TI-PA Filter Library / Framework (FIL)

Rajesh Vanga *Performance Audio - EEE*

ABSTRACT

Digital filters are the building blocks of Digital audio processing. TI-PA Filter Library/Framework (FIL) aims at providing a library of functions for programmers to develop and implement memory/MIPS efficient floating-point audio filters for the TMS320DA6xx, TMS320DA7xx and TMS320DA8xx family of devices.

This document describes how a programmer can use the TI-PA Filter Library/Framework (FIL) for developing algorithms that use filters.

Contents

Tables

1 Introduction

Digital filters are the basic blocks of Audio Signal Processing and TI-PA Filter Library/Framework, abbreviated as “FIL”, aims at providing an easy tool/library for programmers to develop and implement memory/MIPS efficient filter based audio algorithms. FIL includes a code library of filter implementations for TMS320DA6xx, TMS320DA7xx, and TMS320DA8xx platforms written in C, serial assembly, and hand assembly; optimized for single and double precisions. FIL has a filter independent generic-interface, provided in two layers: XDAIS-ASP interface layer and simple filter function-call layer. These are explained in detail. FIL control code takes care that it calls the best filter implementation that is possible, with the given filter parameters. This helps to hide TI-PA filter library specific details from the programmer.

2 Getting started

It is assumed that the user is familiar with various filter terminologies and XDAIS algorithm interface (Reference 3). The XDAIS-ASP layer of FIL is meant for algorithm development in TI-PA Framework only. The lower filter layer can be used independently.

The user is expected to have the following:

- Texas Instruments C6000 PC Code Composer Studio IDE version 2.20.14 for DA6xx, version 3.1 for DA7xx, version 3.3 or version4 for DA8xx and supplementary materials.
- FIL interface files, FIL Library which are part of package provided.

3 FIL interface

We have two interfacing layers for using FIL filter implementations. This gives the application programmer the choice to use the appropriate layer. The lower filter layer is for users who are not interested in XDAIS-ASP interface and automatic filter selection:

Layer I : XDAIS-ASP

Layer II: Filter

3.1 Layer-1 (XDAIS-ASP interface layer)

Layer-1 of FIL module follows the rules of ASP and XDAIS. It can be called following the steps given in the simple example given below :

```
#include "fil.h"
#include "fil_tii.h"

volatile int FilterEnable;
void main(void)
{
    int reset = 0;
    FIL_Handle filHandle;

    FIL_init();

    if((filHandle = FIL_create(&FIL_TII_IFIL, (FIL_Params *)&FIL_USER_PARAMS) ) != 0)
    {
        while(1)
```

```
        {
            if(FilterEnable)
            {
                if(reset)
                    FIL_reset(filHandle, &pAudioFrame);
                reset = 0;
                FIL_apply(filHandle, &pAudioFrame);
            }
            else
                reset = 1;
        }
    }
    FIL_delete(filHandle);
    FIL_exit();
}

// main()          – The user application that uses FIL.
// FIL_TII_IFIL    – The algorithm handle of FIL, that has all FIL function pointers.
// FIL_USER_PARAMS – User handle of initial values, used while the FIL handle is created.
// filHandle       - Handle for instantiated filter object.
// pAudioFrame     – This is the pointer to the Audio-frame handle of PA-framework.
```

In the above example, a filter object is instantiated and a handle (e.g. filHandle) is maintained for each filter object. The entry points into the library are explained below:

- FIL_init() : Algorithm initialization is done here. FIL currently does nothing in this function.
- FIL_create() : Memory allocation for the filter object and initialization of filter object handle with the values given through the function arguments is done here. All the FIL algorithm function pointers are passed through the first argument (&FIL_TII_IFIL) and all user-filter initial parameters are passed through the second argument (&FIL_USER_PARAMS). This function returns the pointer of the created and initialized filter object handle. The structure of the second argument (&FIL_USER_PARAMS) is explained below.
- FIL_delete() : This function does the de-allocation of the filter object and all the memory allocated in the FIL_create() function. The filter object handle is passed as the argument.
- FIL_exit() : Called during system shutdown, to perform any run-time finalization.
- FIL_apply() : This function internally calls the FIL control code, which finally calls the appropriate optimized filter routines. First argument is the filter object handle and the second is the PA-Framework handle, which contains information regarding the input audio stream to be processed.
- FIL_reset() : This function is used to “reset” the filter state memory, which is normally required when there is some filtering discontinuity.

  The structure of the global variable FIL_USER_PARAMS in the above example, i.e. the handle given by the user for filter object initialization, is explained below :

```
typedef struct IFIL_Params {
    Int size; // Size of the structure itself, in bytes, i.e. sizeof(IFIL_Params)
    Int use;
    IFIL_Status  *pStatus;
    IFIL_Config  *pConfig;
} IFIL_Params;
```

- *use* - Global enable/disable control register; **0** : disable filtering, **1** : enable filtering. This is initialized by the initial value given during filter object creation.
- *pStatus* - Pointer to the “status structure” initial values.
- *pConfig* - Pointer to the “config structure” initial values.

```
typedef volatile struct IFIL_Status {
    Int size; // Size of the structure itself in bytes, i.e. sizeof(IFIL_Status)
    PAF_ChannelMask mode;
    XDAS_Int16  use;
    PAF_ChannelMask maskSelect;
    PAF_ChannelMask maskStatus;
} IFIL_Status;
```

- *mode* – This is a bit field, generally 16 bits, which mentions, in bit wise, all the possible channels that may be filtered. This is taken as a channel-select template, which should be always the superset to ‘maskSelect’ bit field (explained). Memory allocation for the filter, during filter object creation, is done against this template. This should not be changed after object creation.

  E.g. *mode* s= 0000 0000 0110 1001b, i.e. channels 0,3,5 and 6 may be requested to be filtered. Memory allocation will be done only for these channels.

- *use* - Global enable/disable control register; **0** : disable filtering, **1** : enable filtering. This is initialized by the initial value given during filter object creation.
- *maskSelect* – This is a bit field, generally 16 bits, used to request which all channels are to be filtered. Bit(***i***) = 1 means to apply the given filter on channel ‘***i*** ’. This field has to be a subset to the selected channels mentioned in *mode* field and can be changed according to the requirement, after initiation.
- *maskStatus* – This is to report to the user which all channels had been actually filtered. If filtering was not done, it will be reset to zero. The field is read only, i.e. the user is not allowed to change its value, other than while initializing.

```
typedef struct IFIL_Config {
    Int size; // Size of the structure itself in bytes, i.e. sizeof(IFIL_Config)
    const PAF_FilCoef  * pCoefs;
    Void *pVars;
 } IFIL_Config;
```

- *pCoefs* – This is the pointer to the filter specific coefficient/parameter structures and is explained later. ‘PAF_FilCoef’ structure is explained below.
- *pVars* – This is the pointer to filter state/delay memory. This field is initialized by FIL during FIL_create().

```
typedef struct PAF_FilCoef  {
    Uint type;
    Uint sampleRate;
 } PAF_FilCoef ;
```

Every filter coefficient/parameter structure is expected to have the above two fields, filter type and sample rate, as their first two elements. The remaining coefficient structure, that follows these two elements, is explained in the respective filter appendices.

- "*type*" is a bit field, of length 32 bits, that briefly specifies the filter type and possibly few other filter dependent parameters which are explained in respective filter appendices. "Type" field has enough information to allocate memory for the filter.

MS = Multi-Sample rate(1b) **P= Coeff Precision(2b)** **G = Filter Group(6b)**
SG = Filter Sub-Group(4b) **UC = Uni-Coefficient (1b)** **S = Filter Specs(18b)**

MS	P	P	G	G	G	G	G	G	SG	SG	SG	SG	UC	S17	S16

S15	S14	S13	S12	S11	S10	S9	S8	S7	S6	S5	S4	S3	S2	S1	S0

  - **MS** : Whether the coeff. structure contains different coefficients for different sampling rates (1) or not (0).
  - **P** : We have 2 bits to specify the data type of the coefficients.(DP=00b, SP=01b)
  - **G** : Mentions the basic group of filter.

    E.g. Impulse Response (IR) filter group – 000 000b
  - **SG** : Gives the subgroup of the filter.

    E.g. Direct Form 2 Infinite Impulse Response (IIR) filter subgroup – 0000b

    E.g. Finite Impulse Response (FIR) filter subgroup – 0001b

    E.g. Direct Form 1 Infinite Impulse Response (IIR) filter subgroup – 0010b
  - **UC** : Says whether a single filter coefficient set is applied to all the channels (1) or not (0). If '0', it implies that the coefficient structure has a separate coefficient set for each channel. Then the number of coefficient sets equals the number of 1's in 'mode' field of the IFIL_Status structure and the coefficient sets are packed in the same order as the 1's in the mode, starting from the right (LSB) to left(MSB).
  - **Sn** : Filter specifications. This field is used to specify additional filter parameters and its interpretation is filter specific.

- **"***sampleRate***"** is a 32 bit field to specify whether coefficients are present for a particular sampling rate. Bit(***i***) = 1 specifies that a coefficient set is provided for the sampling rate "***i*** ". The sampling rates and bit number '**i**' are related as given below:

Bit(i)	SampRate
0	Unknown
1	None
2	32kHz
3	44.1kHz
4	48kHz
5	88.2kHz
6	96kHz

7	192kHz
8	64kHz
9	128kHz
10	176.4kHz
11	8kHz
12	11.025kHz
13	12kHz
14	16kHz
15	22.05kHz
16	24kHz

Note:

- If MS = 0: the first coefficient set is used irrespective of input data stream sample rate.
- If MS = 1:
  - If the sample rate of the stream is 'none', filtering is not performed
  - If coefficients are not provided for the sample rate of the stream, then filtering is not performed
  - If the sample rate of the stream is 'unknown', then the stream is filtered using the 48kHz coefficients

Note: Layer-1 (XDAIS-ASP layer) user can skip section 3.2, Layer-2(filter layer), details and continue from section 4, FIL features.

3.2 Layer-2 (Filter layer)

This is the direct filter function call layer. The filter function prototype for an arbitrary filter 'xyz' will be as given below:

```
Int Filter_xyz( PAF_AudioFilterParam * param); // Returns '0' if success.
```

"param" is a pointer to the lower layer generic filter handle structure "PAF_AudioFilterParam".

```
typedef struct PAF_AudioFilterParam {
    Void  ** pIn ;
    Void  ** pOut ;
    Void  ** pCoef ;
    Void  ** pVar ;
    Uchar    channels;
    Int      sampleCount;
    Uint     use;
}  PAF_AudioFilterParam;
```

Let N be the maximum number of channels that are processed at a time.

- *pIn* – Pointer to the array of 'input channel data' pointers.

  pIn → [0] → [$X_0(n)$] , where X(n) is the input data block, of length "sampleCount"
  [1] → [$X_1(n)$]
  ....
  [N-1] → [$X_{N-1}(n)$]

- *pOut* - Pointer to the array of 'output channel data' pointers.

pOut → [0] → [$Y_0(n)$] , where Y(n) is the output data block, of length "sampleCount"
[1] → [$Y_1(n)$]
....
[N-1] → [$Y_{N-1}(n)$]

- *pCoef* - Pointer to the array of 'channel coefficient' pointers.

pCoef → [0] → [$C_0(n)$] , where C(n) is the coefficient sequence of a channel.
[1] → [$C_1(n)$]
....
[N-1] → [$C_{N-1}(n)$]

- *pVar* - Pointer to the array of 'channel state/private memory' pointers.

pVar → [0] → [$m_0(n)$] , m(n) is the memory sequence belonged to a channel.
[1] → [$m_1(n)$]
....
[N-1] → [$m_{N-1}(n)$]

- *channels* – Specifies the number of channels that are to be filtered, starting sequentially from channel 0.
- *sampleCount* – Specifies the sample length of the input data block.
- *use* – This is a 32 bit field that can be used, to fit in some extra parameters or as a pointer to additional filter parameters. The exact interpretation is filter dependent and is explained in the respective filter appendix.

3.2.1 Memory allocation

The user of the lower filter layer has to take care of allocating and initializing memory for,

a) The state/delay memory required for the filter

b) The lower layer filter object handle of structure type 'PAF_AudioFilterParam' , and

c) The pIn[N], pOut[N], pCoef[N] and pVar[N] pointer arrays, which the filter handle elements point to.

Allocation for state/delay memory

Int FIL_varsPerCh(Uint type)

The function FIL_varsPerCh ( ) returns the memory requirement for a particular filter, in bytes per channel. This is the size of private memory that a channel requires, i.e. size of array $m_x(n)$ given above. It takes the "type" field of the coefficient structure, as argument and calculates the memory per channel. Thus the user of the inner layer need not worry about the internals of filter implementation and memory requirements.

Void FIL_memReset(Void *varPtr, Int size)

The function FIL_memReset() can be used to clear Allocated memory for state/delay memory. This function takes void pointer and size as input and clears the buffer pointed by pointer varPtr for number of bytes indicated by size.

E.g. 512 tap SP FIR filtering of 3 channels : In this case, FIL_varsPerCh ( ) will return 512*sizeof(Float)=2048, which is the delay memory requirement for one channel.

Allocation for filter object handle structure, PAF_AudioFilterParam

Memory required = sizeof(PAF_AudioFilterParam).

Allocation for the pointer arrays, pIn[N], pOut[N], pCoef[N] and pVar[N]

Each of them has a length = N * sizeof(void *).

Note: The pointers must be initialized to appropriate base addresses, before passing it to the filter function.

4 FIL features

4.1 Filter states

When using the XDAIS-ASP layer, the ASP layer control code will automatically allocate memory required for the filter states. The filter states are initially set to '0', while allocating. Further, the states are updated and maintained, following the rules of the corresponding filter. The filter implementation structure used will be optimized for memory. The filter state structures are explained in detail in the respective filter appendices.

4.2 Filter implementation

The filters are implemented and optimized to obtain the best performance possible. They have been designed for systems where the input is available in blocks and allows in-place processing too. The source code may be in C (*.c), serial assembly (*.sa) or in hand assembly (*.asm), and is written for TMS320DA6xx, TMS320DA7xx, TMS320DA8xx floating point DSP platform. The filters available in FIL are explained in the respective appendices.

4.3 Tracking the filtering process

FIL_apply( ) returns zero upon successful filter operation. Anything return value other than zero should be considered as an error. Furthermore, there could be instances where FIL passes its input to output without processing and returns "FIL_SUCCESS". This could be tracked by looking into the *maskStatus* element of the *IFIL_Status* structure. Reading the IFIL_Status structure elements can do further probing.

Appendix A. IR filter group

This appendix explains Impulse Response filter group specific. It includes details of the coefficient structure, 'type' field, filter implementation & performance, together with a few simple illustrative examples. For working examples, refer to a provided Audio Example that uses FIL.

Coefficient structure

As we have seen, XDAIS-ASP coefficient structure should have two header elements, 'type' and 'sampleRate'. So, extending the structure for IR filter group, we append a 'unit length' pointer array, which will be the first element of the array of coefficient pointers. This 'unit length' array could be useful to access the remaining coefficient pointers by proper offsetting. The data type of this pointer depends on the coefficient data type, as shown below. The coefficient data type PAF_AudioData must be considered equivalent to float unless otherwise stated.

Single Precision coefficient structure:

```
typedef struct {
    Uint  type;
    Uint  sampleRate;
    Float *cPtr[1];
} PAF_FilCoef_SP;
```

Double Precision coefficient structure:

```
typedef struct {
    Uint   type;
    Uint   sampleRate;
    Double *cPtr[1];
} PAF_FilCoef_DP;
```

PA-Framework data type coefficient structure:

```
typedef struct {
    Uint   type;
    Uint   sampleRate;
    PAF_AudioData *cPtr[1];
} PAF_FilCoef_PAF;
```

The 'array of coefficient pointers' for a multi-channel, multi-rate coefficient structure is given below. The coefficient pointers for the specified channels (in *mode* field of *IFIL_Status*) and sample rates (in *sampRate* field of *PAF_FilCoef*) are packed and arranged, with channels row wise and sample rates column wise. Thus, for a uni-coefficient case, number of columns will reduce to 1 and for a single sample rate case, the number of rows will reduce to 1.

```
/*      Ch : 1            Ch : 2      ....      Ch : M                          */
{
   {&cCh1_SmpRate1[0], &cCh2_SmpRate1[0], …. &cChM_SmpRate1[0] }, /*Sample Rate: 1*/
   {&cCh1_SmpRate2[0], &cCh2_SmpRate2[0], …. &cChM_SmpRate2[0] }, /*Sample Rate: 2*/
      ….                ....                    ....              ...
   {&cCh1_SmpRateN[0], &cCh2_SmpRateN[0], …. &cChM_SmpRateN[0] }, /*Sample Rate: N*/
}
```

Where for,n tap IIR,

cChM_SmpRateN[ ] = { b0, b1 ... bn, // Feed forward coefficients
a1, a2 ... an } // Feed back coefficients

and n tap FIR,

cChM_SmpRateN[ ] = { b0, b1 ... bn } // Feed forward coefficients

Filter Type field

IIR :

MS	P	P	0	0	0	0	0	0	0	0	DF1	0	UC	X	X

mx1	tap	Tap	tap	tap	tap	tap	tap	Tap	tap	tap	ch	ch	ch	ch	ch

mx1 – The mx1 bit is to Indicate the Mixed Precision1 (SP-DP) type of filters,1 in this field will select mx1 filters for other filters (which includes SP and mx(DP-DP)) this should be zero.

DF1 – The df1 bit indicates whether the IIR implementation type is direct form 1 or direct form 2. 1 means direct form 1, and 0 means direct form 2.

Note: The mx1 (SP-DP) Filters are available only on DA7xx DSP's.

FIR :

MS	P	P	0	0	0	0	0	0	0	0	0	1	UC	X	X

X	tap	Tap	tap	tap	tap	tap	Tap	Tap	tap	tap	ch	ch	ch	ch	ch

ch – This specifies the number of channel for which coefficients are provided. (ch = 1 to 31)

tap – This specifies the tap or order of the filter.(tap = 1 to 1023)

Layer-2 interface

In the FIL layer-2 handle structure "PAF_AudioFilterParam", LSB 16 bits of "use" element is used to specify the taps or order of the filter.

Filter core implementation and performance

IIR – FIL implements a set of IIR filters for various combinations of taps and concurrently processed channels, with inputs and outputs in SP and coefficients and processing in SP or DP. Filters outside this set are either derived from the existing ones or use the generic N tap-N channel implementation. The table of IIR filter implementations, with their performances and "layer-2" function names, are given in the table below. The given performance numbers are the CPU clocks taken for processing a single audio sample of a single channel.

Table 1. DF2 IIR table, Single Precision

IIR	Single Precision		General		UniCoef		UniCoef/InPlace	
Tap	Ch	Available	Base function name - Filter_iir"**ABC**"( )					
			Clks/Sample/Ch on TMS320DA6xx		Clks/Sample/Ch on TMS320DA7xx			
1 Tap	1	Yes	T1Ch1					
			8	8				
	2	Yes	T1Ch2					
			4	4				
	3	Yes	T1Ch3					
			2.7	2.7				
	4	Yes	T1Ch4					
			2.5	2.5				
	5 = 3+2	Derived	T1ChN		T1Ch5_u			
			3.2	3.2	1.6	1.6		
	6 = 3+3	Derived	T1ChN				T1Ch6_ui	
			2.7	2.7			1.5	1.5
	7 = 4+3	Derived	T1ChN		T1ChN		T1Ch7_ui	
			2.6	2.6	2.29	2.29	1.57	1.57
	8 = 4+4	Derived	T1ChN		T1ChN/T1Ch8_u			
			2.5	2.5	2	1.5		
	N	Derived	T1ChN					
			2.5-3.1	2.5-3.1				
2 Tap	1	Yes	T2Ch1					
			8	8				
	2	Yes	T2Ch2					
			4	4				
	3	Yes	T2Ch3					
			3.3	3.3				
	4 = 3+1	Yes	T2ChN		T2Ch4_u			
			4.5	4.5	3	2.5		
	5 = 3+2	Derived	T2ChN		T2Ch5_u			
			3.6	3.6	-	2.6		
	6 = 3+3	Derived	T2ChN		T2Ch6_u			
			3.3	3.3	-	2.5		
	7 = 3+3+1	Derived	T2ChN		T2ChN/T2Ch7_u			
			4	4	3.14	2.57		
	8 = 3+3+2	Derived	T2ChN		T2ChN/T2Ch8_u			
			3.5	3.5	3	2.5		
	N	Derived	T2ChN					
			3.3-3.8	3.3-3.8				
T	N	Yes	TNChN					
			9*T	9*T				

UniCoef (u) – UniCoef indicates all channels use same set of Coefficients.

Inplace(i) – Inplace Indicates Output pointer is same as Input .

The vacant boxes in all the tables indicates that particular filter is not available as such and possibly can be derived from other filters.

DF2 Mixed precision (DP-DP) IIR filters[1]

In the Mixed precision class of IIR filters, the input x(n) and output y(n) of the filter are single precision data, where as both the Coefficients and Delay states are double precision data. These filters are not available for TMS320DA8xx DSPs. In these Filters, the input x(n) is converted to double precision data, the filtering is done on double precision data and the output is converted from double precision to single precision data y(n). The filter implementation structure of mixed precision (DP-DP) IIR filters is slightly different from the normal IIR implementation. For optimization purpose, the common input gain in the feed-forward path is taken out as a single initial input gain. Thus all the feed-forward coefficients will be divided by a factor of 'b_0', i.e.

*y(n) = b0(x(n) + b1/b0*x(n-1) +..) + a1*y(n-1)+...*

Refer to the example provided for more details about coefficient arrangement, filter structure, etc.

Table 2. DF2 IIR, mixed precision (DP-DP), table

IIR *Coeff =DoublePrecision*			General		UniCoef(u)		UniCoef/inplace(ui)	
Tap	Ch	Available	Base function name - Filter_iir"**ABC**"( )					
			Clks / sample / ch on TMS320DA6xx		Clks / sample / ch on TMS320DA7xx			
1 tap	1	Yes	T1Ch1_mx					
			18	18				
2 tap	1	Yes	T2Ch1_mx				T2Ch1_ui_mx	
			20	20			18	18
	2	Yes			T2Ch2_u_mx		T2Ch2_ui_mx	
					16	16	12.5	11.5
	N	Derived	T2ChN_mx		T2ChN_mx		T2ChN_mx	
			20	20	even~16 odd~16+4/N	even~16 odd~16+4/N	even~12.5 odd~12.5+5.5/N	even~11.5 odd~11.5+5.5/N
4 tap	1	Yes	T4Ch1_mx					
			28	20				
	N	Derived	T4ChN_mx					
			28	20				

[1] This filter is not available on TMS320DA8xx

DF2/DF1 Mixed precision1 (SP-DP) IIR filters

In DF2 Mixed precision1 (mx1) class of IIR filters, the input x(n), output y(n) and the Coefficients of the filter are single precision data, where as the Delay states are double precision data. These filters are Available for DA7xx and Da8xx based DSP's. The filter implementation structure of mixed precision1 (mx1 SP-DP) IIR filters is slightly different from the normal IIR implementation. For optimization purpose, the common input gain in the feed-forward path is taken out as a single initial input gain. Thus all the feed-forward coefficients will be divided by a factor of 'b_0', i.e.

*y(n) = b0(x(n) + b1/b0*x(n-1) +..) + a1*y(n-1)+...*

In DF1 form, the filter requires more state memory because it has separate delay line for feedback part and feed-forward part. However, the feed-forward part delay line is in single precision, and consequently the computation is in single by single to double precision, taking less cycles than single by double to double precision as in DF2 cases. Currently the DF2 implementation is limited to support only 2nd order IIR, and the DF1 implementation uses different equation as below:

*y(n) = b0 * x(n) + b1 * x(n-1) + b2 * x(n-2) + a1 * y(n-1) + a2 * y(n-2)*

The coefficient structure for DF1 filter is shown below.

pCoef[n] →

b2
b1
b0
a2
a1

Refer to the example provided for more details about coefficient arrangement, filter structure, etc.

Table 3. IIR, mixed precision1(SP-DP), table

IIR	*Coeff =SinglePrecision*		**DF2**			**DF1**
			General	UniCoef(u)	UniCoef/inplace(ui)	
Tap	Ch	Available	Base function name –Filter_iir“**ABC**”( )			
			Clks / sample / ch on TMS320DA7xx			
2 tap	1	Yes	T2Ch1_mx1 15			12
	2	Yes		T2Ch2_u_mx1 8		6
	3	Yes		T2Ch3_u_mx1 8.6		8
	4	Yes		T2Ch3_u_mx1 9		6
	N	Derived	T2ChN_mx1 15	T2ChN_mx1 even~13 odd~13+4/N		Even : 6 Odd : (N+1) * 6 / N
4 tap	1	Yes	T4Ch1_mx1 18			
	N	Derived	T4ChN_mx1 18			

FIR – The implementations have been optimized for high filter orders (> 64), preferably a power of 2, and 'running filtering'. It uses circular buffering for the filter states and maintains a circular index number in the MSB 16bits of the 'use' element of layer-2 filter handle, which is shared by the 'taps' parameter (LSB 16 bits).

Table 4. FIR table

FIR *Single Precision*			General		UniCoef(u)	UniCoef/inplace(ui)
Tap	Ch	Available	Base function name - Filter_fir"**ABC**"( )			
			Clks / sample / ch on TMS320DA6xx		Clks / sample / ch on TMS320DA7xx	
T	1	Yes	TNCh1			
			T/2	T/2		
	N	Derived	TNChN			
			T/2	T/2		

Appendix B. IR-Cascade filter group

This appendix explains the details specific to **IR-Cascade**(Impulse-Response Cascade) filter group.

Filter Type field

Under **IR-Cascade**(group=0x1) we have the following sub-groups.

IIR-SOS-DF2(sub-group=0x1) **:**

Here, for a given order N, the filter is implemented in *Second Order Sections*(SOS), i.e. as cascaded biquads, with Direct Form II structure. The filter *type* field is given below.

MS	P	P	0	0	0	0	0	1	0	0	0	1	UC	X	X

X	tap	tap	tap	Tap	tap	tap	tap	tap	tap	tap	ch	ch	ch	ch	ch

ch – This specifies the number of channel for which coefficients are provided. (ch = 1 to 31)

tap – This specifies the tap or order of the filter(tap = 4 to 1022, as it currently supports only even taps > 4). For eg. If tap=6 it means 3 cascaded biquads to be applied.

All the other *type* field bits are same as for the **IR** filter group.

Coefficient structure

The FIL coefficient structure is same as of **IR** filter group, except for how the coefficient values are arranged, as explained below.

For optimization purpose, the input gain(b0) in the feed forward path of all cascade stages is taken out as a single initial input gain. For example gain in below example is equal to cascade1(b0)* cascade2(b0)*…*cascaden(b0)

IIR-SOS-DF2 :

For tap=2*number of cascaded biquads,

```
CoeffDataType coeffArray[ ] =
{
    gain,
     b1,    // Cascade 1
     b2,
     a1,
     a2,
     b1,    // Cascade 2
     b2,
     a1,
     a2,
     ..
     ..
};
// where H(z)  = gain*H1(z)H2(z)...Hn(z)
//       Hn(z) = (1 + b0*z-1 + b1*z-2)
//               ----------------------
//               (1 - a1*z-1 - a2*z-2)
```

Layer-2 interface

In the FIL layer-2 handle structure “PAF_AudioFilterParam”, LSB 16 bits of “use” element is used to specify the taps or order of the filter.

Filter core implementation and performance

IIR-SOS-DF2 –The table of SOS(DF2) filter implementations, with their performances and “layer-2” function names, are given in the table below. The given performance numbers are the CPU clocks taken for processing a single audio sample of a single channel.

Table 5. IIR-SOS-DF2 table

IIR-SOS	*Single Precision*	General		UniCoef(u)		UniCoef/inplace(ui)	
Ch	Cascaded biquads (1 Biquad = 2 tap DF2 structure)	Base function name - Filter_iir“**ABC**”( )					
		Clks / Biquad/ sample / ch on TMS320DA6xx		Clks / Biquad/ sample / ch on TMS320DA7xx			
1 ch	2	T2Ch1_c2_df2					
		4	4				
	3	T2Ch1_c3_df2					
		2.67	2.67				
	4	T2Ch1_c4_df2					
		2.25	2.25				
	5	T2Ch1_c5_df2					
		2.4	2.2				
	N	T2Ch1_cN_df2					
		2.25 to 3	2.2 to 3				
2 ch	2					T2Ch2_c2_i_df2*	
						2.25	2.25
	3			T2Ch2_c3_u_df2			
				-	2.16		
	4			T2Ch2_c4_u_df2			
				-	2.12		
	5			T2Ch2_c5_u_df2			
					2.1		
N ch	N	T2ChN_cN_df2					
		2.25 to 3	2.2 to 3				

* Filter_iirT2Ch2_c2_i_df2 is not Unicoef/Inplace, but an Inplace Filter

Appendix C. Examples

IR group

Example.1 IIR filter (Single Precision) - Layer-1

IIR filter equations (order – 2):

*Difference equation: y(n) = Bo*x(n) + B1*x(n-1) + B2*x(n-2) + A1*y(n-1) + A2*y(n-2)*

Z transform: $H(z) = \frac{B0 + B1*z^{-1} + B2*z^{-2}}{1 - A1*z^{-1} - A2*z^{-2}}$

File : iir_fil_coef.c

```
#include "std.h"
#include "fil.h"
#include "fil_tii.h"

/* Coefficients for 32, 88.2 & 64kHz.*/
PAF_AudioData IIRFilterCoeff_32[5] = {
 +0.392526E+00, /* coefficient +B0 */
 -0.578005E+00, /* coefficient +B1 */
 +0.338812E+00, /* coefficient +B2 */
 +0.374990E+01, /* coefficient +A1 */
 -0.902626E+00  /* coefficient +A2 */
};
PAF_AudioData IIRFilterCoeff_88[5] = {
 +0.925261E+00, /* coefficient +B0 */
 +0.780052E+00, /* coefficient +B1 */
 -0.388123E+00, /* coefficient +B2 */
 +0.749904E+01, /* coefficient +A1 */
 -0.026265E+00  /* coefficient +A2 */
};
PAF_AudioData IIRFilterCoeff_64[5] = {
 +0.252612E+00, /* coefficient +B0 */
 -0.800534E+00, /* coefficient +B1 */
 +0.881256E+00, /* coefficient +B2 */
 +0.499078E+01, /* coefficient +A1 */
 -0.262690E+00  /* coefficient +A2 */
};

static const struct IIRFilterStructType{
 LgUns type;
 LgUns sampRate;
 PAF_AudioData *IIR_Coeff[3];
}IIRFilterStruct = {
 0xA0040041, /* type – SP : Unicoefficient : taps=2 : ch=1 */
 0x00000124, /* sample rates 32, 88 and 64KHz */
 {
  IIRFilterCoeff_32,
  IIRFilterCoeff_88,
  IIRFilterCoeff_64
 }
};


IFIL_Status IIIR_PARAMS_STATUS = {
 sizeof(IFIL_Status), /* size */
 0x0C07,              /* mode */ /* ch–0,1,2,10 & 11 */
 0x1,                 /* use */ /* ON */
```

```
 0x0C03,              /* mask select */ /* ch–0,1,10 & 11 */
 0x0                  /* mask status */
};

IFIL_Config IIIR_PARAMS_CONFIG = {
 sizeof(IFIL_Config), /* size */
 (const PAF_FilCoef *)&IIRFilterStruct /* IIR filter Coefficients */
};

const IFIL_Params IIIR_PARAMS = {
 sizeof(IFIL_Params), /* size */
 0x1, /* Unused */
 &IIIR_PARAMS_STATUS,
 &IIIR_PARAMS_CONFIG
};
```

File : iir_main.c

```
#include "std.h"
#include "fil.h"
#include "fil_tii.h"

void main(void)
{
    FIL_Handle iirHandle;

    FIL_init();

    if((iirHandle = FIL_create(&FIL_TII_IFIL, (FIL_Params *)&IIIR_PARAMS) ) != 0)
    {
     FIL_apply(iirHandle, &pAudioFrame);
    }

    FIL_delete(filHandle_1);
    FIL_exit();
}
```

Example.2 FIR filter (Single Precision) - Layer-1

FIR filter equations (order – 2) :

*Difference equation : y(n) = Bo*x(n) + B1*x(n-1) + B2*x(n-2)*

Z transform : H(z) = B0 + B1 z^{-1} + B2* z^{-2}*

File : fir_fil_coef.c

```
#include "std.h"
#include "fil.h"
#include "fil_tii.h"

/* Coefficients for 32, 88.2 & 64kHz.*/
PAF_AudioData FIRFilterCoeff_32[3] = {
 +0.925266E+00, /* coefficient +B0 */
 -0.578005E+00, /* coefficient +B1 */
 +0.338812E+00, /* coefficient +B2 */
 };
PAF_AudioData FIRFilterCoeff_88[3] = {
 +0.925261E+00, /* coefficient +B0 */
 +0.780052E+00, /* coefficient +B1 */
 -0.388123E+00, /* coefficient +B2 */
};
PAF_AudioData FIRFilterCoeff_64[3] = {
 +0.9876E+00, /* coefficient +B0 */
```

```
 -0.800534E+00, /* coefficient +B1 */
 +0.881256E+00, /* coefficient +B2 */
};

static const struct FIRFilterStructType{
 LgUns type;
 LgUns sampRate;
 PAF_AudioData *FIR_Coeff[3];
}FIRFilterStruct = {
 0xA00C0041, /* type – SP : Unicoefficient : taps=2 : ch=1 */
 0x00000124, /* sample rates 32, 88 and 64KHz */
 {
  FIRFilterCoeff_32,
  FIRFilterCoeff_88,
  FIRFilterCoeff_64
 }
};

IFIL_Status IFIR_PARAMS_STATUS = {
 sizeof(IFIL_Status), /* size */
 0x0C07,              /* mode */ /* ch–0,1,2,10 & 11 */
 0x1,                 /* use */ /* ON */
 0x0C03,              /* mask select */ /* ch–0,1,10 & 11 */
 0x0                  /* mask status */
};

IFIL_Config IFIR_PARAMS_CONFIG = {
 sizeof(IFIL_Config), /* size */
 (const PAF_FilCoef *)&FIRFilterStruct /* FIR filter Coefficients */
};

const IFIL_Params IFIR_PARAMS = {
 sizeof(IFIL_Params), /* size */
 0x1, /* Unused */
 &IFIR_PARAMS_STATUS,
 &IFIR_PARAMS_CONFIG
};
```

File : fir_main.c

```
#include "std.h"
#include "fil.h"
#include "fil_tii.h"

void main(void)
{
    FIL_Handle firHandle;

    FIL_init();

    if((firHandle = FIL_create(&FIL_TII_IFIL, (FIL_Params *)&IFIR_PARAMS) ) != 0)
    {
     FIL_apply(firHandle, &pAudioFrame);
    }

    FIL_delete(filHandle_1);
    FIL_exit();
}
```

Example.3 IIR filter (Single Precision) - Layer-2

IIR filter equations (order – 2) :

*Difference equation : y(n) = Bo*x(n) + B1*x(n-1) + B2*x(n-2) + A1*y(n-1) + A2*y(n-2)*

$$Z \text{ transform : } H(z) = \frac{B0 + B1*z^{-1} + B2*z^{-2}}{1 - A1*z^{-1} - A2*z^{-2}}$$

- Channels filtered = 2

- Different coefficients are used for different channels.

- In the below example the user application (IIR_APP) is assumed to be XDAISed.

File : iir_fil_coef.c

```
#include "std.h"
#include "filters.h"  //These Header files are available in T:/pa/asp/fil/alg &
                      //T:/pa/asp/fil/src make sure to include this path in your Build


/* Coefficients for 48KHz.*/
Float IIRFilterCoeff_ch0_48[5] = {
 +0.392526E+00, /* coefficient +B0 */
 -0.578005E+00, /* coefficient +B1 */
 +0.338812E+00, /* coefficient +B2 */
 +0.374990E+01, /* coefficient +A1 */
 -0.902626E+00  /* coefficient +A2 */};

Float IIRFilterCoeff_ch1_48[5] = {
 +0.92526E+00, /* coefficient +B0 */
 -0.78005E+00, /* coefficient +B1 */
 +0.38812E+00, /* coefficient +B2 */
 +0.74990E+01, /* coefficient +A1 */
 -0.02626E+00  /* coefficient +A2 */
};
static const Float *IIRFilterCoeff[2] = {
  IIRFilterCoeff_ch0_48,
  IIRFilterCoeff_ch1_48
};
```

File : iiir.h

```
/*
 *  This header defines all types, constants, and functions shared by all
 *  implementations of the ASP Example Demonstration algorithm.
 */
#ifndef IIIR_
#define IIIR_

#include <ialg.h>
#include <xdas.h>

#include "icom.h"
#include "paftyp.h"
#include "filters.h"
```

```
/*
 *  ======== IIIR_Obj ========
 *  Every implementation of IIIR *must* declare this structure as
 *  the first member of the implementation's object.
 */

typedef struct IIIR_Obj {
    struct IIIR_Fxns *fxns;    /* function list: standard, public, private */
} IIIR_Obj;

/*
 *  ======== IIIR_Handle ========
 *  This type is a pointer to an implementation's instance object.
 */
typedef struct IIIR_Obj *IIIR_Handle;

/*
 *  ======== IIIR_Status ========
 *  This Status structure defines the parameters that can be changed or read
 *  during real-time operation of the algorithm.  This structure is actually
 *  instantiated and initialized in iiir.c.
 */
typedef volatile struct IIIR_Status {

    Int size;             /* This value must always be here, and must be set to the
                             total size of this structure in 8-bit bytes, as the
                             sizeof() operator would do. */
    XDAS_Int8 mode;       /* This is the 8-bit IIR Mode Control Register.  All
                             Algorithms must have a mode control register.  */

    ........
} IIIR_Status;

/*
 *  ======== IIIR_Config ========
 *  Config structure defines the parameters that cannot be changed or read
 *  during real-time operation of the algorithm.
 */
typedef struct IIIR_Config {
    Int dummy;
    PAF_FilParam *pParams; //Declare a pointer for Layer -2 Interface
} IIIR_Config;

/*
 *  ======== IIIR_Params ========
 *  This structure defines the parameters necessary to create an
 *  instance of a IIR object.
 *
 *  Every implementation of IIIR *must* declare this structure as
 *  the first member of the implementation's parameter structure.  This structure is
 *  actually instantiated and initialized in iiir.c.
 */
typedef struct IIIR_Params {
    Int size;
    const IIIR_Status *pStatus;
    IIIR_Config config;
} IIIR_Params;


.........
```

File : iir_app_ialg.c

```
/*  Layer-2 IIR example.*/

#include <std.h>
#include <ialg.h>

#include <iiir.h>
#include <iir_app.h>
#include <iir_app_priv.h>

#include <filters.h>

/* ======== IIR_APP_activate ======== */
Void IIR_APP_activate(IALG_Handle handle)
{
}

/* ======== IIR_APP_alloc ======== */
Int IIR_APP_alloc(const IALG_Params *algParams,
                IALG_Fxns **pf, IALG_MemRec memTab[])
{
   const IIIR_Params *params = (Void *)algParams;
   Int ch = 2;

   /* pIn[],pOut & pCoef[] may not be allocated if they are already given */
   Int ptrArrays = 3; /*pIn[2],pOut[2] & pVar[2]*/

    if (params == NULL) {
        params = &IIIR_PARAMS;  /* set default parameters */
    }

    /* Request memory for IIR_APP object */
    memTab[0].size = (sizeof(IIR_APP_Obj)+3)/4*4 + (sizeof(IIR_APP_Status)+3)/4*4;
    memTab[0].alignment = 4;
    memTab[0].space = IALG_EXTERNAL;
    memTab[0].attrs = IALG_PERSIST;

    /* Request Memory for Layer-2 Interface */
    memTab[1].size = ( sizeof(PAF_FilParam) + 3 ) /4*4;
    memTab[1].alignment = 4;
    memTab[1].space = IALG_EXTERNAL;
    memTab[1].attrs = IALG_PERSIST;

    memTab[2].size = FIL_varsPerCh(0x20000041)*ch; // The function FIL_varsPerCh()will take
    memTab[2].alignment = sizeof(float);          // filter type field as parameter and
    memTab[2].space = IALG_EXTERNAL; // returns number of bytes required for Delay Memory
    memTab[2].attrs = IALG_PERSIST;  //of one Channel. This function is available in fil.lib
    //Notice that the Filter mask above  points to sp-iir-2tap-1ch instead of sp-iir-2tap-2ch
    //this is because the function FIL_varsPerCh() will return Delay memory required for 1
    //Channel
    return (3);
}

/* ======== IIR_APP_deactivate ======== */
Void IIR_APP_deactivate(IALG_Handle handle)
{
}

/* ======== IIR_APP_free ======== */
Int IIR_APP_free(IALG_Handle handle, IALG_MemRec memTab[])
{
    return (*handle->fxns->algAlloc)(NULL, NULL, memTab);
}
```

```
/* ======== IIR_APP_initObj ======== */
Int IIR_APP_initObj(IALG_Handle handle,
                const IALG_MemRec memTab[], IALG_Handle p,
                const IALG_Params *algParams)
{
    IIR_APP_Obj *iir = (Void *)handle;
    const IIIR_Params *params = (Void *)algParams;
    PAF_FilParam * pParams;
    Int ch = 2;
    Int varPerCh;

    varPerCh = FIL_varsPerCh(0x20000041);

    if (params == NULL)
    {
        params = &IIIR_PARAMS;  /* set default parameters */
    }

    /* The Memory of Layer-2 Interface is assigned to the pointer declared
    in Config structure of handle */

    iir->config.pParams = (PAF_FilParam *)memTab[1].base;

    pParams =   (PAF_FilParam *) iir->config.pParams;

    pParams->pCoef = (Void **)IIRFilterCoeff;

    /* Pointer array intialization */
    pParams->pVar[0] = (Void *)memTab[2].base;
    pParams->pVar[1] = (Void *)(memTab[2].base + varPerCh);

    /* The i/p and o/p are to be initialized */

    /* The variable memory is initialized to zero */
    FIL_memReset((Void *)memTab[2].base, (Int)memTab[2].size );
    // This function will clear the memory pointed by the pointer for size number of bytes
    //This function is available in fil.lib

    return (IALG_EOK);
}

/* ======== IIR_APP_control ======== */
/*  Implementation of the control operation. */
Int IIR_APP_control(IALG_Handle handle, IALG_Cmd cmd, IALG_Status *pStatus)
{
}

/*
 *  ======== IIR_APP_moved ========
 */
Void IIR_APP_moved(IALG_Handle handle,
                const IALG_MemRec memTab[], IALG_Handle p,
                const IALG_Params *algParams)
{
}

/* This function returns the number of memTabs used for allocating memory */
Int IIR_APP_numAlloc()
{
    return(3);
}
```

File : iir_app_iiir.c

```
/*
 *  IIR Module implementation - APP implementation of an
 *  ASP Example Demonstration algorithm.
 */

#include <std.h>

#include <iiir.h>
#include <iir_app.h>
#include <iir_app_priv.h>
#include <iirerr.h>

#include <xdas.h>

#include "filters.h"
#include "paftyp.h"

#if PAF_AUDIODATATYPE_FIXED
#error fixed point audio data type not supported by this implementation
#endif /* PAF_AUDIODATATYPE_FIXED */

/*
 *  ======== IIR_APP_apply ========
 *  APP's implementation of the apply operation.
 *
 *  This function is called after the IIR_APP_reset function, below, and
 *  deals with control data AND sample data.
 *
 */

#ifndef _TMS320C6X
#define restrict
#endif /* _TMS320C6X */

Int IIR_APP_apply(IIIR_Handle handle, PAF_AudioFrame *pAudioFrame)
{
    IIR_APP_Obj * restrict iir = (Void *)handle;

    PAF_FilParam * pParams;

     ....
    /* Layer-2 filter handle pointer “pParams” has to be initialized. */
      pParams = iir->config.PParams;
    // For Filtering Channel 0 and 1 of pAudioFrame we can assign input and output
    // pointers as follows
      pParams->pIn = (void **)pAudioFrame->data.sample ;
      pParams->pOut = (void **)pAudioFrame->data.sample ;
      pParams->sampleCount = (int)pAudioFrame->sampleCount;

    //pCoef and pVar are assigned in IIR_APP_initObj()
    //Note:- For filtering streams with multiple sample rates the pCoef pointer
    //Should be updated dynamically with proper set of Coefficients depending upon sample
    //rate. The pVar memory should also be cleared dynamically when there is a change in
    //Sample rate

    //Call the Filter with layer-2 Interface
    Filter_iirT2Ch2(pParams); /* Direct filter function call */

    ....
}
```

```
Int
IIR_APP_reset(IIIR_Handle handle, PAF_AudioFrame *pAudioFrame)
{
   IIR_APP_Obj * restrict iir = (Void *)handle;

   PAF_FilParam * pParams = iir->config.pParams;
   Void * pVar;

    pVar = (Void *)pParams->pVar[0]; //Clear the Delay memory of Channel-0
    FIL_memReset(pVar, FIL_varsPerCh(0x20000041);
    pVar = (Void *)pParams->pVar[1]; //Clear the Delay memory of Channel-1
    FIL_memReset(pVar, FIL_varsPerCh(0x20000041);

    return 0;
}
```

Example.4 IIR filter (Mixed Precision, DP-DP) - Layer-2

IIR filter equations (order – 2) :

*Difference equation : y(n) = Bo*x(n) + B1*x(n-1) + B2*x(n-2) + A1*y(n-1) + A2*y(n-2)*

Z transform : $H(z) = \frac{B0 + B1*z^{-1} + B2*z^{-2}}{1 - A1*z^{-1} - A2*z^{-2}}$

- Channels filtered = 2

- Different coefficients are used for different channels.

- In the below example the user application (IIR_APP) is assumed to be XDAISed.

File : iir_fil_coef.c

```
#include "std.h"

/* Coefficients for 48KHz.*/
Double IIRFilterCoeff_ch0_48[5] = {
 +0.92526E+00, /* coefficient +B0 */
 -0.78005E+00, /* coefficient +B1/B0 */
 +0.38812E+00, /* coefficient +B2/B0 */
 +0.74990E+01, /* coefficient +A1 */
 -0.02626E+00  /* coefficient +A2 */};

Double IIRFilterCoeff_ch1_48[5] = {
 +0.2526E+00, /* coefficient +B0 */
 -0.8005E+00, /* coefficient +B1/B0 */
 +0.8812E+00, /* coefficient +B2/B0 */
 +0.4990E+01, /* coefficient +A1 */
 -0.2626E+00  /* coefficient +A2 */
};
static const Double *IIRFilterCoeff[2] = {
  IIRFilterCoeff_ch0_48,
  IIRFilterCoeff_ch1_48
};
```

The remaining files are similar to the normal IIR (SP) filter, except the FIL_varsPerCh(0x20040041) should be replaced with FIL_varsPerCh(0x00040041) given in Example 3.

Example.5 IIR filter (Mixed Precision, SP-DP) - Layer-2

IIR filter equations (order – 2) :

*Difference equation : y(n) = Bo*x(n) + B1*x(n-1) + B2*x(n-2) + A1*y(n-1) + A2*y(n-2)*

Z transform : $H(z) = \frac{B0 + B1*z^{-1} + B2*z^{-2}}{1 - A1*z^{-1} - A2*z^{-2}}$

- Channels filtered = 2

- Different coefficients are used for different channels.

- In the below example the user application (IIR_APP) is assumed to be XDAISed.

File : iir_fil_coef.c

```
#include "std.h"

/* Coefficients for 48KHz.*/
Float IIRFilterCoeff_ch0_48[5] = {
 +0.92526E+00, /* coefficient +B0 */
 -0.78005E+00, /* coefficient +B1/B0 */
 +0.38812E+00, /* coefficient +B2/B0 */
 +0.74990E+01, /* coefficient +A1 */
 -0.02626E+00  /* coefficient +A2 */};

Float IIRFilterCoeff_ch1_48[5] = {
 +0.2526E+00, /* coefficient +B0 */
 -0.8005E+00, /* coefficient +B1/B0 */
 +0.8812E+00, /* coefficient +B2/B0 */
 +0.4990E+01, /* coefficient +A1 */
 -0.2626E+00  /* coefficient +A2 */
};
static const Float *IIRFilterCoeff[2] = {
  IIRFilterCoeff_ch0_48,
  IIRFilterCoeff_ch1_48
};
```

The remaining files are similar to the normal IIR (SP) filter, except the FIL_varsPerCh(0x20040041) should be replaced with FIL_varsPerCh(0x00048041) given in Example 3.

Example.6 An IIR filter function “Filter_iirT2Ch2( )” (Single Precision)

```
/************************ IIR FILTER ***************************************/
/****************** ORDER 2 and CHANNELS 2 *********************************/
/*                                                                         */
/* Filter equation  : H(z) =  b0 + b1*z~1 + b2*z~2                         */
/*                           ----------------------                        */
/*                            1  - a1*z~1 - a2*z~2                         */
/*                                                                         */
/*   Direct form    : y(n) = b0*x(n) + b1*x(n-1) + a1*y(n-1) + a2*y(n-2)   */
/*                                                                         */
/*   Canonical form : w(n) = x(n)    + a1*w(n-1) + a2*w(n-2)               */
/*                    y(n) = b0*w(n) + b1*w(n-1) + b2*w(n-2)               */
/*                                                                         */
/*   Filter variables :                                                    */
/*      y(n) - *y,         x(n)   - *x                                     */
/*      w(n) -             w(n-1) - w1         w(n-2) - w2                 */
/*      b0 - filtCfsB[0], b1 - filtCfsB[1], b2 - filtCfsB[2]               */
/*      a1 - filtCfsA[0], a2 - filtCfsA[1]                                 */
/*                                                                         */
/*                             w(n)    b0                                  */
```

```
/*     x(n)-->---[+]---->--------@------>>-----[+]--->--y(n)                */
/*                |              |              |                           */
/*                ^              v              ^                           */
/*                |     a1    +-----+    b1     |                           */
/*               [+]----<<----| Z~1 |---->>----[+]                          */
/*                |           +-----+           |                           */
/*                |              |              |                           */
/*                ^              v              ^                           */
/*                |     a2    +-----+    b2     |                           */
/*                 ----<<-----| Z~1 |---->>-----                            */
/*                           +-----+                                        */
/*                                                                          */
/****************************************************************************/
#include "filters.h"

#pragma CODE_SECTION(Filter_iirT2Ch2, ".text:Filter_iirT2Ch2")

Int Filter_iirT2Ch2( PAF_FilParam *pParam )
{
    FIL_CONST PAF_AudioData * xL, * xR;
    PAF_AudioData * restrict yL, * restrict yR;
    FIL_CONST PAF_AudioData *filtCfsBL, *filtCfsBR;
    FIL_CONST PAF_AudioData *filtCfsAL, *filtCfsAR;
    PAF_AudioData * restrict filtVarsL, * restrict filtVarsR;
    PAF_AudioData accum1,accum2,accum3,accum4;
    PAF_AudioData input_dataL, input_dataR;
    Int count, samp;
    PAF_AudioData w1L, w2L, w1R, w2R ;

    xL = (PAF_AudioData *)pParam->pIn[0];
    xR = (PAF_AudioData *)pParam->pIn[1];

    yL = (PAF_AudioData *)pParam->pOut[0];
    yR = (PAF_AudioData *)pParam->pOut[1];

    filtCfsBL  = (PAF_AudioData *)pParam->pCoef[0];
    filtCfsAL  = filtCfsBL + 3;

    filtCfsBR  = (PAF_AudioData *)pParam->pCoef[1];
    filtCfsAR  = filtCfsBR + 3;

    filtVarsL = (PAF_AudioData *)pParam->pVar[0];
    filtVarsR = (PAF_AudioData *)pParam->pVar[1];

    count = pParam->sampleCount;

    w1L = filtVarsL[0];
    w2L = filtVarsL[1];

    w1R = filtVarsR[0];
    w2R = filtVarsR[1];

    for (samp = 0; samp < count; samp++)
    {
        accum1 = filtCfsAL[0]*w1L;
        accum2 = filtCfsAL[1]*w2L;
        accum3 = filtCfsBL[1]*w1L;
        accum4 = filtCfsBL[2]*w2L;

        input_dataL = *xL++;

        w2L = w1L;
```

```
        w1L = input_dataL + accum2 + accum1;

        *yL++ = filtCfsBL[0]*w1L + accum3 + accum4;

        accum1 = filtCfsAR[0]*w1R;
        accum2 = filtCfsAR[1]*w2R;
        accum3 = filtCfsBR[1]*w1R;
        accum4 = filtCfsBR[2]*w2R;

        input_dataR = *xR++;

        w2R = w1R;

        w1R = input_dataR + accum2 + accum1;

        *yR++ = filtCfsBR[0]*w1R + accum3 + accum4;
    }

    filtVarsL[0] = w1L;
    filtVarsL[1] = w2L;
    filtVarsR[0] = w1R;
    filtVarsR[1] = w2R;

    return(FIL_SUCCESS);
}
```

IR-Cascade group

Example.1 IIR-SOS-DF2 filter (Single Precision) – using FIL Layer-1

Filter equations (order – 4):

*Difference equation: y(n) = Bo*x(n) + B1*x(n-1) + B2*x(n-2) + A1*y(n-1) + A2*y(n-2)*

Z transform: $H(z) = gain * \frac{(1 + B1 * z^{-1} + B2 * z^{-2})}{(1 - A1 * z^{-1} - A2 * z^{-2})} \frac{(1 + B'1 * z^{-1} + B'2 * z^{-2})}{(1 - A'1 * z^{-1} - A'2 * z^{-2})}$

File : fil_coef.c

```
#include "std.h"
#include "fil.h"
#include "fil_tii.h"

/* Coefficients for Left and Right channels */
PAF_AudioData Coeff_Left[] = {
 +0.392526E+00, /* gain */
 -0.578005E+00, /* B1   */ /* Cascade 1 */
 +0.338812E+00, /* B2   */
 +0.374990E+01, /* A1   */
 -0.902626E+00  /* A2   */
 -0.578005E+00, /* B'1  */ /* Cascade 2 */
 +0.338812E+00, /* B'2  */
 +0.374990E+01, /* A'1  */
 -0.902626E+00  /* A'2  */
};
PAF_AudioData Coeff_Right[] = {
 +0.92526E+00, /* gain */
 -0.78005E+00, /* B1   */ /* Cascade 1 */
 +0.38812E+00, /* B2   */
 +0.74990E+01, /* A1   */
 -0.02626E+00  /* A2   */
```

```
 -0.78005E+00, /* B'1  */ /* Cascade 2 */
 +0.38812E+00, /* B'2  */
 +0.74990E+01, /* A'1  */
 -0.02626E+00  /* A'2  */
};

static const struct IIRCascFilterStructType{
 unsigned int type;
 unsigned int sampRate;
 PAF_AudioData *IIRCasc_Coeff[2];
} IIRCascFilterStruct = {
 0x20880082, /* type – SP : taps=4 : ch=2 */
 0x00000000,  /* Sample Rate field – NA */
 {
  Coeff_Left,
  Coeff_Right,
 }
};


IFIL_Status IIRCasc_PARAMS_STATUS = {
 sizeof(IFIL_Status), /* size */
 0x03,                /* mode */ /* Max channels filtered–0,1*/
 0x1,                 /* use */ /* ON */
 0x03,                /* mask select */ /* Select ch–0,1*/
 0x0                  /* mask status */
};

IFIL_Config IIRCasc_PARAMS_CONFIG = {
 sizeof(IFIL_Config), /* size */
 (const PAF_FilCoef *)&IIRCascFilterStruct /* IIR filter Coefficients */
};

const IFIL_Params IIRCasc_PARAMS = {
 sizeof(IFIL_Params), /* size */
 0x1, /* Unused */
 &IIRCasc_PARAMS_STATUS,
 &IIRCasc_PARAMS_CONFIG
};
```

File : iir_main.c

```
#include "std.h"
#include "fil.h"
#include "fil_tii.h"

void main(void)
{
    FIL_Handle filHandle;

    FIL_init();

    if((filHandle = FIL_create(&FIL_TII_IFIL, (FIL_Params *)& IIRCasc_PARAMS) ) != 0)
    {
     FIL_apply(filHandle, &pAudioFrame);
    }

    FIL_delete(filHandle);
    FIL_exit();
}
```

References

1. Baudendistel, Kurt (et al.), *PA3 User's Guide (MN1001)*, Momentum Data Systems, Inc.,2002
2. Baudendistel, Kurt, *Audio Stream Processing* application report, Momentum Data Systems, Inc., 2002
3. Torud, Stig, Making DSP Algorithms Compliant with the TMS320 DSP Algorithm Standard (SPRA579B), Texas Instruments, Inc., November 2000