis of the FIR (Finite Impulse Response) filter. In our example, we used eight data samples, and multiplied each one by a ***coefficient*** before summing them to form the output.

This is what would be termed an ***8-tap FIR*** filter (sometimes an 8-coefficient or 8-point filter). We have used a very simple example; it is very rare for the coefficients to be the same, but by doing this we have created a Low Pass Filter (LPF).

It should be obvious that averaging will remove low frequencies; but how do we remove high frequencies? The answer lies in the coefficients. By altering them, we can make the filter respond in different ways to different frequencies.

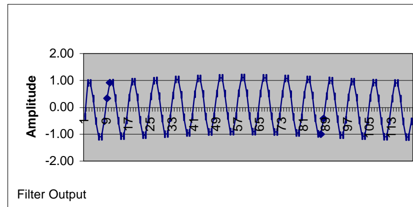
Imagine now that we have an audio signal and we want to remove noise from it. The noise is from the mains power supply, so is relatively low frequency at 50 or 60Hz.



How could we remove this low frequency noise? Let us try altering the coefficients of our LPF to see if we can make it into a High Pass Filter (HPF). We will also use fewer samples to make the maths easier!

Audio	=	1	* Reading (n)
	+	-1	* Reading (n-1)
	+	1	* Reading (n-2)
	+	-1	* Reading (n-3)

Once again, we have used extremely simple coefficients to show the principles. By subtracting the signal from itself we remove DC values; and by repeating this sequence we will greatly reduce the low frequency noise.



Designing FIRs

Plainly the FIR filter is not difficult to understand. We take a set of samples a fixed time apart, and multiply them by a set of coefficients. This has an effect on the signal; by varying the coefficients we can choose what the filter does.

The combination of the length of the filter (number of taps) and the values of the coefficients determine the filter's operation. Designing the filter is just a case of deciding how many taps and choosing the coefficients.

There are many techniques for selecting coefficients. If you are good at maths you could use a spreadsheet like Microsoft Excel, or there are many design packages which will do the job for you.

One such package is ScopeFIR from IOWEGIAN. A trial version is downloadable from www.iowegian.com, although there are many others.

Features of FIRs

The FIR is very intuitive. It is easy to see how it would work, and it has several really attractive features – here is a snapshot:

- Like all digital filters, it is ***not affected by temperature, time or humidity.*** A digital filter will always perform the way it was designed…
- The FIR generally has ***"linear phase"***. A signal passing through the filter will be delayed by a fixed time period, so the relationship between a high frequency and low frequency passing through the filter stays the same. This isn't always the case with filters…
- The FIR is ***"inherently stable"***. Analogue filters (and IIR filters, described later) are very similar to oscillators. Get your design wrong, and the filter may oscillate… this cannot happen with an FIR filter.

These features make the FIR a popular choice. However, it has some downsides; notably, the "roll-off" rate of the filter.

The idea of a filter is that it allows some frequencies to pass, and stops other frequencies. For example, we might want a filter that passes all frequencies up to 1KHz, and stops all frequencies above 1KHz.

A filter that achieves this is sometimes known as a "brick wall filter". Unfortunately they cannot be created; there is always a transition zone between the "pass-band" and the "stop band".

The more information the filter has about the signal, the steeper the transition between pass-band and stop-band can be. This drives the FIR filter to ever-larger filters, increasing both computing time and memory requirements.

The IIR Filter

1. Concepts of the IIR

Let us go back to our car fuel tank. We used an FIR filter to remove the noise; but with only eight taps, and a very simple set of coefficients, the filter has a limited effect. It reduced the noise but did not eliminate it.

If we want to improve the filter's performance we would first select a better set of coefficients using one of the design tools we mentioned! After that, we have two options:

1. Increase the number of taps.
2. Use a different technique.

Increasing the number of taps increases the amount of signal the filter can "see", so would significantly improve its performance. However, this comes at the expense of increased storage and computation.

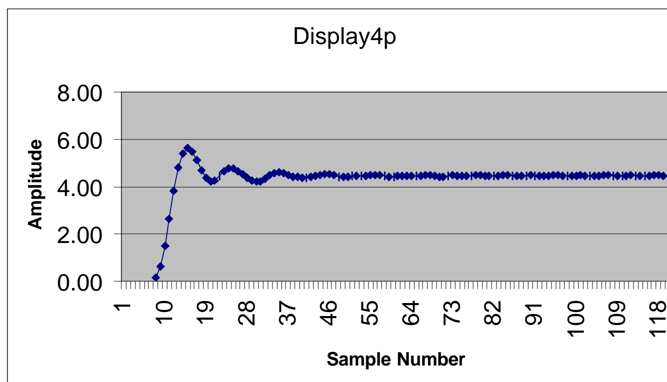
There is another technique we could use. If we were to include the ***current output*** of the filter in our calculations, we would have more information about the signal. More than that, the new output would now include a proportion of the previous output. The previous output would contain part of the output before… and so on back to the point when the filter started.

Thus, by feeding the output back into the filter, we are supplying the filter with a history of the signal from the point the filter started. This can achieve spectacular results, and is the reason for the name "Infinite Impulse Response". If the input of an IIR filter is fed with an "impulse" (a signal spike) and then with zero, the output will always contain some element of that impulse.

Let us go back to the fuel tank. If we use the same amount of maths, but include some of the filter output, we could use an equation like this:

Tank Level	=	a0 * Reading (n)
	+	a1 * Reading (n-1)
	+	a2 * Reading (n-2)
	+	a3 * Reading (n-3)
	+	a4 * Reading (n-4)
	-	b1 * Output (n-1)
	-	b2 * Output (n-2)
	-	b3 * Output (n-3)
	-	b4 * Output (n-4)

This equation represents an IIR (Infinite Impulse Response) filter. It's "fourth order" because there are four pairs of coefficients (a1 & b1, a2&b2…)



The output shows one of the oddities of the IIR - it takes a few moments to settle down after initialisation. However, once settled, the output is clean - there is little or no trace of the waves that caused the noise.

As with the FIR, the coefficients are they key to the IIR. There are many design packages around, although they are more complex so fewer are free!

A common approach is to translate analogue filters (such as a Butterworth) to digital. This approach yields filters which are loosely modelled on classical filters like the Butterworth, Bessel or Elliptical filter.

IIR Terminology

Where FIR filters are defined by the number of taps or coefficients they have, IIR filters are defined by the number of stages.

It is common to talk about an IIR as being of a specific order - such as "2nd order" or "4th order". Generally, the higher the order of the filter, the more spectacular its effects.

"Order" is a term from the maths behind the IIR filter. The maths behind DSP is usually expressed in the "Z-domain" - this is the digital equivalent of the s-domain used to describe analog systems.

The filter we just described is mathematically:

Output (n) = a0 + a1*

This is a ratio of two fourth-order polynomials. There are two z-domain equations in the fraction. Either can become zero…

If the numerator (top line of the fraction) becomes zero, the output goes to zero. This, unsurprisingly, is called a ***"zero"*** for the filter.

If the denominator (bottom line of the filter) becomes zero, the fraction becomes a "divide by zero" and the output heads for infinity… this is known as a ***"pole"*** of the filter.

The filter's poles are crucial to IIR design. Put them in the wrong place and the filter is unstable. It may simply saturate, driving the output to the maximum or minimum value; or it may oscillate. However, the more stable the filter is, the less spectacular its effects - the most efficient design is where the filter is close to becoming unstable. This makes designing IIR filters an interesting mathematical challenge!

Tracking the poles and zeroes can get complex for higher order filters. In order to keep it simple, many developers and design tools will implement the IIR as a ratio of "biquadratic equations". This gives the familiar term "IIR biquad" - a 2nd order IIR filter, several of which may be combined to achieve the effect of a higher order filter.

Features of the IIR

IIR filters share the benefits of digital processing that the FIR had. However they have several more features which are worth mentioning:

- Spectacular performance. An IIR can create far sharper filters than FIR can, given the same amount of computing power. It still can't create a "perfect" filter though!
- Phase delay. An IIR filter can be designed to have specific phase characteristics - including non-linear phase. This can be useful in many applications; but note that achieving linear phase can be hard - if you need linear phase you want to use an FIR.
- Stability. The IIR is not inherently stable. It has a feedback loop which means it can oscillate. This is generally caused by badly chosen coefficients.

The latter point is important. If you use a filter design tool you shouldn't experience this. However, moving between maths formats can change the coefficients enough to make a filter oscillate - most design packages will let you check this.

Implementing Filters on the TMS320C6000 Family

The TMS320C6000 processors are good at filtering, having been designed for the types of operation common in signal processing systems. This section details how these filters could be implemented, and gives benchmarks for the performance of those filters.

Per-Sample or Block Processing?

In signal processing systems, samples are taken in a continuous stream - typically from a sensor operating in real time. Sometimes it is important that the output is calculated as quickly as possible; in other cases, the output can be delayed, providing the system guarantees that there will be an output within a guaranteed time period.

As an example, a system calculating control information will generally calculate an output as soon as the input data is available, and may well output that data before the next input sample is captured. This allows the best possible control of a high-speed system. This is known as ***"per-sample processing".***

However, another system may gather a large set of data before even starting to process it. Then, when it has this data, the processing can be performed in a large block. (The processor may be capturing another block of data while this block is being processed.) This is known as ***"block processing"***, and may be more efficient.

Block processing implementations in software can reduce the input of the overheads setting up the data for processing. However, the FIR and IIR filters are very efficient on the ‘C6000 so block processing is not required.

FIR Filters

To calculate each output of the FIR filter, we multiply a set of samples by a set of coefficients. This is simple to express in a high level language such as C:

```
Output = 0;
For(n=0;n<taps;n++)

{

Output = Output + sample[n] * coefficient[n];

}
```

When a new sample arrives, it is added to the sample set, and the oldest sample is disposed of. This can be performed using the circular addressing hardware of the 'C6000 family, but cannot be expressed in C.

The processor's dual data paths can be used to process two samples per processor cycle; and the efficient looping can mean that for long FIR filters, performance approaches 2 taps/cycle.

To achieve these speeds "intrinsic" operators must be used in the C code. This allows the developer to pass additional control information to the compiler; an alternative is to use a C-callable library function.

For the general case with N taps, the C6000's performance can be summarised thus:

	Theoretical maximum	16 bit integer data, C code with intrinsics	16 bit integer data, hand-coded assembly	32 bit floating point data, hand-coded assembly
FIR (N taps)	0.5*N cycles	0.5*N + 10 cycles	0.5*N + 4 cycles	0.5*N+5 cycles

*Figure 1. Performance of an N tap FIR code, as measured on a 'C6201 (for the two 16 bit integer benchmarks) and a 'C6701 (for the floating-point benchmark). For example, to calculate the number of cycles consumed by an 16 tap FIR filter working on 16 bit integer data: the number of cycles is (0.5*16+4) cycles = 12 cycles. For a 200 MHz 'C6201 one cycle is 5 ns, so our 16 tap FIR filter would consume (12*5=) 60 ns. Both the C code and the assembly routines require a number of taps that is a multiple of 4.* ***Please note that these values are valid only if both code and data are placed in on-chip memory. If code or data is placed in external memory, performance is much lower.***

IIR Filters

The IIR filter is more complex than the FIR to implement - indeed it resembles two FIR filters, one for the forward path (the a0 / a1 /… coefficients) and one for the feedback path (b0 / b1 / b2… coefficients).

Most implementations treat the IIR as a single block rather than as a pair of FIRs. Generally the code does not use looping, as the "order" of an IIR filter is usually low.

The most efficient way of implementing the IIR is through a library function or assembly code. This gives the greatest control over the ordering of the maths operations, which can have an effect on the performance of the filter.
Using library functions, the following results can be achieved for filters with an *even* order of *four or more*; note that the larger filters are more efficient due to the reduced overhead. Performance is shown in the table below.

	Theoretical maximum	16 bit integer data, optimised C code	16 bit integer data, C code + intrinsics	32 bit floating point data, hand-optimised assembly
IIR (Nth order)	N cycles	2.25*N+36 cycles	1.25*N+21 cycles	N+10 cycles

*Figure 2. Performance of an Nth order IIR filter, as measured on a 'C6201 (for the two 16 bit integer benchmarks) and a 'C6701 (for the floating-point benchmark). For example, to calculate the number of cycles consumed by an 8th order IIR filter working on 16 bit integer data: the number of cycles is (1.25*8+21) cycles = 31 cycles. For a 200 MHz 'C6201 one cycle is 5 ns, so our 8th order IIR filter would consume (31*5=) 155 ns. A factor of 2.25 or 1.25 may seem strange, but is the result of memory bank conflicts that happen once in 4 kernel loops, resulting in an average extra factor of 1 cycle/4 = 0.25. In a to-be-developed library function we hope to eliminate this factor.* ***Please note that these values are valid only if both code and data are placed in on-chip memory. If code or data is placed in external memory, performance is much lower.***

Summary

Implementing filters on the TMS320C6000 is straightforward. They can usually be implemented using C code, and the basic building blocks are available as assembly-coded libraries for maximum efficiency.

The 'C6000 also gives very good performance in filter codes. For the FIR, the dual data paths can be fully utilised; while the more complex IIR filter is also implemented efficiently.

Calculating coefficients for the filters is more complex. However, there are design packages available which will do this automatically, requiring only a broad filter design specification from the user.