AN2099

PSoC® 1, PSoC 3, and PSoC 5 - Single-Pole Infinite Impulse Response (IIR) Filters

Author: David Van Ess, Praveen Sekar
Associated Project: Yes
Associated Part Family: CY8C3XXX, CY8C5xxx, CY8C2xxxx
Software Version: PSoC Creator™ 2.0, PSoC Designer™
Related Application Notes: None

Abstract

AN2099 describes a topology for a single-pole infinite impulse response (IIR) filter. It includes equations and software to implement this topology; the associated example projects give the user access to filter routines in either assembly or C.

Introduction

In many applications, you need to apply a post-processing digital filter on your data. A common example is the filtering of data from an ADC. This application note shows an IIR filter that can be constructed using the CPU (without any addition digital filtering hardware) and the equations necessary for calculating the roll-off frequency (f_0). A comparison with the moving average FIR filter, which is commonly used for filtering digital data in a CPU, is also shown.

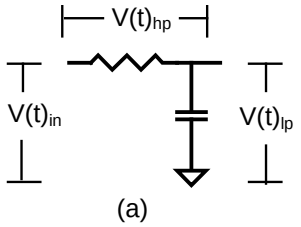
Infinite Impulse Response (IIR) Filters

An IIR filter is a recursive filter; that is, the output is used to calculate future values. Theoretically, an impulse injected into the input continues to flow through the signal loop. It takes infinite time for the effect of the impulse to die down completely.

The single-pole passive RC filter shown in Figure 1 has the characteristics of an IIR filter. If you give an impulse input, it takes an infinite time for the capacitor voltage to go down completely to zero.

We will build the topology of the single-pole IIR digital filter from the single-pole passive RC filter shown in Figure 1.

Figure 1. Single-Pole Passive RC Filters



In Figure 1, the low-pass filter output is available across the capacitor and the high-pass filter output is available across the resistor.

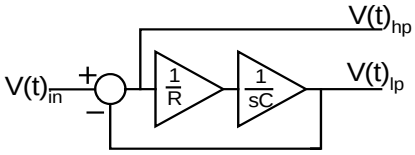
The output voltage, V_{lp}, at any instant is a function of the current flowing into the capacitor which is a function of V_{in}-V_{lp}. This represents a negative feedback.

To construct a negative feedback equivalent to the analog RC filter in Figure 1, consider the following statements:

- At any instant, V_{in}-V_{lp} is divided by R to give a current, which is then integrated to give V_{lp}.
- V_{in}-V_{lp} represents the high-pass filtered output.

Figure 2 shows the negative feedback topology equivalent to that in Figure 1.

Figure 2. Negative feedback topology for Single-Pole RC Filter



Equations 1 and 2 define its operation:

$$V_{hp} = V_{in} - V_{lp} \quad \text{Equation 1}$$

$$V_{lp} = \frac{V_{hp}}{sCR} \quad \text{Equation 2}$$

Combining Equations 1 and 2 produces the transfer functions in Equations 3 and 4:

$$\frac{V_{hp}}{V_{in}} = \frac{sCR}{1+sCR} \quad \text{Equation 3}$$

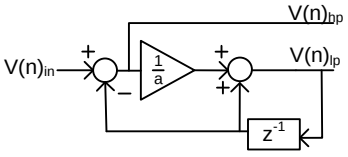
$$\frac{V_{lp}}{V_{in}} = \frac{1}{1+sCR} \quad \text{Equation 4}$$

These are the standard transfer functions for high-pass and low-pass filters. The roll-off frequency f_0 is shown in Equation 5:

$$f_0 = \frac{1}{2\pi CR} \quad \text{Equation 5}$$

The topology for a sampled system is constructed from the topology in Figure 2 and is shown in Figure 3. Figure 3 shows that the integrator is replaced with an accumulator (summer), and the term RC is replaced by a scaling factor. The z^{-1} box represents a register that stores the previous value of $V(n)_{lp}$.

Figure 3. IIR Topology for Sampled Single-Pole RC Filters



Equations 6 and 7 define its operation:

$$V_{hp} = V_{in} - V_{lp} z^{-1} \quad \text{Equation 6}$$

$$V_{lp} = \frac{V_{hp}}{a} + V_{lp} z^{-1} \quad \text{Equation 7}$$

where z^{-1} represents a unit sample delay. Replacing z^{-1} with a unit sample delay gives the difference Equations 8 and 9.

$$V_{hp}(n) = V_{in} - V_{lp}(n-1) \quad \text{Equation 8}$$

$$V_{lp}(n) = \frac{V_{hp}(n)}{a} + V_{lp}(n-1) \quad \text{Equation 9}$$

Combining Equations 6 and 7, we get the transfer functions in Equations 10 and 11.

$$\frac{V_{hp}}{V_{in}} = \frac{a(1-z^{-1})}{a+z^{-1}(1-a)} \quad \text{Equation 10}$$

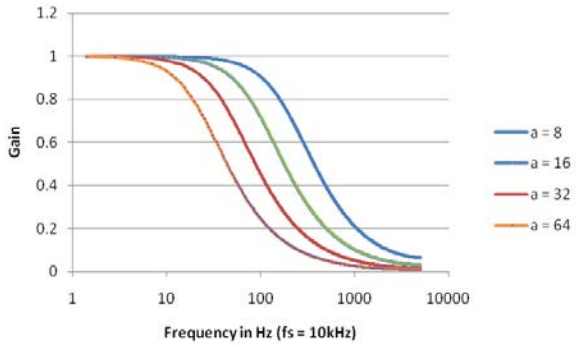
$$\frac{V_{lp}}{V_{in}} = \frac{1}{a+z^{-1}(1-a)} \quad \text{Equation 11}$$

It can be shown that the roll-off frequency f_0 for the digital filter represented by Equations 6 and 7 can be approximated by Equation 12. (See Appendix A)

$$f_0 = \frac{f_s}{2\pi a} \quad \text{Equation 12}$$

Equation 12 is the same as Equation 5 with RC = a. The roll-off frequency is dependent on the sample frequency $\mathbf{f}_s$, but more importantly the attenuation value (a). Changing the attenuation value easily changes the filter's roll-off frequency. If you increase the attenuation factor, you can push the filter cut-off frequency lower. However, this results in an increased filter settling time.

Figure 4. Filter Cut-off Versus Attenuation Factor



Filter Settling Time

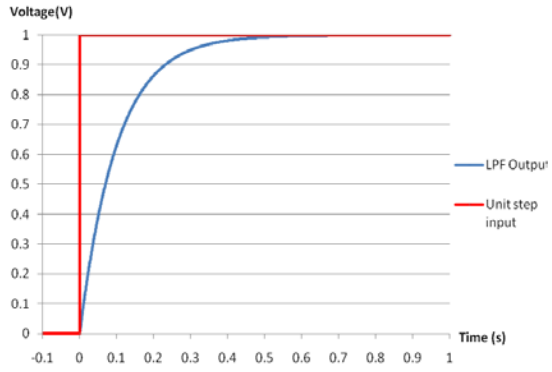
Settling time in a low-pass filter is the time taken by the output to reach a certain percentage of a step input.

For the analog RC low-pass filter shown in Figure 1, the output for a step input will be an exponential rise defined by Equation 11.

$$V_{out} = V_{in}(1 - e^{-t/RC}) \quad \text{Equation 13}$$

For Vin = 1 V; R = 1 k, and C = 10 uF, the filter settling (Vout Vs time) for a unit step input is shown by the blue line in Figure 5.

Figure 5. Analog LPF Unit Step Response



To find the time taken by the filter to reach 99.9% of the input, we substitute, V_{out} = 0.999; V_{in} = 1; R = 1 k; C = 10 uF in Equation 13.

$0.999 = (1 - e^{-100t})$ or

t = 0.7 s

0.7 seconds are necessary to reach 99.9% of the step input. This time depends on the choice of R and C.

Similar to the analog low-pass filter output, the digital low-pass filter output also takes a finite time to reach within a certain percentage of its input. In this case, the settling time is determined by the attenuation factor, **a**.

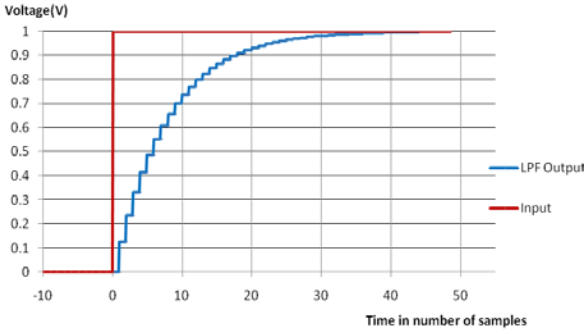
It can be shown (see Appendix B) that for a unit step input, the amplitude of the nth sample output, s[n], is given by Equation 14.

$$s[n] = 1 - \left(1 - \frac{1}{a}\right)^n$$

Equation 14

Equation 14 is the digital equivalent of Equation 13 with V_{in} = 1. Figure 6 plots s[n] (the unit step response) Vs n (the number of samples) for an attenuation factor of 8. The red line shows the input and the blue line shows the filter output.

Figure 6. Digital LPF Unit Step Response



To calculate the number (n) of samples required for the output to reach 99.9% of the input, you must substitute s[n] = 0.999 in Equation 13 and compute n.

$$0.999 = \left(1 - \frac{1}{8}\right)^n \quad \text{or}$$

$$n = 52$$

It takes 52 samples for the filter to reach 99.9% of the input voltage. If you filter an ADC output using this LPF (a = 8) and if the ADC sample rate is 52 sps, it would take approximately a second for the filter to reach 99.9% ADC input voltage.

The filter settling time is also specified in terms of number of bits.

The settling time for the filter output to reach n-bit accuracy is the time taken by the filter to settle to within

$(\frac{1}{2^n})$ of the input voltage.

Table 1 shows the settling time for 10, 12, and 16-bit accuracies.

For 10-bit accuracy, the output should reach within 0.1% of the input value; For 12-bit accuracy, the output should reach within 0.025% of the input value and for 16-bit accuracy, the output should reach within 0.0015% of the input value. For a 1 V input, the filter achieves 10-bit accuracy at 999.03 mV, 12-bit accuracy at 999.75 mV and 16-bit accuracy at 999.985 mV.

Table 1. Filter Settling Times

Attenuation factor, a	10-bit accuracy	12-bit accuracy	16-bit accuracy
1	0	0	0
2	10	12	16
4	25	29	39
8	52	63	84
16	108	129	173
32	219	262	350
64	441	527	706
128	885	1058	1417
256	1765	2120	2838

How to Implement the Filter

To implement the low-pass and high-pass filters shown in Figure 3, we can directly use Equations 8 and 9. The steps are given below

1. Using your sample frequency (**f_s**) and the roll-off frequency that you want to use (**f_0**), find the necessary divisor (**a**) using Equation 12.
2. Subtract the old **V_{lp}** from **V_{in}**. This is the new **V_{hp}**.
3. Divide **V_{hp}** by **a**.
4. Add the value generated in step 3 to the old **V_{lp}**. This is the new **V_{lp}**. Note that only **V_{lp}** needs to be saved for the calculation of the next values.

Step 3 requires you to perform a division (involves floating point arithmetic) by the attenuation factor (a), which can take any real value greater than unity. If you choose the attenuation factor carefully, you can perform the division with just shifts and add. Let's limit the attenuation factor to this set of values:

$$a = \left\{ \frac{256}{255}, \frac{256}{254}, \frac{256}{253}, ..., \frac{256}{1} \right\}$$ Equation 15

For example, if the attenuation factor is 256/99, Equation 9 becomes:

$$V_{lp}(n) = V_{lp}(n-1) + (V_{in}(n) - V_{lp}(n-1)) * \frac{99}{256}$$ Equation 16

$$\frac{99}{256} = \frac{64}{256} + \frac{32}{256} + \frac{2}{256} + \frac{1}{256} = \frac{1}{4} + \frac{1}{8} + \frac{1}{128} + \frac{1}{256}$$

This can be easily done by a combination of shifts and add as shown by the C code snippet (Code 1). The variable 'filt' denotes the low-pass filter output and the variable 'input' denotes the low-pass filter input.

Code 1

```
filt = filt + ((input-filt) >> 2) + ((input-
filt) >> 3) + ((input-filt) >> 7) + ((input-
filt) >> 8);
```

Similarly, any attenuation factor in the set can be implemented with just shifts and additions.

The negative side of limiting the attenuation factor in this way is that you may not have enough choices when the attenuation factor gets higher. For example, if you want an attenuation factor of 145, you must use 128 or 256. But if you want an attenuation factor of 6.5, you can choose a value very close to it, 6.5 which is 256/39. But, this should not be a serious problem because higher attenuation factors are generally not desired due to their higher settling times.

For most practical purposes, you might require an attenuation factor in the range of 4 to 64 and can easily match it to one of the available values.

Binary Weighted IIR Filters

If the value of the attenuation factor, a, is a power of 2, it is much easier to implement the filter. All that is required is one shift and one addition. If the attenuation factor is 2^i, the filtering can be done very easily using the following line of code.

```
filt = filt + (input-filt) >> i;
```

This executes much faster and takes less code space as well. In most cases, digital filtering is done to reduce the noise, where the requirement to implement the actual cut-off frequency is not stringent. Binary-weighted IIR filters can be used in those cases.

Single-pole IIR Filter versus Moving Average FIR Filter

A moving average filter is also commonly used for reducing noise in the digital output. A moving average filter is implemented by simply taking the average of the last N samples as defined by Equation 17. The higher the value of N, the lower is the filter cut-off.

$$y[n] = \frac{x[n] + x[n-1] + x[n-2] + ... + x[N-1]}{N}$$

Equation 17

The pros and cons of the moving average filter and the single-pole IIR filter are as follows.

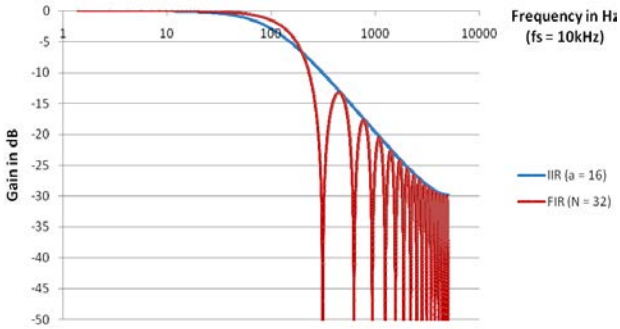
- A single-pole IIR filter is very easy to implement and requires a storage space of just one memory element to store the past output. But, the settling time increases as the attenuation factor increases and the filter takes infinite samples to settle to the input value.
- A Moving Average FIR filter is relatively complex to implement and requires a storage space of N elements, where N is the number of samples averaged. The filter takes just N samples to settle to the input value.
- A moving average filter with N elements has a frequency response roll-off comparable to an IIR filter with attenuation factor, a = N/2. Or, in other words, the roll-off of a 32-tap moving average filter can be matched by an IIR filter with an attenuation factor of just 16. Figure 7 shows the frequency response performance of a 32-element moving average filter and a single-pole IIR filter of attenuation factor 16.

  However, if you compare time responses, you can see that it takes 108 samples (Table 1) for the IIR filter to settle to 99.9% of the input while the moving average FIR takes just 32 samples to settle to 100% of the input signal.

Table 2. Single-Pole IIR Versus Moving Average FIR

	Single-pole IIR	Moving Average FIR
Defining parameter	Attenuation factor, a	Number of samples averaged, N
Firmware implementation	Easy	Relatively complex
Memory storage required	Single-pole IIR	Moving Average FIR
Settling time	Infinite	N

Figure 7. Frequency Responses of Moving Average and Single-Pole IIR filters



Associated Projects

This application note includes four projects:

- IIR_Filter_PSoC3_5 - PSoC Creator workspace containing two projects, one for PSoC 3 (IIR_PSoC3) and one for PSoC5 (IIR_PSoC5)
- AN2099_asm - PSoC designer project for PSoC 1 in assembly
- AN2099_C -PSoC designer project for PSoC 1 in C.

PSoC 3 Filter Implementation

The PSoC Creator project IIR_PSoC3, performs low-pass filtering on the 20-bit Delta Sigma ADC output and displays the filtered and unfiltered value on the LCD.

The filter routine takes the filter input as an argument and returns the filtered output after low-pass filtering the input data with an attenuation factor of 16.

Code 2

```
int32 LowPassFilter(int32 input)
{
  int32  k;
  input <<= 8;
  filt = filt + ((input-filt) >> 4);
  k = (filt>>8) + ((filt & 0x00000080) >> 7);
  return k;
}
```

In this code, the statement

```
filt = filt + ((input-filt) >> 4);
```

performs the filtering with an attenuation factor of 16. Attenuation factor can be easily changed to the required value by replacing this line of code with a code similar to Code 1 (code 1 implements an attenuation factor of 2.56)

The variable 'input' is the input to the low-pass filter. The variable 'filt' is a long int variable that accumulates the filter running sum.

To avoid any loss of precision due to right shifts, we left shift the input variable. The input is left-shifted by 8 so that we can perform right shift of 8 (maximum right shift for a = 256) without losing precision. For this reason, the variable filter should at least be 8 bits wider than the input. The variable 'filter' is declared a long int variable while 'input' is declared an int variable. This left-shift by 8 is compensated by performing a right-shift by 8 before returning the LPF output. This is done by the following statement.

```
k = (filt>>8) + ((filt & 0x00000080) >> 7);
```

While right-shifting, perform rounding-off instead of truncation. The additive term `((filt & 0x00000080) >> 7)` checks the Most Significant bit (bit 8) of the bits shifted out and performs the round-off before shifting the 8 bits out.

PSoC 5 Filter Implementation

In PSoC 5, the filtering can be done much faster due to the availability of a single cycle multiply instruction in ARM cortex M3. In PSoC 5, the filter routine for low-pass filtering (a = 256/17) looks like this:

Code 3

```
int32 LowPassFilter(int32 input)
{
  int32 k;
  input <<= 8;
  filt = filt + (((input-filt) >> 8) * 17);
  k = (filt>>8) + ((filt & 0x00000080) >> 7);
  return k;
}
```

The third statement performs the filtering. As you can see, the division by 256 is done by right-shifting by 8 and the multiplication is performed directly (without shift and add). It is very easy to change the attenuation factor to any value in Equation 15 by directly changing the value 17 to the required value. Note that the division is performed before the multiplication so that the variable filt stays within the int32 limit.

The variable filt needs to be initialized and is done by the SimpleLowPassInit function shown below. In both the PSoC 3 and PSoC 5 projects, the variable filt is initialized with the first ADC sample. This improves the initial settling time of the filter. (Only the initial settling time because the first value is acquired directly rather than the filter settling to the first value)

Code 4

```
void LowPassInit(int32 init)
{
       filt = init << 8;
}
```

This function left-shift the initial value by 8 to make it compatible with the LowpassFilter function, which also performs a left-shift by 8.

If you program the PSoC Creator project in a PSoC 3 chip, you can see that the filtered value is comparatively more stable than the raw ADC output. Try changing the attenuation factor (a), and notice the change in settling time and stability of the filtered output.

PSoC 1 Filter Implementation

The two PSoC 1 based projects provide both low-pass and high-pass filter functions written in assembly. The functions have been written in assembly in PSoC 1 because PSoC 1 CPU is low MIPS compared to that of PSoC 3 and PSoC 5. The functions can be found in the file *IIRFilters.asm* in the PSoC Designer project. One project has the main file written in assembly, the other project has the main file in C.

Both projects are similar to the PSoC Creator project in that they take the ADC input and show both the ADC output and low-pass filtered output in an LCD.

IIR High-Pass Filter – PSoC 1

The iSimpleHighPassFilter function in the file IIRFilters.asm implements a high-pass filter with an attenuation factor of 256. Equation 18 defines the roll-off frequency. An attenuation of 256 is selected because it can easily be implemented with a byte shift.

$$f_0 = \frac{f_s}{2\pi 256} = \frac{f_s}{1608} \qquad \text{Equation 18}$$

For a sample rate of 5 ksps, the roll-off frequency is 3.1 Hz.

The function takes a 16-bit signed input and, using the old $\mathbf{V_{lp}}$, calculates the next value. The new 16-bit $\mathbf{V_{hp}}$ is returned. This function assumes that 16-bit data is input through the **X** (MSByte) and **A** (LSByte) registers. The output is a 16-bit high=pass value returned through the X and A registers. Code 1 shows the function.

This function is not called in the project. Only the iSimpleLowPassFilter function is called. But if needed, you can call this function in your project the same way the iSimpleLowPassFilter function is called.

Code 5

```
area bss(RAM)
    iVlp:     BLK  3
    ;[iVlp]     = MSByte
    ;[iVlp + 1] = LSByte
    ;[iVlp + 2] = Residue
area text(ROM,REL)
export  SimpleHighPassFilter
export _SimpleHighPassFilter
;;---------------------------------
;;  SimpleHighPassFilter:
;;
;;  Take input and output new
;;  highpassvalue
;;  INPUTS:  X,A Vin
;;  OUTPUTS: X,A Vhp
;;---------------------------------
 SimpleHighPassFilter:
_SimpleHighPassFilter:
   sub  A,[iVlp+1]
   swap A,X
   sbb  A,[iVlp]
   ;Vhp now in A,X
   cmp A,128 ;test if Vhp is neg
   swap A,X
   if1: jc elseif1 ;(if neg)
      add [iVlp+2],A
      swap X,A
      adc [iVlp+1],A
      swap A,X
      adc [iVlp],ffh ;extended neg sign
      ret
   elseif1:;(pos)
      add [iVlp+2],A
      swap X,A
      adc [iVlp+1],A
      swap A,X
      adc [iVlp],0 ;extended pos sign
      ret
   endif1:
;---------------------------------
```

An initialization function is required to set the initial value for $\mathbf{V_{lp}}$. Code 6 shows this function:

Code 6

```
;;---------------------------------
;;  SimpleHighPassInit:
;;
;;  Initializes the Vlp value
;;  INPUTS:  X,A Init Value
;;  OUTPUTS: None.
;;---------------------------------
 SimpleHighPassInit:
_SimpleHighPassInit:
   mov [iVlp],X
   mov [iVlp + 1],A
   mov [iVlp + 2],0
ret
;;---------------------------------
```

IIR Low Pass Filter – PSoC 1

The iSimpleLowPassFilter function implements a low-pass filter with attenuation factor 256/12. The cut-off frequency is given by Equation 19. To change the attenuation, you have to change the SHIFT ACCUM sequence in the assembly code.

$$f_0 = \frac{f_s * 12}{2\pi 256} = \frac{f_s}{134} \qquad \text{Equation 19}$$

Code 3 is the function that takes a 16-bit signed input and, using the old $\mathbf{V_{lp}}$, calculates the next value. The new 16-bit $\mathbf{V_{lp}}$ is returned:

Code 7

```
macro  SHIFT
   asr [TempReg]
   rrc [TempReg + 1]
   rrc [TempReg + 2]
endm

macro  ACCUM
   mov A,[TempReg + 2]
   add [iVlp1 + 2],A
   mov A,[TempReg + 1]
   adc [iVlp1 + 1],A
   mov A,[TempReg]
   adc [iVlp1],A
endm

export  iSimpleLowPassFilter
export _iSimpleLowPassFilter
;;-----------------------------------
;;  iSimpleLowPassFilter:
;;
;;  Take input and output new
;;  higpassvalue
;;  INPUTS:  X,A Vin
;;  OUTPUTS: X,A Vhp
;;-----------------------------------
 iSimpleLowPassFilter:
_iSimpleLowPassFilter:
   push A
   mov A,0
   sub A,[iVlp1+2]
   mov  [TempReg + 2],A
   pop  A
   sbb  A,[iVlp1+1]
   mov  [TempReg + 1],A
   mov  A,X
   sbb  A,[iVlp1]
   mov  [TempReg],A

   SHIFT
   SHIFT
   SHIFT
   SHIFT
   SHIFT
   ACCUM ;32
   SHIFT
   ACCUM  ;64

   mov  A,[iVlp1 + 2]

   add  A,128
   mov  X,[iVlp1]
   mov  A,[iVlp1 + 1]
   adc  A,0
   swap X,A
   adc  A,0

   swap X,A
ret
;-----------------------------------
```

This function also requires an initialization function. It is shown in the following example code:

Code 8

```
export  SimpleLowPassInit
export _SimpleLowPassInit
;;-----------------------------------
;;  SimpleLowPassInit:
;;
;;  Initializes the Vlp value
;;  INPUTS:  X,A Init Value
;;  OUTPUTS: None.
;;-----------------------------------
 SimpleLowPassInit:
_SimpleLowPassInit:
   mov [iVlp1],X
   mov [iVlp1 + 1],A
   mov [iVlp1 + 2],0
ret
;;-----------------------------------
```

Summary

Single-pole IIR filters are useful for signal processing. Four projects have been presented that can easily be tailored to your filtering requirements.

Appendix A

The actual cut-off frequency for the single-pole low-pass filter can be derived from Equation 20

$$\frac{V_{lp}}{V_{in}} = \frac{1}{a + z^{-1}(1-a)} \qquad \text{Equation 20}$$

Substituting $z^{-1} = e^{-jw}$, we get,

$$\frac{V_{lp}}{V_{in}} = \frac{1}{a + e^{-jw}(1-a)}$$

$$\frac{V_{lp}}{V_{in}} = \frac{1}{a + (\cos w - j \sin w)(1-a)}$$

The magnitude of the transfer function is given by Equation 21

$$\left|\frac{V_{lp}}{V_{in}}\right| = \frac{1}{\sqrt{\left((a + (1-a)\cos w)^2 + ((1-a)\sin w)^2\right)}}$$

Equation 21

$$\left|\frac{V_{lp}}{V_{in}}\right| = \frac{1}{\sqrt{\left((1 + 2a^2 - 2a(1-(1-a)\cos w\right)}}$$

At the cut-off frequency ($w = w_0$), $\left|\frac{V_{lp}}{V_{in}}\right| = \frac{1}{\sqrt{2}}$

$$\frac{1}{\sqrt{2}} = \frac{1}{\sqrt{\left((1 + 2a^2 - 2a(1-(1-a)\cos w_0\right)}}$$

or

$$f_0 = \frac{f_s}{2\pi}\cos^{-1}\left(1 - \frac{1}{2a(a-1)}\right)$$

Equation 22

Equation 21 gives the actual expression for the cut-off frequency for the single-pole low-pass IIR filter. But for a > 8 (Filter attenuation factors commonly used are between 8 and 64), Equation 21 can be approximated by Equation 23 which is given below

$$f_0 = \frac{f_s}{2\pi a} \qquad \text{Equation 23}$$

Table 3 shows the difference between the two cut-off frequencies (Equations 12 and 21) at different values of attenuation factor at a sample frequency of 10 kHz

Table 3. Cut-off frequency approximation error

a	Cut-off calculated using Equation 12	Actual Cut-off (Equation 21)
2	795.7747155	1150.267281
4	397.8873577	461.0511704
8	198.9436789	212.8383013
16	99.47183943	102.7519182
32	49.73591972	50.53386767
64	24.86795986	25.0648066
128	12.43397993	12.48286862
256	6.216989965	6.229172189

Appendix B

The settling time of the single-pole digital IIR filter can be derived from Equation 11, which is given below

$$\frac{V_{lp}}{V_{in}} = \frac{1}{a - z^{-1}(a-1)}$$

Equation 24

If you rearrange Equation 9, you get Equation 25, which is the frequency response of a digital low-pass filter with attenuation factor, a.

$$H(z) = \left(\frac{1}{a}\right)\frac{1}{1 - z^{-1}\left(1 - \frac{1}{a}\right)}$$

Equation 25

If you take inverse z-transform, you get the impulse response of the filter (h[n]).

$$h[n] = \left(\frac{1}{a}\right)\left(1 - \frac{1}{a}\right)^{n}$$

Equation 26

Summation of the impulse response over 0 to n samples gives the unit step response (s[n]).

$$s[n] = \left(\frac{1}{a}\right)\sum_{0}^{n}\left(1 - \frac{1}{a}\right)^{n}$$

Equation 27

Simplifying Equation 27, we get Equation 28

$$s[n] = 1 - \left(1 - \frac{1}{a}\right)^{n}$$

Equation 28

Document History

Document Title: PSoC® 1, PSoC 3, and PSoC 5 - Single-Pole Infinite Impulse Response (IIR) Filters – AN2099

Document Number: 001-38007

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	1520284	DWV	10/01/2007	Old application note updated to CY web.
*A	2711571	YARA	05/27/2009	Added CY8C29x66, CY8C27x43, CY8C24x23A, CY8C24x94, CY8C21xxx, and CY8C20xxx part families. Updated Software version to PD 5.0.
*B	3248285	DSG	05/04/2011	Added section Implementation in C
*C	3394927	PFZ	10/12/2011	Major rewrite of the document
*D	3457966	PFZ	12/09/2011	Template Update Updated project for PSoC Creator 2.0
*E	3492061	PFZ	01/11/2012	Fixed Table 2.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc cypress.com/go/plc
Memory	cypress.com/go/memory
Optical Navigation Sensors	cypress.com/go/ons
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 5

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Xx and xx are registered trademarks of Cypress Semiconductor Corp. All other trademarks or registered trademarks referenced herein are the property of their respective owners.

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709

Phone : 408-943-2600
Fax : 408-943-4730
Website : www.cypress.com

© Cypress Semiconductor Corporation, 2007-2012. The information contained herein is subject to change without notice. Cypress Semiconductor Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in a Cypress product. Nor does it convey or imply any license under patent or other rights. Cypress products are not warranted nor intended to be used for medical, life support, life saving, critical control or safety applications, unless pursuant to an express written agreement with Cypress. Furthermore, Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress products in life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

This Source Code (software and/or firmware) is owned by Cypress Semiconductor Corporation (Cypress) and is protected by and subject to worldwide patent protection (United States and foreign), United States copyright laws and international treaty provisions. Cypress hereby grants to licensee a personal, non-exclusive, non-transferable license to copy, use, modify, create derivative works of, and compile the Cypress Source Code and derivative works for the sole purpose of creating custom software and or firmware in support of licensee product to be used only in conjunction with a Cypress integrated circuit as specified in the applicable agreement. Any reproduction, modification, translation, compilation, or representation of this Source Code except as specified above is prohibited without the express written permission of Cypress.

Disclaimer: CYPRESS MAKES NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Cypress reserves the right to make changes without further notice to the materials described herein. Cypress does not assume any liability arising out of the application or use of any product or circuit described herein. Cypress does not authorize its products for use as critical components in life-support systems where a malfunction or failure may reasonably be expected to result in significant injury to the user. The inclusion of Cypress' product in a life-support systems application implies that the manufacturer assumes all risk of such use and in doing so indemnifies Cypress against all charges.

Use may be limited by and subject to the applicable Cypress software license agreement.