The Filter Wizard
issue 25: Sample Multiple Channels 'Simultaneously' With A Single ADC
Kendall Castor-Perry

Despite its title, this is indeed an article about a filtering technique. Using it, you can sample a number of analogue input channels sequentially with a single ADC in such a way that the data appears to have been acquired at the same instant in time on every channel. It sounds like magic – but it's just Wizardry!

So: sequential sampling, what's the problem? After all, if you've got, let us say, four channels that you need to sample at some rate Fs, and an ADC that'll sample fully at 4Fs and that can be driven through a 4-way multiplexer, then isn't that a nice economical way of getting the job done?

Here's the catch. If you need to do any sample-by-sample processing involving data from more than one channel, you're going to have to 'pretend' that readings you took from different channels at different times actually happened at the **same** time, so that you can combine them in a calculation.

Say you want to calculate the instantaneous energy being absorbed in an AC power circuit. You sample the voltage and the current, and multiply those readings together. The result gives you the energy, when multiplied by the length of time you assumed the samples were valid over – essentially, the sample period. Integrate these readings over time and you accumulate the total energy. Hey Presto, it's an electricity meter!

But! If you use a multiplexed ADC, you didn't take the voltage and the current readings at the same time. By pretending that you did, you've introduced a time delay between the voltage and the current sequences. This is equivalent to a small phase shift, which really messes up any calculation that's sensitive to the phase difference between voltage and current.

The error doesn't come from the sampling process itself, but from our choice of how to interpret and manipulate the sample values. Let's set up an example with four channels; much of this material is taken from a real project I architected for a super-precise electricity meter built with PSoC3. Let's refer to the four channels by the letters W through Z, and with sample number i written as W(i), and so on. With our ADC multiplexing round the four channels in order, it's pretty clear that the sequence of data words straight out of that ADC goes like sequence {1} and arrives at rate 4Fs:

W(1), X(1), Y(1), Z(1), W(2), X(2), Y(2), Z(2), W(3), X(3), Y(3), Z(3)... {1}

Our problems come when we take the simplistic approach of breaking this stream up into four separate streams in which we assume that samples with the same index can be treated as simultaneous, as in the set of sequences {2}, each now at rate Fs:

W: W(1), W(2), W(3), W(4)...
X: X(1), X(2), X(3), X(4)...
Y: Y(1), Y(2), Y(3), Y(4)...
Z: Z(1), Z(2), Z(3), Z(4)... {2}

The standard way of breaking up sequence {1} is to 'stuff' zeros into the locations where other channels are being sampled, to get four streams, sequences {3}. This is an example of interpolation:

W:	W(1),	0,	0,	0,	W(2),	0,	0,	0,	W(3),	0,	0...
X:	0,	X(1),	0,	0,	0,	X(2),	0,	0,	0,	X(3),	0...
Y:	0,	0,	Y(1),	0,	0,	0,	Y(2),	0,	0,	0,	Y(3)...
Z:	0,	0,	0,	Z(1),	0,	0,	0,	Z(2),	0,	0,	0... {3}

These streams are all running at 4Fs, and they clearly don't 'line up' any more. It's meaningless to do any arithmetic on values that are vertically aligned when written out as in {3}, i.e. happening "at the same time". One of any pair is guaranteed to be zero, and therefore so is their product. What's more, by taking the readings from each channel at Fs and producing an output stream that repeats at 4Fs, we've created 'images' of the frequency spectrum of our signals, centred on multiples of Fs; see figure 1, taken from the National Instruments website. This high frequency information wasn't present on our W~Z inputs originally. Haven't we made things even worse now?

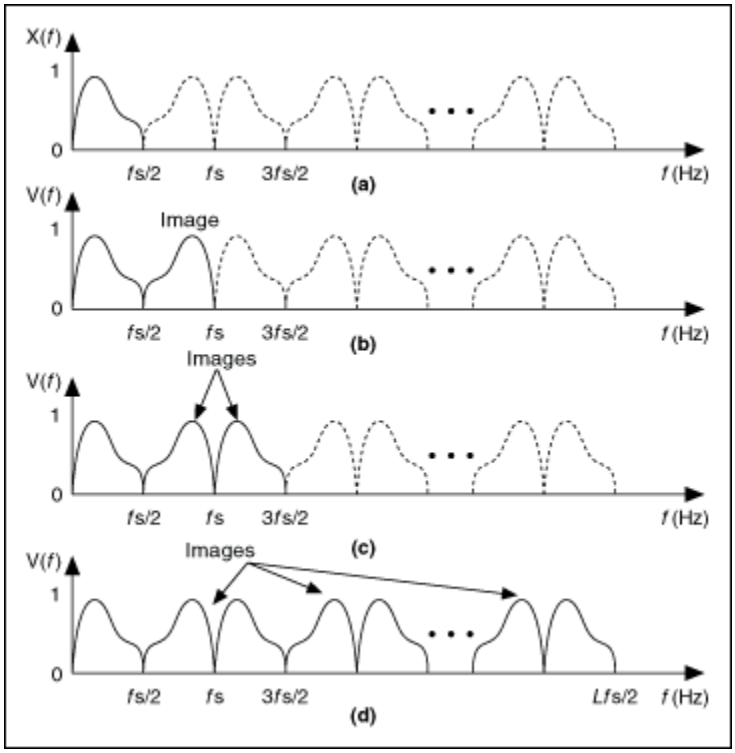

Figure 1: The generation of images through interpolation (from National Instruments).

All is not lost. There's a method for getting rid of the high frequency rubbish and getting our sample rate back down to Fs. It's called decimation, and to do it we use a decimation filter. The term 'decimation' has drifted in meaning from its origin as a translation of a dreadful punishment exacted by the Romans on disobedient or underperforming army units. In the rather less blood-thirsty discipline of signal processing, it refers to the selection of every Nth sample ("decimation by N") from a stream of data. It's sometimes called downsampling, especially in the audio world. In our example here, N=4, of course.

Decimation involves reducing the sample rate, and we know that we run the risk of aliasing, unless we take steps to remove those components from the signal whose frequency is higher than 0.5Fs. In other words, a decimation filter is really just a digital antialiasing filter. There are several types of filter topology we could choose to do the job in our case. Here I'll use an FIR lowpass filter, and the benefits of that choice will become obvious later on.

We're going to filter sequences {3} in such a way that we can then decimate them from 4Fs down to Fs. So we need a lowpass filter with a good stopband that begins at 0.5Fs when it's operating on 4Fs data. The frequency response of a suitable 128-tap FIR filter is shown in figure 2. It's a linear phase filter designed with the ever-so-easy "windowed sinc" method, using nothing more complicated than a spreadsheet. The stopband is never poorer than 75 dB down, which limits potential aliasing errors to a reasonable level for this application. You might spot that the passband gain is 12 dB, i.e. 4x. The reason for that choice of scaling factor will also become clear later.

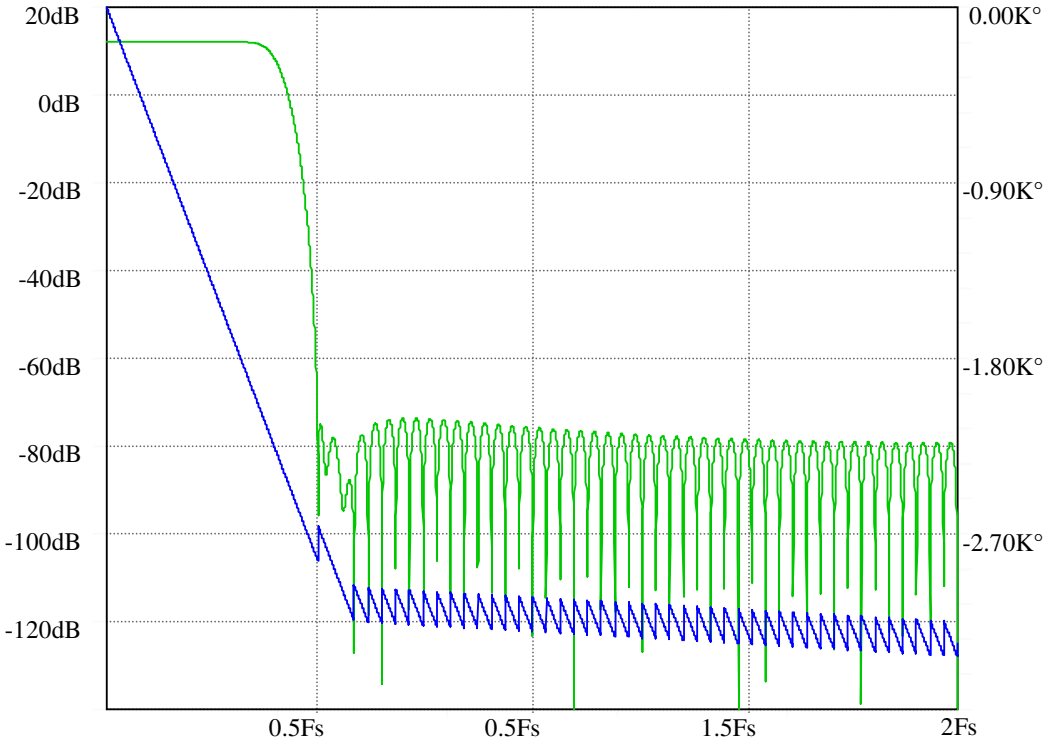

Figure 2: Magnitude and phase response of our starting 128-tap FIR lowpass.

So let's take four of these 128-tap filters, one for each of the sequences {3}. Samples go into, and come out of, each filter at 4Fs, to make new sequences {4}, none of whose samples are now zero:

W': W'(1), W'(2), W'(3), W'(4), W'(5), W'(6), W'(7), W'(8), W'(9)...
X': X'(1), X'(2), X'(3), X'(4), X'(5), X'(6), X'(7), X'(8), X'(9)...
Y': Y'(1), Y'(2), Y'(3), Y'(4), Y'(5), Y'(6), Y'(7), Y'(8), Y'(9)...
Z': Z'(1), Z'(2), Z'(3), Z'(4), Z'(5), Z'(6), Z'(7), Z'(8), Z'(9)... {4}

And because we've filtered off the high frequency stuff, we can now decimate each stream by a factor of four, meaning that we only take every fourth output sample, for example to give sequences {5} at Fs.

W": W'(1), W'(5), W'(9)...
X": X'(1), X'(5), X'(9)...
Y": Y'(1), Y'(5), Y'(9)...
Z": Z'(1), Z'(5), Z'(9)... {4}

So far, we could have used any form of digital filter, but using an FIR offers a great simplification. Look at sequences {3} again: three-quarters of the data values are zero. Each output value from the 128-tap FIR filter used to create sequences {4} is the result of adding together 128 individual products of input data value and coefficient value. But there's no point doing the multiply inside the filter structure when we know in advance that the data value is going to be zero. There's a large potential saving here; how do we exploit this?

The solution is to partition the FIR filter's impulse response into four subsets that match the positions of the non-zero samples in sequences {3}. This turns our 128-tap filter into four distinct 32-tap subfilters. Then, we use the 'misaligned' input sequences {2} as the inputs to these filters, each just running at Fs. Figure 3 shows how we split up the impulse response into the four 'phases' that get assembled into the four different filters. We still get sequences {4} as the output, it's just that we've eliminated all the redundant multiplies by zero.

Now, instead of implementing four (identical) 128-tap filters running at 4Fs, we just have to implement four (different) 32-tap filters each running at Fs. That's a pretty significant saving. We never actually need to create and decimate the sequences {3}. The new filters have the original frequency response, but it just hits its stopband right at Nyquist.

This is an example of **polyphase decimation**. There's a delay difference between each of these subfilters, and it's **exactly equal** to the time error caused by our original misalignment of the sample data in creating sequences {2}, which was one-quarter of the sample period of the output data between channels. This time difference therefore 'realigns' the data in the time domain, eliminating the error. How cool is that?

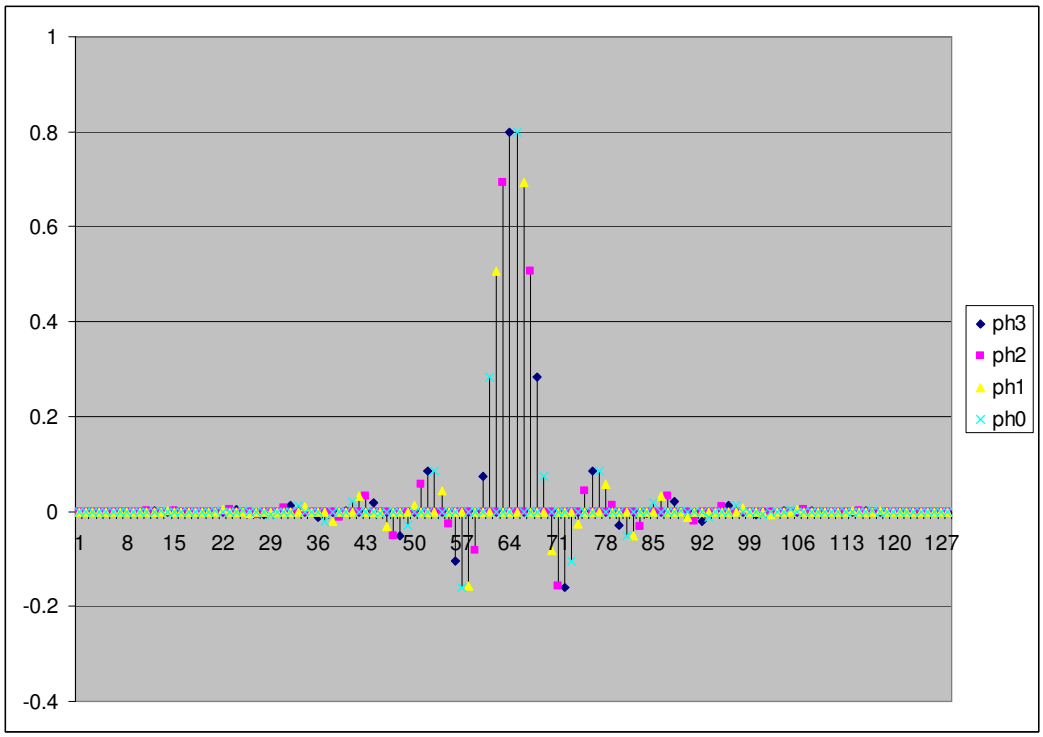

Figure 3: Breaking up our 128-tap FIR filter into four phases.

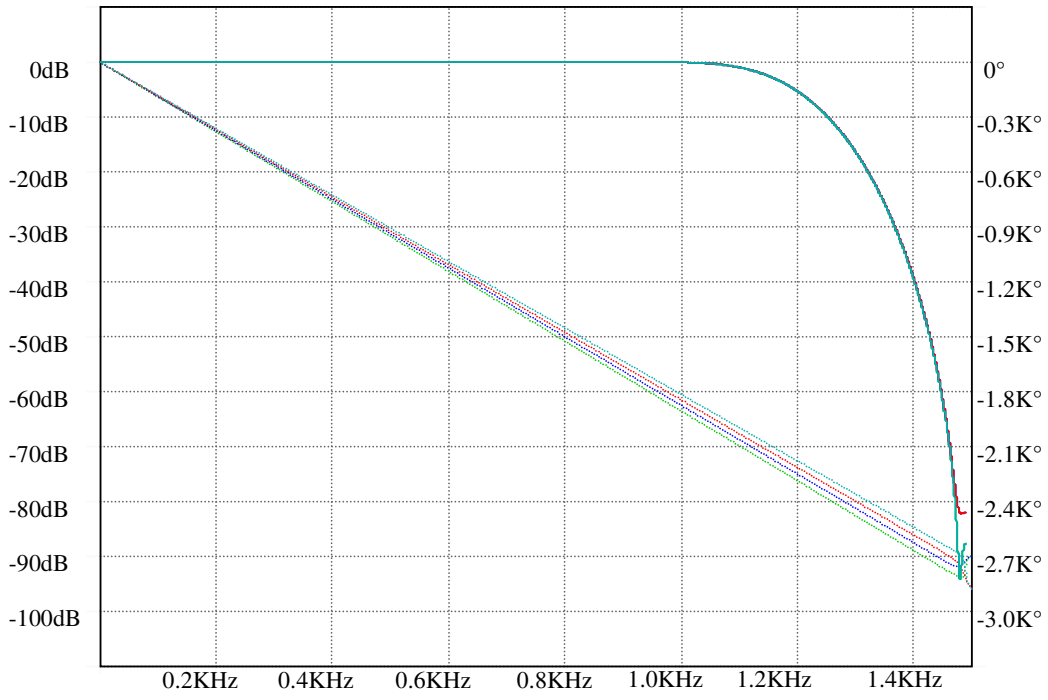

Figure 4: The four subfilters, showing overlapping magnitude and staggered phase.

Because we divided our impulse response into four sets, each ‘carries’ one-quarter of the signal. Figure 2 showed a gain of 4x to compensate this right from the start. You could

also start with a unity gain filter and just multiply all the final coefficients by four at the end, with the same result. Figure 4 shows the frequency and phase responses of the four subfilters; they now have unity gain. The amplitude responses are identical, but the phase responses are skewed apart as you'd expect from four filters that have slightly different delay. Figure 5 shows the group delay values of the four filters directly, along with an expanded passband plot. The simulations were taken for an Fs of 3000 Hz, which was the sample rate we used in the metering work.

If you'll pardon the cliché, the proof of the pudding is in the eating. Figures 6 and 7 are "before and after" plots for a segment of data from a four-channel .wav file acquired from a multiplexed-ADC project built on a PSoC3 development board. The source signal consists of equal amounts of 50 Hz and its first 21 harmonics, phased randomly. It was generated by playing out a synthesized .wav file through a PC's line output, and applied to the four multiplexed inputs (through carefully matched input AC coupling networks). We took a lot of care to ensure that everything on one channel settled really well before switching over to the next multiplexer channel (otherwise you get crosstalk, which can cause significant accuracy problems in a meter). Because the four channels aren't sampled at the same time, the data points for the four streams plotted in figure 6 aren't identical. This extreme waveform was chosen to make a visual point, even though it would be a rather unusual signal to apply to an actual electricity meter.

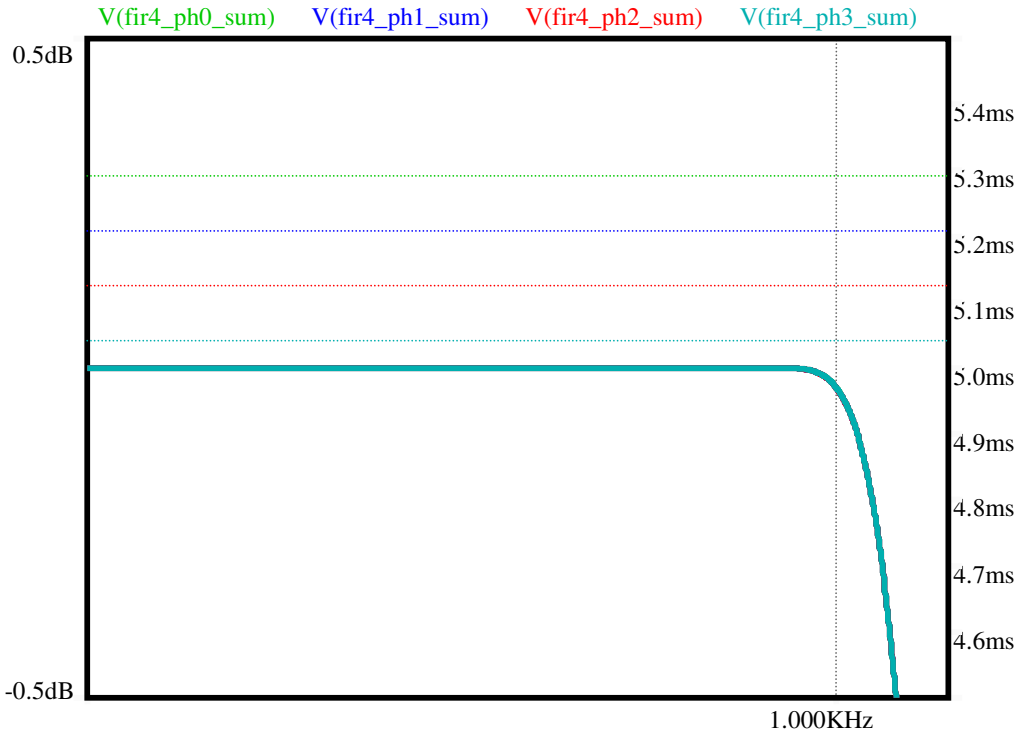

Figure 5: Closer detail of the passband gain and the different group delay values.

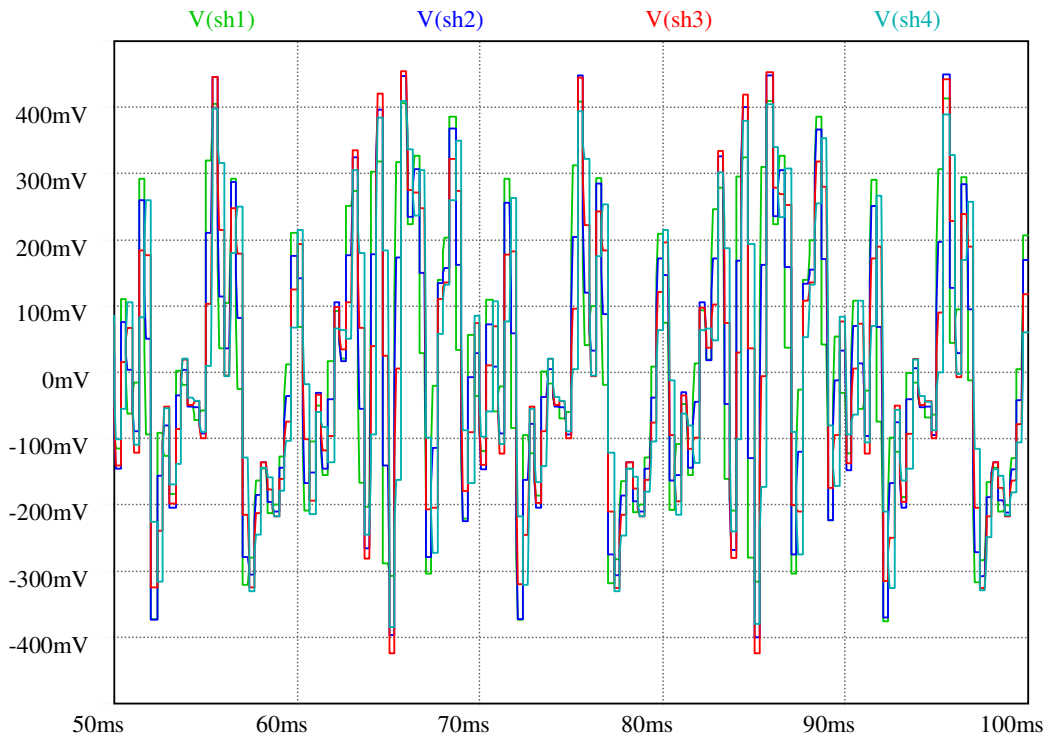

Figure 6: Four channels of sequentially acquired data from a single input signal.

Figure 7 shows the outputs from the four subfilters, all fed with the samples of figure 6, implemented using the Digital Filter Block in the PSoC3. Remember that the four filters have identical magnitude responses, and differ only in that their group delays are spread apart by units of a quarter of a sample time. Passage through these filters rather magically reshapes the applied waveforms so that the output signals are all identical, to within very close tolerances.

Are there downsides to this approach? Well, we've put each channel's data through a 32-tap filter, and the average group delay experienced by the channels is that of a 32-tap linear phase filter running at Fs, and that's 16/Fs. You might want that to be rather lower, especially at lower sample rates or in a closed-loop system. If you don't need such a wide band of flat frequency response, you can reduce the number of taps in the initial filter to reduce the delay. If you really do need the bandwidth, another approach is to use a **minimum-phase** FIR filter as your starting point. In sacrificing linear phase response, the low frequency group delay can be greatly reduced. The technique still works perfectly (I've tried it in simulation). Sophisticated filter design programs such as FilterShop from LinearX can produce minimum-phase FIR filters.

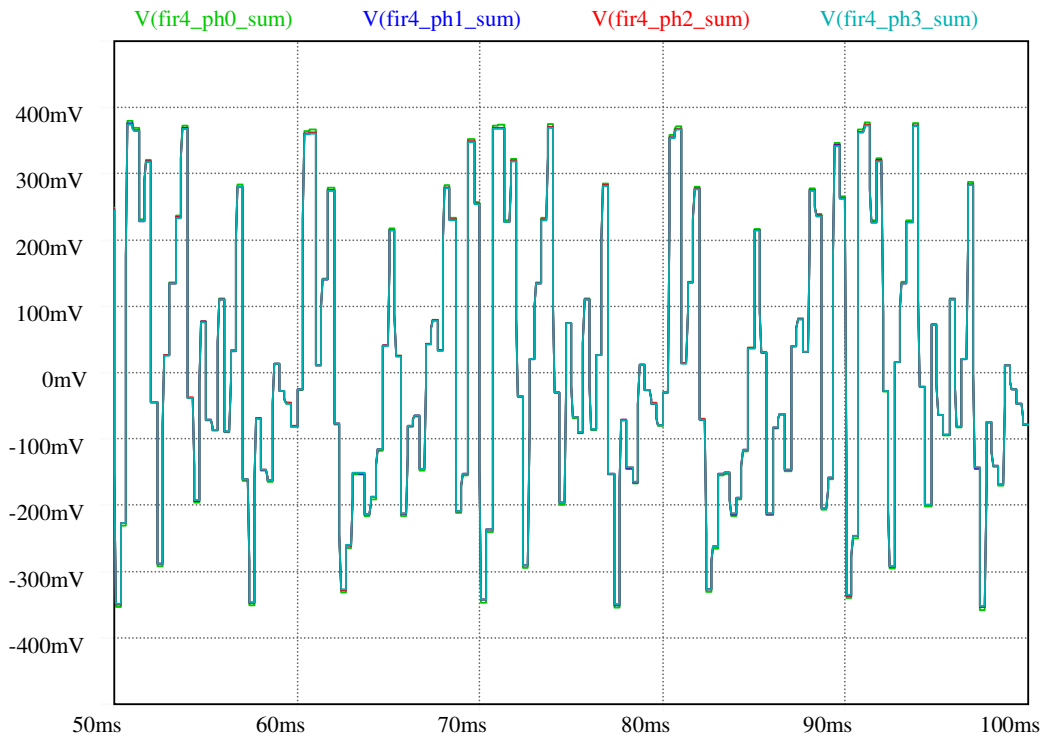

Figure 7: The four channels after passage through their 'realignment filters'.

This work was done to support our electricity metering analysis, but it has quite a few other applications. If you're analyzing structures, the cross-correlation terms in the vibration will be useless unless you can rely on simultaneity in the data sets. One intriguing application I looked at recently was a gunfire detection system sampling eight microphones at 60k samples per second each, with the Digital Filter Block implementing eight 12-tap subfilters derived from a minimum phase prototype, at an aggregate rate of 480 kHz. You can't do that in your average microcontroller – even a fancy 32-bit one!

So, remember, you don't need multiple ADCs just because you want data samples that behave like they were taken simultaneously across all channels. One single high quality ADC with a good input mux and some digital filtering Wizardry, and you're all set. Hope we're all aligned on that! / Kendall