BY BARRY L DORR • DORR ENGINEERING

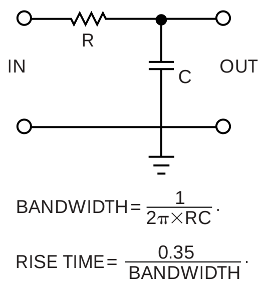
A simple software lowpass filter suits embedded-system applications

BUILDING A DIGITAL EQUIVALENT OF AN ANALOG LOWPASS RC FILTER REQUIRES JUST A COUPLE OF LINES OF FIXED-POINT C CODE.

Hardware-design engineers have long recognized the usefulness of RC lowpass filters (**Figure 1**). Lowpass filters are useful for performing signal conditioning, removing noise from a signal, or rejecting unwanted signals. The first-order recursive filter is the digital equivalent of the RC filter, and, as the ratio of the sample frequency to the bandwidth increases, their responses become identical.

All lowpass filters produce a weighted average of the current input value and past inputs. A filter's characteristics depend on the weighting used for the past inputs. For example, a lowpass filter (**Figure 2**) smoothes noisy input signals. One possibility for weighting the past-input samples is to give them equal weight. Although this approach produces a useful lowpass filter, it makes more intuitive sense to weigh newer samples more heavily than older samples.

A designer can use recursion to implement a weighting function (**Figure 3**). This approach minimizes the processing impact of the multiplications in this filter because the designer can implement the multiplications as left or rights shifts in the software. The designer feeds data samples with bit width B1 into the filter at a fixed sample rate. Bit width B2 may be larger than bit width B1. The summing block adds the input sample to the last output sample, and the delay is a digital delay of one sample. The parameter k is a shift value that controls the filter bandwidth. Mathematically, the recursion in **Figure 3** is: $y(n)=(1-2^{-k})\times y(n-1)+x(n)$, where x is the input, y is the output, and n is the sample index.



$$\text{BANDWIDTH}=\frac{1}{2\pi\times RC}.$$

$$\text{RISE TIME}=\frac{0.35}{\text{BANDWIDTH}}.$$

Figure 1 Lowpass filters are useful for performing signal conditioning, removing noise from a signal, or rejecting unwanted signals.

As an example of filter operation, suppose that k=4 so $1-2^{-k}=0.9375$, the value at the delay output is zero, and the filter input is a single sample of one followed by all zeros. If you use the summing block as the output, the first output from the filter is one. When you feed this output back to the summing block, the multiplier scales it, and it becomes 0.9375. The next output is 0.9375^2 or 0.8789, and the nth output is 0.9375^n. This sequence is the impulse response of the filter, but it is also the weighting function (**Figure 4**). Good stuff is going on here for those who like math, but we'll stick to the implementation.

SPECIFYING THE FILTER RESPONSE

A designer can specify the filter response in either the frequency domain or the time domain. Which one to use depends on the type of problem you are working on. If noise or unwanted tones are the problems, then the frequency-domain specification or bandwidth is appropriate. To reject impulse-noise hits, or smooth measured data, the time-domain specification or rise time may work better.

When working in the frequency domain, the designer specifies the 3-dB frequency or bandwidth of the filter. At this frequency, the amplitude of the output drops to 0.707 times the amplitude that a dc signal causes (**Figure 5a**). You can estimate the atten-

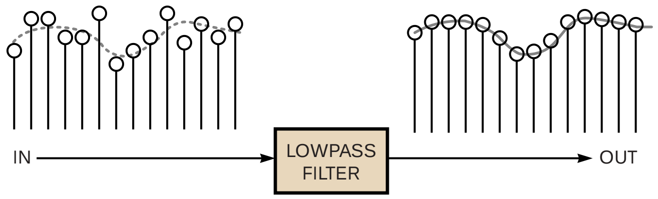


Figure 2 A filter's characteristics depend on the weighting used for the past inputs. For example, a filter smoothes noisy input signals.

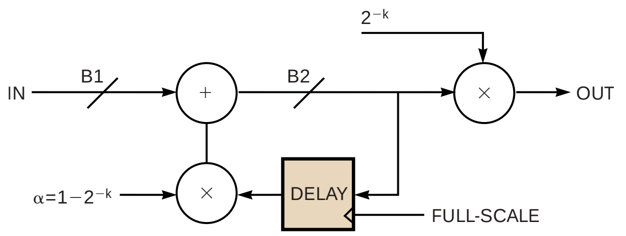


Figure 3 A designer can use recursion to implement a weighting function.

uation at higher frequencies because doubling the frequency approximately halves the amplitude. Table 1, which is normalized to a sample rate of 1 Hz, shows the normalized bandwidth and rise time for several values of k. To get the actual bandwidth, multiply the value in the table by the sampling frequency.

When working in the time domain, the designer specifies the rise time of the output in response to a step input. Figure 5b shows the filter output when the input changes from all zeros to all ones, which causes the output to gradually move from zero to one. The rise time is the time necessary for the filter output to move from 10 to 90% of the final value. Table 1 specifies the rise time in number of samples.

Before writing code to implement this filter, the designer must specify the number of bits necessary for the summing block and the delay. Because the filter is basically an averaging device, the range of the summing-block output is larger than the range of the input signal if there is to be no loss in precision. Fortunately, it is easy to predict the growth in the output-register width because the dc gain from the input to the summing-block output is 2^k, so bit width B2 from Figure 2 is k bits wider than bit width B1. You can maintain a unity gain by multiplying the summing-block output by 2^{-k}.

The fixed-point code in Listing 1 implements a filter for the case of k=4. Using a power of two for k enables the use of right shifts in the code to avoid the performance hit from the multiplier blocks. Using a 32-bit integer for the summing block and delay accommodates the four bits of growth in the output register. When using this code, make sure that your compiler sign extends when right-shifting a signed number.

The first-order recursive filter, or "leaky integrator," is a simple yet powerful filter that is a time-tested approach for many filtering applications, and this implementation requires no multiply instructions. The steps to quickly and accurately implement this filter are to specify the filter using either the rise time or the bandwidth, allocate k additional bits for the summing block and delay to accommodate register growth, and implement the filter by substituting shifts instead of multiply instructions as the example code shows. EDN

TABLE 1 NORMALIZED BANDWIDTH AND RISE TIME FOR VARIOUS VALUES OF k

k	Bandwidth (normalized to 1 Hz)	Rise time (samples)
1	0.1197	Three
2	0.0466	Eight
3	0.0217	16
4	0.0104	34
5	0.0051	69
6	0.0026	140
7	0.0012	280
8	0.0007	561

LISTING 1 CODE FOR THE SIMPLE FILTER

```
#define FILTER_SHIFT 4          // Parameter K
#define sint32 (signed long)    // Specify 32-bit integer
#define sint16 (signed short)   // Specify 16-bit integer

sint32 filter_reg;             // Delay element – 32 bits
sint16 filter_input;           // Filter input – 16 bits
sint16 filter_output;          // Filter output – 16 bits

// Update filter with current sample.
filter_reg = filter_reg - (filter_reg >> FILTER_SHIFT) + filter_input;

// Scale output for unity gain.
output = filter_reg >> FILTER_SHIFT;
```

AUTHOR'S BIOGRAPHY

Barry Dorr is the president of Dorr Engineering Services (San Marcos, CA), a consulting company that specializes in signal processing, modem development, and embedded servos. He holds a bachelor's degree from California Polytechnic State University (San Luis Obispo, CA) and a master's degree from San Diego State University and is a registered professional engineer in California. Outside work, he enjoys spending time with his children and playing the trombone. Dorr also teaches a course in embedded servo systems. For more information, go to www.dorrengineering.com/downloads.htm. You can reach him at bdorr@dorrengineering.com.

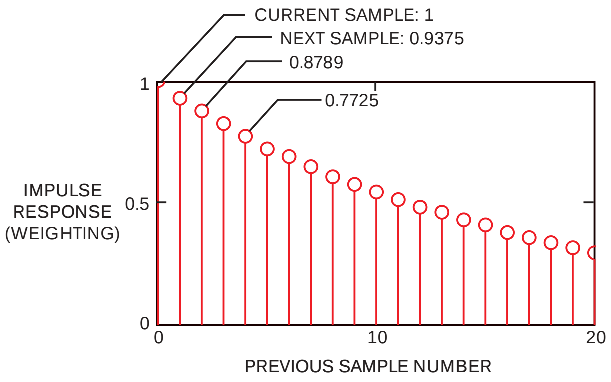


Figure 4 This sequence is the impulse response of the filter, but it is also the weighting function; k=4, so $1-2^{-k}=0.9375$.

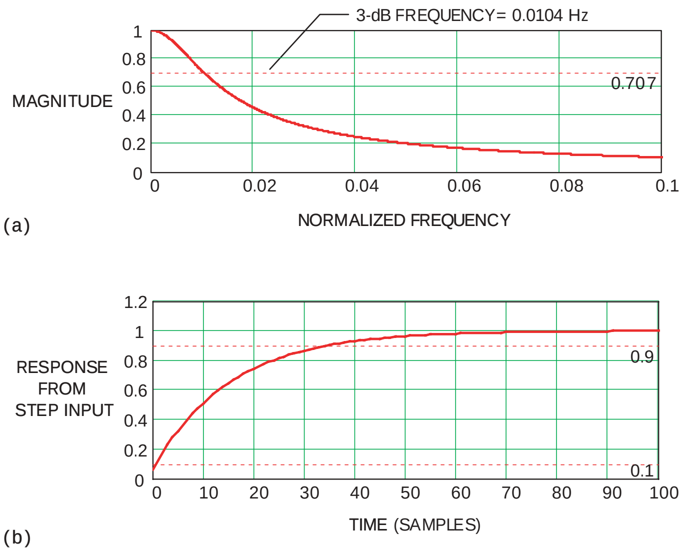


Figure 5 At this frequency, the amplitude of the output drops to 0.707 times the amplitude that a dc signal causes (a). When the input changes from all zeros to all ones, the output gradually moves from zero to one (b).