**Please note that Cypress is an Infineon Technologies Company.**
The document following this cover page is marked as “Cypress” document as this is the company that originally developed the product. Please note that Infineon will continue to offer the product to new and existing customers as part of the Infineon product portfolio.

**Continuity of document content**
The fact that Infineon offers the following product as part of the Infineon product portfolio does not lead to any changes to this document. Future revisions will occur when appropriate, and any changes will be set out on the document history page.

**Continuity of ordering part numbers**
Infineon continues to support existing part numbers. Please continue to use the ordering part numbers listed in the datasheet for ordering.

# THIS SPEC IS OBSOLETE

Spec No: 001-32902

Spec Title: BINARY WEIGHTED SINGLE-POLE IIR LOW-PASS FILTERS IN PSOC(R) 1 - AN2276

Sunset Owner: Meenakshi Sundaram Ravindran [msur]

Replaced by: 001-38007

# Binary Weighted Single-Pole IIR Low-Pass Filters in PSoC® 1

# AN2276

**Author:** Dave Van Ess
**Associated Project:** Yes
**Associated Part Family**: Any PSoC® 1 Parts
**Software Version**: PSoC Designer™ 5.1
**Associated Application Notes**: AN2099



## Abstract

AN2276 describes how to implement binary weighted single pole infinite impulse response (IIR) low-pass filters in PSoC® 1. Many applications require filtering data after it is in a sampled digital form. This was previously discussed in Application Note *AN2099 – PSoC® 1/3/5 - Single-Pole IIR Filters: To Infinity And Beyond*. A brief review of low-pass IIR filters is given. Equations are developed and software is presented to implement this topology giving the user access to filter routines in either assembly or 'C'. PSoC 3 and 5 are capable of implementing digital filters in their dedicated digital filter block (DFB), and are not covered in this application note.

## Introduction

A single-pole IIR low-pass filter is easy to implement. It requires:

- Subtraction
- Division
- Addition

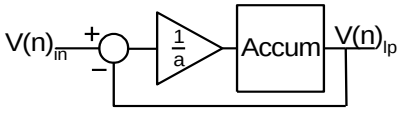
This was covered in Application Note *AN2099 – PSoC®1/3/5 - Single-Pole IIR Filters: To Infinity And Beyond* and should be reviewed by the reader prior to reading this application note.

For special cases, where the divisor is a binary value ($2^n$), the division can be reduced to a number ( $n$ ) of shifts.

## Infinite Impulse Response Filters

Figure 1 shows the topology for an IIR low-pass filter. In this type of filter, the current output is made up of the previous output attenuated by some amount, plus the current input attenuated by some amount.

Figure 1. IIR Topology for Low-Pass Filter




Equation 1 defines the system while Equation 2 defines the transfer function:

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

$$\frac{V_{lp}}{V_{in}} = \frac{1}{1 + a(z-1)} \quad \text{Equation 2}$$

The conversion from a $z$ transform to a Laplace transform requires Equation 3:

$$z = e^{\frac{s}{f_s}} \cong 1 + \frac{s}{f_s} \quad \{f_s := SampleRate\} \quad \text{Equation 3}$$

Substituting Equation 3 into Equation 2 results in the Laplace transfer Equation 4:

$$\frac{V_{lp}}{V_{in}} = \frac{1}{1 + s\frac{a}{f_s}} \quad \text{Equation 4}$$

The roll-off frequency $f_0$ is shown in Equation 5:

$$f_0 = \frac{f_s}{2\pi a} \quad \text{Equation 5}$$

The roll-off frequency is dependent on the sample frequency $f_s$, but more importantly, the attenuation value $a$ . Merely changing the attenuation value easily changes the filter's roll-off frequency.

If the attenuation value is limited to a binary value ($2^n$), the division operation is simplified and can be implemented with a series ( $n$ ) of right shifts.

## An IIR Low-Pass Filter in Three Easy Steps

The three steps to implement the filter just described are listed as follows:

1. Subtract the old output signal from the input signal.

$$v_{in} - v_{out} z^{-1} \qquad \text{Equation 6}$$

2. Shift right $n$ times.

$$\frac{v_{in} - v_{out} z^{-1}}{2^n} \qquad \text{Equation 7}$$

3. Add to old output.

$$v_{out} z^{-1} + \frac{v_{in} - v_{out} z^{-1}}{2^n} \qquad \text{Equation 8}$$

Two examples of this type of filter are shown. Both have an attenuation value of 4 (achieved with 2 bitwise shifts to the right). *cLowPass4th* implements a filter for 8-bit signed values and is located in *cLowPass.asm*. *iLowPass4th* implements a filter for 16-bit signed values and is located in *iLowPass.asm*. The functions *SetAccum_for_cLowPass* and *SetAccum_for_iLowPass* are included to initialize the filters' accumulators. *cLowPass.h and iLowPass.h* are also included to allow 'C' programs access to these functions. All four of these files are located in the project file associated with this Application Note. This project was implemented using the CY8C27xxx base part, but it easily can be cloned to any member of the PSoC 1 family.

The function *cLowPass4th* requires 101 CPU cycles. This is shown in Code 1.

Code 1. The Function cLowPass4th

```
;;-------------------------------------
;;  char cLowPass4th(int)
;;  iAcumm =  iAccum + (Input-iAccum)/4
;;  Input  = A
;;  Output = A
;;-------------------------------------
 cLowPass4th:
_cLowPass4th:
   RAM_PROLOGUE RAM_USE_CLASS_4
   RAM_SETPAGE_CUR (>iAccum)
   mov  X,0              ;cTemp = Input
   swap A,X
   sub  A,[cAccum + 1]   ;cTemp -= iAccum
   swap A,X
   sbb  A,[cAccum]

   asr  A                ;cTemp >>
   swap A,X
   rrc  A
   ;one section per addition shift;
   swap A,X              ;cTemp    ;
   asr  A                          ;
   swap A,X                        ;
   rrc  A                          ;
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   adc  [cAccum + 1],A  ;cAccum += iTemp
   swap A,X
   adc  [cAccum],A

   mov  A,[cAccum + 1]  ;return(cAccum)
   add  A,80h
   mov  A,[cAccum]
   adc  A,0
   RAM_EPILOGUE RAM_USE_CLASS_4
   ret
```

The function *iLowPass4th* requires 182 CPU cycles as shown in Code 2.

Code 2. The function *iLowPass4th*

```
;---------------------------------------
;   int iLowPass4th(int)
;   iAcumm =  iAccum + (Input-iAccum)/4
;   Input = X,A
;   Output = X,A
;---------------------------------------
 iLowPass4th:
_iLowPass4th:
   RAM_PROLOGUE RAM_USE_CLASS_2
   RAM_PROLOGUE RAM_USE_CLASS_4
   RAM_SETPAGE_CUR (>lAccum_)
   push X                ;&iTemp = Input
   mov  X,SP
   push A
   mov  A,0

   sub A,[iAccum + Residue] ;iTemp-=iAccum
   push A
   mov  A,[iAccum + LowByte]
   sbb  [X +LowByte],A
   mov  A,[iAccum + HighByte]
   sbb  [X + HighByte],A
   pop  A
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   asr  [X + HighByte]         ;iTemp >>
   rrc  [X + LowByte]
   rrc  A
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   ;;one for each shift;;;;;;;;;;;;;;;;;;
   asr  [X + HighByte]         ;iTemp >> ;
   rrc  [X + LowByte]                    ;
   rrc  A                                ;
   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
   adc  [iAccum +Residue],A ;iAcumm+= iTemp
   pop  A
   adc  [iAccum + LowByte],A
```

```
    pop  A
    adc  [iAccum + HighByte],A

    ;return(iAccum)
    mov  A,[iAccum + Residue]
    add  A,80h
    mov  A,[iAccum + LowByte]
    adc  A,0
    swap A,X
    mov  A,[iAccum + HighByte]
    adc  A,0
    swap A,X
    RAM_EPILOGUE RAM_USE_CLASS_4
    RAM_EPILOGUE RAM_USE_CLASS_2
    Ret
```

The code to test these functions is shown in Code 3.

Code 3. IIR Test Program

```
//-------------------
// IRR Test Program
//-------------------
#include <m8c.h>
#include "PSoCAPI.h"
#include "cLowPass.h"
#include "iLowPass.h"

int  iTemp;
char cTemp;
long lTemp;

void main()
{
   lTemp = lAccum_for_iLowPass;
   SetAccum_for_cLowPass(0);
   SetAccum_for_iLowPass(0);
   while(1){
      iTemp = iLowPass4th(512);
      cTemp = cLowPass4th(100);
   }
}
```

## Build Your Own

Both filters shown have attenuation of 4 and, by definition, a roll-off frequency approximately 4% of the sample frequency, as shown in Equation 5. If a steeper roll off is required, the attenuation needs to be increased. For this example, the *iLowPass4th* function is used to construct an *iLowPass8th* function. It is done in four steps.

1. Open *iLowPass.h*. The first two lines are shown as follows:

   ```
   #pragma fastcall16 iLowPass4th
   extern int iLowPass4th(int iSample);
   ```

   Duplicate these two lines and change the names from *iLowPass4th* to *iLowPass8th* as follows:

   ```
   #pragma fastcall16 iLowPass4th
   extern int iLowPass4th(int iSample);
   #pragma fastcall16 iLowPass8th
   extern int iLowPass8th(int iSample);
   ```

   This allows the ‘C’ programs access to the new function.

2. Open *iLowPass.asm* and find the two lines shown as follows:

   ```
   export  iLowPass4th
   export _iLowPass4th
   ```

   Again, duplicate these lines and change the name to match the new function. The result is shown as follows:

   ```
   export  iLowPass4th
   export _iLowPass4th
   export  iLowPass8th
   export _iLowPass8th
   ```

   This allows all programs access to this new function.

3. Duplicate the function and change the names from *iLowPass4th* to *iLowPass8th*. The top 8 lines of the duplicated function are shown as follows:

   ```
   ;-----------------------------------
   ;   int iLowPass4th(int)
   ;   iAcumm =  iAccum + (Input-iAccum)/4
   ;   Input = X,A
   ;   Output = X,A
   ;-----------------------------------
    iLowPass4th:
   _iLowPass4th:
   ```

   The change is shown as follows:

   ```
   ;-----------------------------------
   ;   int iLowPass8th(int)
   ;   iAcumm =  iAccum + (Input-iAccum)/8
   ;   Input = X,A
   ;   Output = X,A
   ;-----------------------------------
    iLowPass8th:
   _iLowPass8th:
   ```

The new function now exists, however its operation has not been altered.

4. Find the following lines in the new function:

```
;;one for each shift;;;;;;;;;;;;;;;;;;
asr   [X + HighByte]          ;iTemp >> ;
rrc   [X + LowByte]                     ;
rrc   A                                 ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Duplicate as follows:

```
;;one for each shift;;;;;;;;;;;;;;;;;;
asr   [X + HighByte]          ;iTemp >> ;
rrc   [X + LowByte]                     ;
rrc   A                                 ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;one for each shift;;;;;;;;;;;;;;;;;;
asr   [X + HighByte]          ;iTemp >> ;
rrc   [X + LowByte]                     ;
rrc   A                                 ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

The new function has been successfully implemented. It now shifts the difference three times for an attenuation value of 8. Its roll-off frequency is now approximately 2% of the sample frequency, or half the frequency for the attenuation 4 IIR filter. The implementation of filters with even greater attenuation is an exercise left to the reader.

## The Whole is Greater Than the Sum of the Parts

Back in college, you most likely learned that averaging a number of signals together caused the noise to be reduced by the square root of the number of samples. When signal are averaged, it is done in a linear fashion. The results are shown in Equation 9:

$$Signal_{avg} = \sum_{m=0}^{a-1} \frac{Signal}{a} = Signal \qquad \text{Equation 9}$$

However, noise being random, averages are in an RMS fashion. This is shown in Equation 10:

$$Noise_{avg} = \sqrt{\sum_{m=0}^{a-1} \left(\frac{Noise}{a}\right)^2} = \frac{Noise}{\sqrt{a}} \qquad \text{Equation 10}$$

So averaging four samples should reduce the noise by a factor of 2.

For an IIR filter, again the signals add linearly. The results are shown in Equation 11:

$$Signal_{avg} = \frac{Signal}{a} + \frac{(a-1)\cdot Signal_{avg}}{a} = Signal \qquad \text{Equation 11}$$

However, the noise, still random, adds in an RMS fashion. The results are shown in Equation 12:

$$Noise_{avg} = \sqrt{\left(\frac{Noise}{a}\right)^2 + \left(\frac{(a-1)\cdot Noise}{a}\right)^2} = \frac{Noise}{\sqrt{2a-1}} \qquad \text{Equation 12}$$

So processing the samples with an IIR filter, with an attenuation of 4, reduces the noise by a factor of 2.65. Clearly, this is an improvement over averaging four signals. Seven samples would have to be averaged to get the equivalent reduction.

What this boils down to is that the data, when filtered, may pick up resolution. It is possible that it may pick up more resolution than the function type normally returns.

To remedy this, the Accumulator, which is really the filter output for *iLowPassFilter*, is actually a 32-bit wide variable. The next line of code allows 'C' access to this variable.

```
extern long lAccum_for_iLowPass
```

And the following code allows assembly language access to this same variable.

```
export   lAccum_for_iLowPass
export  _lAccum_for_iLowPass
area bss(RAM)
   lAccum_for_iLowPass:
  _lAccum_for_iLowPass:  BLK  4
area text(ROM,REL)
```

For the *cLowPassFilter*, the Accumulator is actually a 16-bit wide variable. Its declaration is shown as follows:

```
extern int iAccum_for_cLowPass;
```

Of course, capabilities exist for assembly language access to this variable.

The user has two options to retrieve the filter data.

1. Use the output returned by the function call. It is the same resolution as the input data.
2. Call the function and ignore the returned output. Retrieve data in the Accumulator variable. This variable is twice as wide as the input variable, but it now contains the extra resolution.

## Additional Resources

Included with this application note's project is an excel spreadsheet demonstrating the advantage of an IIR filter over a mere moving average filter.

## Summary

Single-pole IIR filters are very useful in reducing signal noise and are more effective than just averaging the equivalent number of samples.

Filters for both 8-bit data (*cLowPass4th*) and 16-bit data (*iLowPass4th*) have been presented. Instructions have been given to modify these filters for greater attenuation.

Low-pass filters may increase the resolution of the data and methods have been shown to retrieve this extra resolution.

## About the Author

**Name**: Dave Van Ess
**Title**: Applications Engineer MTS
Cypress Semiconductor
**Background**: BSEE from University of California, Berkeley.
More than 27 Years experience in circuit, signal processing, digital, software, analog, and system design. Holder of six U.S. Patents (plus three pending) for medical systems, signal processing, and digital block enhancements. Author of numerous Application Notes, web casts, and technical articles.
Joined Cypress MicroSystems in 2000.
**Contact**: dwv@cypress.com

# Document History

**Document Title: Binary Weighted Single-Pole IIR Low-Pass Filters in PSoC® 1**

**Document Number: 001-32902**

| Revision | ECN | Orig. of Change | Submission Date | Description of Change |
|---|---|---|---|---|
| ** | 1494923 | MAXK | 09/21/2007 | OLD APP. NOTE: Obtain spec. # for note to be added to spec. system. Update copyright. Add source disclaimer, revision disclaimer, Samples Request Form link, PSoC App. Note Index link. .pdf has been stamped.<br><br>**This note had no technical updates. There is an associated project but it was not updated.** |
| *A | 3187213 | MAXK | 03/03/2011 | Updated title as “AN2276 - Binary Weighted Single-Pole IIR Low-Pass Filters in PSoC® 1”.<br>Updated Associated Part Family in page 1 as “Any PSoC 1 Parts”.<br>Updated Software Version as “PSoC Designer™ 5.1”.<br>Updated Abstract.<br>Updated Infinite Impulse Response Filters.<br>Updated An IIR Low-Pass Filter in Three Easy Steps.<br>Updated The Whole is Greater Than the Sum of the Parts.<br>Added Additional Resources (information about Excel spreadsheet). |
| *B | 3281359 | MAXK | 06/13/2011 | No change. Sunset review spec.<br>Removed application note number from the title. |
| *C | 4371618 | MSUR | 05/06/2014 | Obsolete document |

In March of 2007, Cypress recataloged all of its Application Notes using a new documentation number and revision code. This new documentation number and revision code (001-xxxxx, beginning with rev. **), located in the footer of the document, will be used in all subsequent revisions.



Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone: 408-943-2600
Fax: 408-943-4730
http://www.cypress.com/