C28x Fixed Point DSP Library

Module User's Guide

C28x Foundation Software

v1.01

January 10, 2011

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

TI warrants performance of its semiconductor products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, license, warranty or endorsement thereof.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations and notices. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible or liable for any such use.

Resale of TI's products or services with *statements different from or beyond the parameters* stated by TI for that products or service voids all express and any implied warranties for the associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: Standard Terms and Conditions of Sale for Semiconductor Products.
www.ti.com/sc/docs/stdterms.htm

Mailing Address:
Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Contents

1. Introduction

The Texas Instruments TMS320C28x Fixed Point DSP Library is collection of highly optimized application functions written for the C28x. These functions enable C/C++ programmers to take full advantage of the performance potential of the C28x. This document provides a description of each function included within the library.

2. Installing the Library

2.1. Where the Files are Located (Directory Structure)

As installed, the *C28x Fixed Point DSP Library* is partitioned into a well-defined directory structure. By default, the library and source code is installed into the C:\TI\controlSUITE\libs\dsp\FixedPointLib\<version> directory. Table 1 describes the contents of the main directories used by library:

Table 1. C28x Fixed Point Library Directory Structure

Directory	Description
<base>	Base install directory. By default this is C:\TI\controlSUITE\libs\dsp\FixedPointLib\v100 For the rest of this document <base> will be omitted from the directory names.
<base>\doc	Documentation including the revision history from the previous release.
<base>\lib	The built libraries.
<base>\include	Include files for the library functions. These include function prototypes and structure definitions.
<base>\source	Source files for the library. This also includes a Code Composer Studio project that can be used to re-build the library if required.
<base>\examples_ccsv4	Example projects for the library functions. This includes example projects that show how to call library functions in Code Composer Studio project.
<base>\examples_ccsv4\<example>\matlab	MATLAB code which can be used as reference when debugging the example project with the Fixed Point library. The FFT results in the example projects can be compared to the MATLAB code results

2.2. Build options used to build the library

The v1.00 library is built with C28x codegen tools V5.2.7 with following options:
-g -o3 -d"_DEBUG" -d"LARGE_MODEL" -ml -v28

In v1.01 there are two libraries, the original library with the build settings mentioned above and another version with fpu32 support, titled **C28x_FixedPoint_Lib_fpu32**. This library can be used in benchmarking applications to compare equivalent functions using fixed point math and floating point math.

2.3. Header Files

A library header file is supplied in the <base>/include folder. This file contains structure definitions and function prototypes. The header file also includes the C28x data type definitions shown below:

```
#ifndef DSP28_DATA_TYPES
#define DSP28_DATA_TYPES
typedef int                int16;
typedef long               int32;
typedef long long          int64;
typedef unsigned int       Uint16;
typedef unsigned long      Uint32;
typedef unsigned long long Uint64;
typedef float              float32;
typedef long double        float64;
#endif
```

3. Module Summary

3.1. Fixed Point DSP Module Summary

This release is for the FFT and Filter modules. Other modules will be added in future releases. The functions under FFT, FIR, IIR modules are member functions of header structure. Users are supposed to call the header structure templates rather than calling these functions directly. CFFT and RFFT modules share common computation function sets.

Description	Prototype
Bit Reverse Modules	
CFFT32_brev	void CFFT32_brev(int32 *src, int32 *dst, int16 size);
RFFT32_brev	void RFFT32_brev(int32 *src, int32 *dst, int16 size);
RFFT32_brev_RT	void RFFT32_brev_RT(void *);
FFT Modules	
FFT32_calc	void FFT32_calc(void *);
FFT32_init	void FFT32_init(void *);
FFT32_izero	void FFT32_izero(void *);
FFT32_mag	void FFT32_mag(void *);
FFT32_win	void FFT32_win(void *);
FIR Modules	
FIR16_init	void FIR16_init(void *);
FIR16_calc	void FIR16_calc(void *);
FIR32_init	void FIR32_init(void *);
FIR32_calc	void FIR32_calc(void *);
IIR Modules	
IIR5BIQ16_init	void IIR5BIQ16_init(void *);
IIR5BIQ16_calc	void IIR5BIQ16_calc(void *);
IIR5BIQ32_init	void IIR5BIQ32_init(void *);
IIR5BIQ32_calc	void IIR5BIQ32_calc(void *);

4. Module Descriptions

CFFT32_brev	*Complex FFT Bit Reverse Function*

Description This function reads N-point in-order real data samples stored in alternate memory locations and writes it as N-complex data in bit-reversed order, to cater to the bit-reversal requirement of complex FFT. It supports both in-place and off-place bit reversing.

Declaration void CFFT32_brev(int16 *src, int16 *dst, int16 size);

Arguments

src Pointer to in-order data samples stored in alternate locations.

dst Pointer to destination array, to store bit revered complex output, ***the destination array buffer must be aligned to 4N word boundary (16-bit word length)*** or 2N long words, where N is number of elements in source array that should be power of 2.

size Number of real-data samples (N) to be bit reversed in complex form, it should be power of 2.

Availability C-Callable Assembly (CcA)

Usage Pointer to the source buffer *src, pointer to the destination buffer *dst and length of the buffer size are passed to the CFFT32_brev function:

Item	Description	Format	Q-Values	Comment
src	Input buffer	Pointer to 32-bit integer array	Q31~Q0	Input data.
dst	Output buffer	Pointer to 32-bit integer array	Q31~Q0	Output data. The first call of this function bit-reversed the real part of the input. The second call of this function bi-reversed the imaginary part of the input.
size	Number of the bit reversing elements	int16	Q0	Must be power of 2.

Background Information

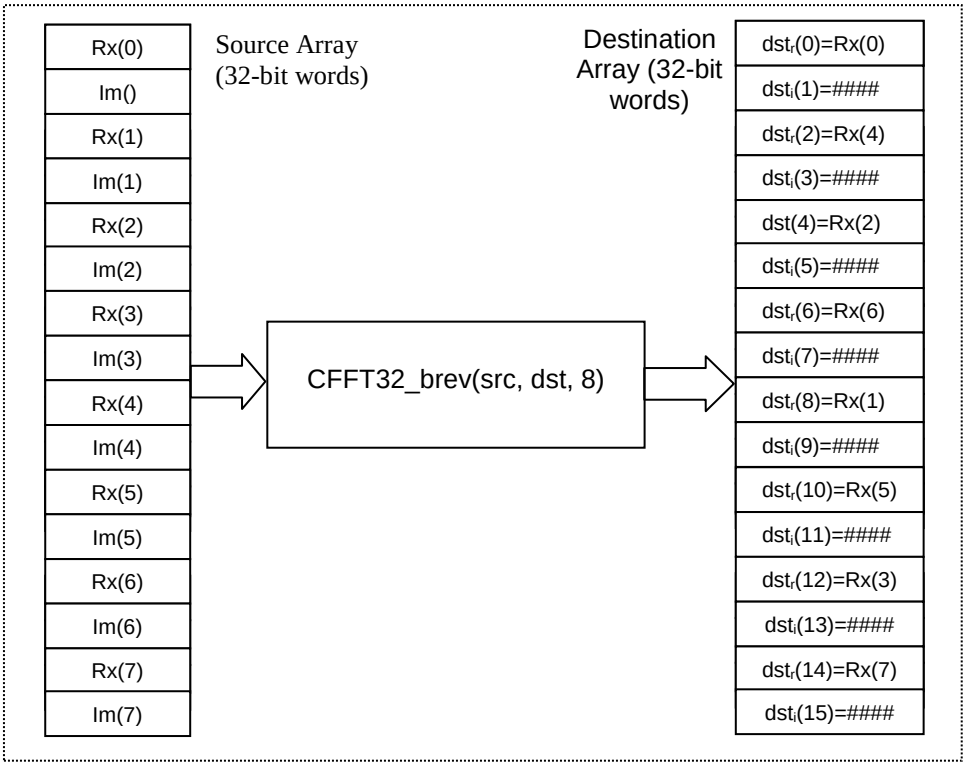
In many real time applications, the data sequences to be processed are real valued. Even though the data is real, complex-valued DFT algorithm can still be used. One simple approach creates a complex sequence from the real sequence; that is real data for the real components and zeros for imaginary components, the complex FFT then be applied directly. Moreover complex FFT needs the input in bit reversed order so that the output at the end of computation will be in natural order.

This function facilitates N point complex FFT computation on the N-point in-order real data sequence stored in alternate memory locations. It reads real input samples and stores it as complex data in bit reversed order to perform complex FFT computation. The real data samples occupy the real part of the complex number and imaginary part will be

zeroed before invoking the complex FFT. If the source and destination pointer is equal, then it performs in-place bit reversal.

It is apparent that, to store the N-real valued sequence (32-bit) in complex form, we need 2N long words to cater to real/imaginary part. ***Desination buffer must be aligned to 4N word boundary (16-bit word length) or 2N long words, where N is the number of samples to be acquired that should be power of 2.***

Below figure exemplifies the bit-reversal of 8 real data samples, to perform complex FFT computation. The real data samples g(n) occupy the real part of the complex data sequence x(n) and imaginary part of x(n) will be cleared before invoking the complex FFT computation routine. Note that the storage buffer must be aligned to 16 longs words to bit reverse 8 real data samples (32-bit) in complex form.



Example: The following sample code obtains the bit-reversed result of the complex input.

```
#include <fft.h>

#define N   128
#define FFT_STAGES   7
 /* Align the INBUF section to 2*FFT_SIZE   */
#pragma DATA_SECTION(ipcb, "FFTipcb");
#pragma DATA_SECTION(ipcbsrc, "FFTipcbsrc");
 long ipcbsrc[2*N];
 long ipcb[2*N];

 long INPUT[2*N];     // Input complex number

 main()
 {
      ……….
      //Input data
      for(i=0; i < (N*2); i=i+2)
      {
          ipcbsrc[i]  =(long)(INPUT[i]);
          ipcbsrc[i+1] = (long)(INPUT[i+1]);
      }
      //Clean up buffer
      for(i=0; i < (N*2); i=i+2)
      {
          ipcb[i]  =0;
          ipcb[i+1] = 0;
      }
   ……….
   //Real part bit reversing
   CFFT32_brev(ipcbsrc, ipcb, N);
   //Imaginary part bit reversing
   CFFT32_brev(&ipcbsrc[1], &ipcb[1], N);
   ……….
}
```

Benchmark Information:

Size	C-Callable ASM
32	282 cycles
64	538 cycles
128	1050 cycles
256	2074 cycles
512	4122 cycles

Note: All buffers and stack are placed in internal memory (zero-wait states in data space).

RFFT32_brev	*Real FFT Bit Reverse Function*

Description This function reads N-point in-order real data samples (32-bit) stored in contiguous locations and writes it as N-complex data in bit-reversed order, to cater to the bit-reversal requirement of complex FFT. It supports both in-place and off-place bit reversing.

Declaration void RFFT32_brev(int16 *src, int16 *dst, int16 size);

Arguments

src Pointer to in-order data samples stored in contiguous locations.

dst Pointer to destination array, to store bit revered complex output, ***the destination array buffer must be aligned to 4N word boundary (16-bit word length)*** or 2N long words, where N is number of elements in source array that should be power of 2.

size Number of real-data samples (N) to be bit reversed in complex form, it should be power of 2.

Availability C-Callable Assembly (CcA)

Object Definition

Item	Description	Format	Q-Values	Comment
src	Input buffer	Pointer to 32-bit integer array	Q31~Q0	Input data is aligned in contiguous region. No zero between two neighboring elements.
dst	Output buffer	Pointer to 32-bit integer array	Q31~Q0	Output data. For real input FFT calculation, the output buffer should be cleaned up before calling this function.
size	Number of the bit reversing elements	int16	Q0	Must be the power of 2.

Background Information

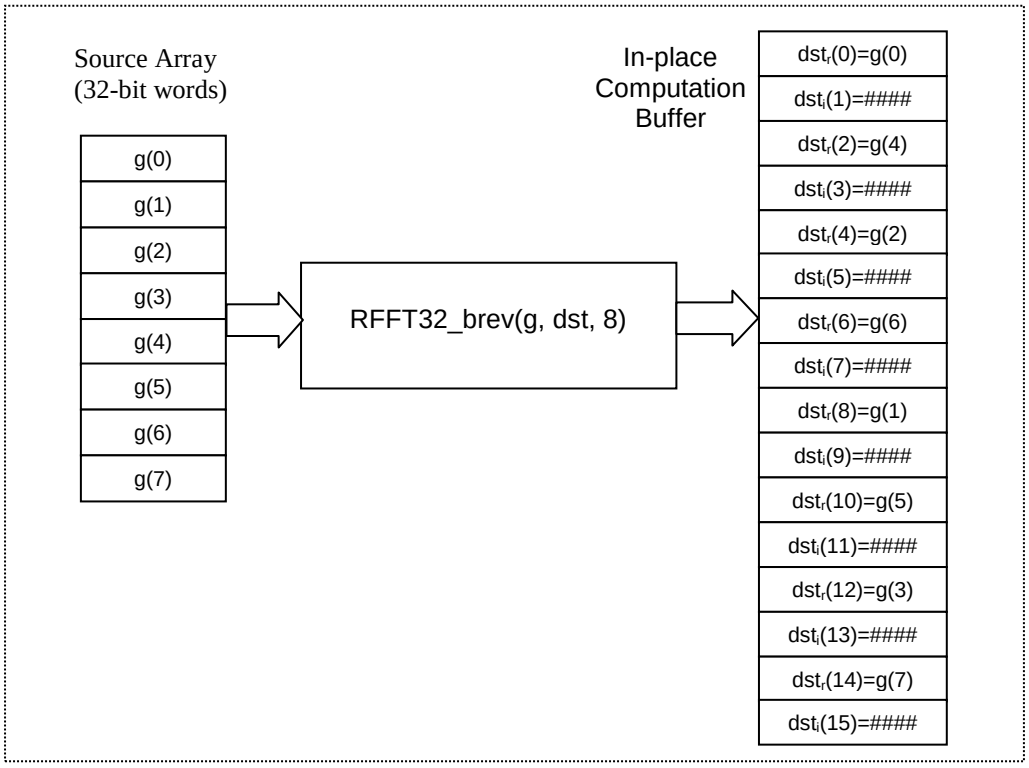
In many real time applications, the data sequences to be processed are real valued. Even though the data is real, complex-valued DFT algorithm can still be used. One simple approach creates a complex sequence from the real sequence; that is real data for the real components and zeros for imaginary components, the complex FFT then be applied directly. Moreover complex FFT needs the input in bit reversed order so that the output at the end of computation will be in natural order.

This function facilitates N point complex FFT computation on the N-point in-order real data sequence stored in contiguous memory locations. It reads real input samples and stores it as complex data in bit reversed order to perform complex FFT computation. The real data samples occupy the real part of the complex number and imaginary part will be zeroed before invoking the complex FFT. If the source and destination pointer is equal, then it performs in-place bit reversal.

It is apparent that, to store the N-real valued sequence (32-bit) in complex form, we need 2N

long words to cater to real/imaginary part. Destination buffer must be aligned to 4N word boundary (16-bit word length) or 2N long words, where N is the number of samples to be acquired that should be power of 2.

Below figure exemplifies the bit-reversal of 8 real data samples, to perform complex FFT computation. The real data samples g(n) occupy the real part of the complex data sequence x(n) and imaginary part of x(n) will be cleared before invoking the complex FFT computation routine. Note that the storage buffer must be aligned to 16 long words to bit reverse 8 real data samples (32-bit) in complex form.



Example: The following sample code obtains the bit-reversed result of the real input.

```
#include <fft.h>

#define N         128         /* FFT_SIZE                    */
#define FFT_STAGES   7        /* log2(FFT_SIZE)              */

/* Align the INBUF section to 2*FFT_SIZE in the linker file   */
#pragma DATA_SECTION(ipcb, "FFTipcb");
#pragma DATA_SECTION(ipcbsrc, "FFTipcbsrc");
long ipcbsrc[N];
long ipcb[2*N];

long INPUT[N];     // Input complex number

main()
{
       ……….
       for(i=0; i < (N); i++)
       {
            ipcbsrc[i]  =(long)(INPUT[i]); //Input data
       }
       //Clean up data buffer
       for(i=0; i < (N*2); i=i+2)
       {
            ipcb[i]  =0;
            ipcb[i+1] = 0;
       }
       ……….
       RFFT32_brev(ipcbsrc, ipcb, N);     // Bit reversed
       ……….
}
```

Benchmark Information:

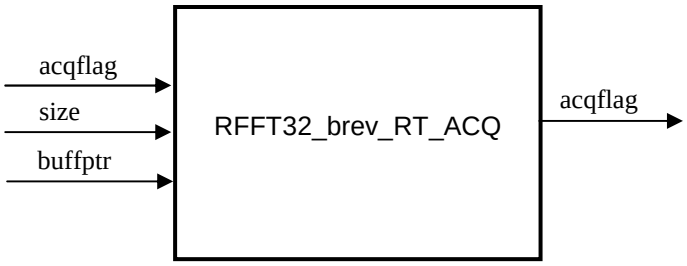
Size	C-Callable ASM
32	251 cycles
64	475 cycles
128	923 cycles
256	1819 cycles
512	3611 cycles

Note: All buffers and stack are placed in internal memory (zero-wait states in data space).

RFFT32_brev_RT	*Real Input Point by Point Bit Reverse Function*

Description

This module acquires N real data samples in real time and stores it as $N/2$-point complex data sequence in bit-reversed order to perform N point real FFT computation.



Header File

`fft.h`

Availability

C-Callable Assembly (CcA)

Object Definition

The structure of RFFT32_ACQ object is defined by following structure definition

```
typedef struct {
        int16   acqflag;
        int16   count;
        int32   input;
        int32   *tempptr;
        int32   *buffptr;
        int16   size;
        void    (*update)(void *);
}RFFT32_brev_RT_ACQ;
```

Item	Description	Format	Range (Hex)	Comment
acqflag	Acquisition flag	int16	0 or 1	Acquisition ENABLE flag. Set this flag to start the acquisition and it will be reset when all the samples are acquired.
count	Integer counter	int16	Q0	Counter to keep track of the acquired samples
input	Data input	int32	Q0~Q31	32 bit integer number
tempptr	Temporary pointer	Pointer to int16	N/A	Temporary pointer, modified in bit reversed order, to store successive samples.
buffptr	Destination buffer pointer	Pointer to int16	N/A	Pointer to the buffer to store the data samples. The buffer should be aligned to 4N words (16-bit word size) or 2N long words, where N is the no. of samples to be acquired.
size	Number of samples	Integer (Q0)	2^k (k=1:15)	Number of samples to be acquired, should be power of 2
update	Member function	Function pointer	N/A	Bit reverse function entry point should be passed to this pointer.

Special Constants and Data types

RFFT32_brev_RT_ACQ

The module definition is created as a data type. This makes it convenient to instance an interface to the FFT module. To create multiple instances of the module simply declare variables of type **RFFT32_brev_RT_ACQ**

RFFT32_brev_RT_ACQ_handle

User defined Data type of pointer to **RFFT32_brev_RT_ACQ** Module

RFFT32_brev_RT_ACQ_DEFAULTS

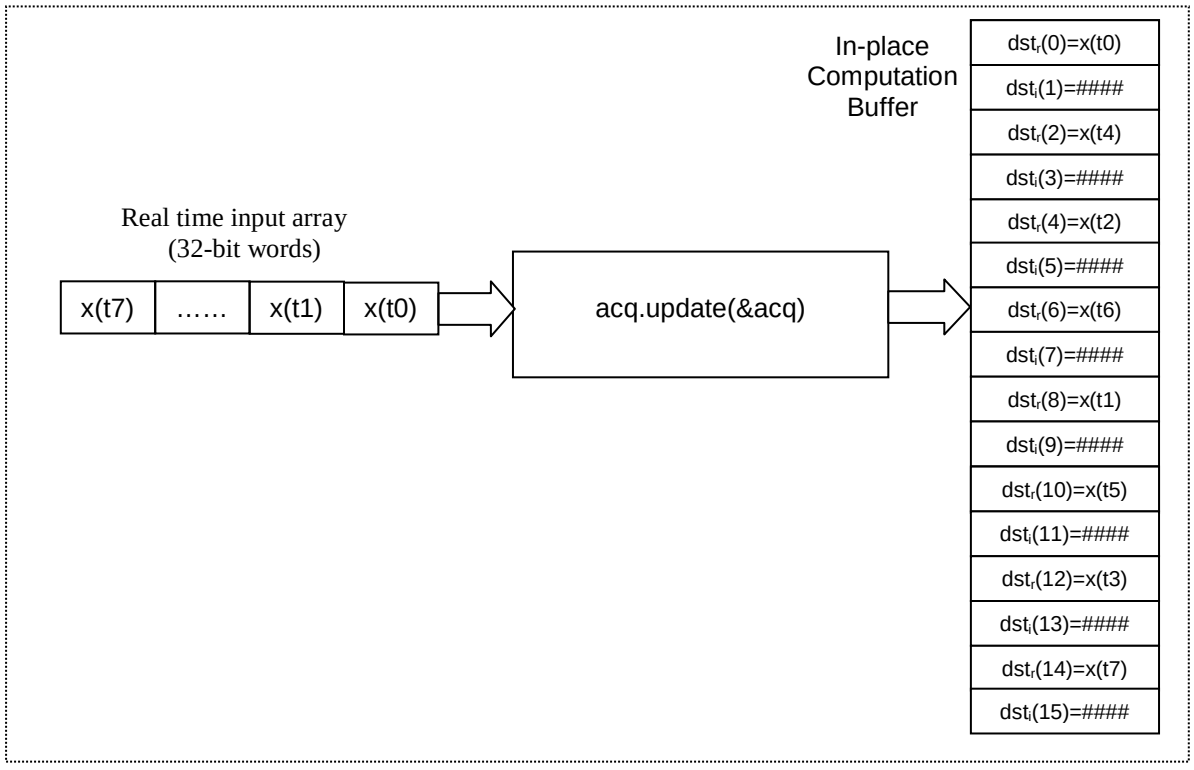
Structure symbolic constant is to initialize RFFT32_brev_RT_ACQ module. This provides the initial values to the terminal variables as well as method pointers.

Methods

void RFFT32_brev_RT(void *);

This routine reads N successive real input samples (32-bit) and stores it as $N/2$ - poincomplex data sequence in bit reversed order. The even data samples occupy the real part and odd data samples occupy imaginary part. Real FFT acquition buffer should be aligned to 2N word (16-bit word size) boundary.

This function starts the acquisition, only if the “acqflag” is set and it resets this flag when samples are acquired. Thus the “acqflag” acts as the trigger for the acquisition module.



Example:

In the header file fft.h, a default header structure templates had been defined and ready to be called

```
#define RFFT32_brev_RT_ACQ_DEFAULTS{
        1,
        0,
        0,
        (int32 *)NULL,
        (int32 *)NULL,
        0,
        (void (*)(void *))RFFT32_brev_RT
}
```

The following sample code shows how to call the point by point bit reversed function.

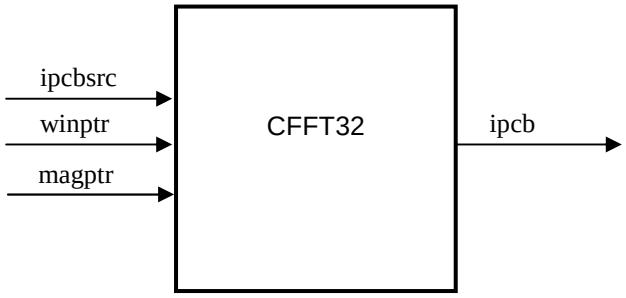
```
#include fft.h

#define     N   32                // FFT size
// Data buffer alignment
#pragma DATA_SECTION(ipcb, "FFTipcb");
#pragma DATA_SECTION(ipcbsrc, "FFTipcbsrc");
long ipcbsrc[2*N];
long ipcb[2*N];

RFFT32_brev_RT_ACQ acq=RFFT32_brev_RT_ACQ_DEFAULTS;

main()
{
      …….
      //Header structure initialization
      acq.buffptr=ipcb;
      acq.tempptr=ipcb;
      acq.size=N;
      acq.count=N;
      acq.acqflag=1;

      for(i=0; i < N; i++)
      {
            acq.input=ipcbsrc[i]; //Input data
            acq.update(&acq);     //One point bit reversed
       }
}
```

Benchmark Information:

Size	C-Callable ASM
1 Block	29 cycles

CFFT32	*32 Bit Complex Fast Fourier Transform Module*

Description This module computes the FFT of N point Complex FFT sequence.



Availability C-Callable Assembly (CcA)

Object Definition

The structure of CFFT32 object is defined by following structure definition

```
typedef struct {
        int32   *ipcbptr;
        int32   *tfptr;
        int16   size;
        int16   nrstage;
        int32   *magptr;
        int32   *winptr;
        int32   peakmag;
        int16   peakfrq;
        int16   ratio;
        void    (*init)(void *);
        void    (*izero)(void *);
        void    (*calc)(void *);
        void    (*mag)(void *);
        void    (*win)(void *);
}CFFT32;
```

Item	Description	Format	Q-values	Comment
ipcbptr	Input data	Pointer to 32-bit integer array	Q31	Computation buffer pointer, this buffer must be aligned to 2Nword (32-bit word size) boundary for N point complex FFT
tfptr	Twiddle factor pointer	Pointer to 32-bit integer array	Q30 or Q31	Not initialized by user. It will assigned by FFT32_init function
size	FFT sample size	Int16	Q0	Must be a power of 2.
nrstage	Number of stages	Int16	Q0	nrstages=log2(size)

magptr	Magnitude output	Pointer to 32-bit integer array	Q30	Magnitude buffer pointer
winptr	Window input	Pointer to 32-bit integer array	Q31	Window coefficient pointer for windowed FFT, only used for real FFT.
peakmag	Peak magnitude	32-bit integer	Not used	Not used.
ratio	Twiddle factor search step	Int16	Q0	ratio=4096/size
init	Member function	Function pointer	N/A	Twiddle factor initialization function
izero	Member function	Function pointer	N/A	Zero imaginary part of specific buffer. Used for CFFT only if want to use CFFT structure to calculate RFFT.
calc	Member function	Function pointer	N/A	FFT calculation function
mag	Member function	Function pointer	N/A	Magnitude calculation function.
win	Member function	Function pointer	N/A	Windowed input function

Special Constants and Data types

CFFT32

The module definition is created as a data type. This makes it convenient to instance an interface to the FFT module. To create multiple instances of the module simply declare variables of type CFFT32

CFFT32_handle

User defined Data type of pointer to CFFT32 Module

CFFT32_xxxP_DEFAULTS: xxx=128, 256, 512 & 1024

Structure symbolic constant to initialize CFFT32 Module to compute “xxx” point complex FFT. This provides the initial values to the terminal variables as well as method pointers.

Methods

void init(CFFT32_handle);
void izero(CFFT32_handle);
void calc(CFFT32_handle);
void mag(CFFT32_handle); (OPTIONAL)
void win(CFFT32_handle); (OPTIONAL)

void init(CFFT32_handle);

The FFT initialization routine updates the twiddle factor pointer with the address of twiddle factor table. Twiddle factors are assembled into “FFTtf” section and contains 768 entries (32-bit words) to facilitate complex FFT computation of upto 1024 points.

void izero(CFFT32_handle);

This function zeros the imaginary part of the complex input sequence in the computation buffer to obtain the FFT of real valued time domain signal.

void calc(CFFT32_handle);

This routine performs radix 2, N point in-place FFT computation on the bit-reversed data sequence (in Q31 format) pointed by the ipcbptr of the FFT module and produce in-order data (in Q31 format) representing frequency domain information. The $1/N$ scaling of FFT is distributed across the stages. Note that the input and output data are in Q31 format. Size of the computation buffer is $2N$ long words (Twice the FFT length).

void win(CFFT32_handle); (OPTIONAL)

This function window the bit reversed data sequence (in Q31 format) in the computation buffer using the window coefficients (in Q31 format) pointed by the *winptr* element of FFT module to reduce the leakage effect. Size of the window coefficient array is $N/2$ long words (1/2 of the FFT length). Note that the windowing function should be invoked only if the computation buffer contains the data sequence in bit reversed order. Window coefficients for hamming, hanning & blackman window to cater to 128, 256, 512, 1024-point complex FFT can be calculated by Matlab. For example, following command will generate length of N=32 hamming window and store the first N/2 of them in win[N/2] array since this window is symmetric.

```
s=floor(2^31*hamming(N));
win=s(1:N/2);
```

Window function is not necessary to be hamming, it can be blackman, hann, kaiser, etc. Please refer “help window” in Matlab command window, user can select whatever they prefer.

void mag(CFFT32_handle); (OPTIONAL)

This routine obtains the Magnitude Square of the complex FFT output (in Q31 format) and stores back the result (in Q30 format) either in the computation buffer or in a dedicated array as commanded by the *magptr* element of the complex FFT module. Note that the magnitude output is stored in Q30 format. The size of the array to hold the magnitude outputs is N long words (Equal to the FFT length).

$$X(k) = X_r(K) + jX_i(k)$$

$$|X(k)|^2 = X_r(k)^2 + X_i(k)^2$$

Alignment Requirements:

The computation buffer should be aligned to $4 \times N$ ***words (16-bit words size) boundary*** or $2 \times N$ long words (32-bit word size), in order to get the samples in bit-reversed order by using the bit-reversal utility CFFT32_brev

Linker Command File (Align computation buffer to 512words for 128 point FFT)

```
FFTipcb ALIGN(512) : { } >       RAML4  PAGE 1
FFTipcbsrc      ALIGH(512)   { } >      RAML5  PAGE 1
FFTmag                         >      RAML6  PAGE 1
FFTtf                          >      RAML7  PAGE 1
```

Example:

For 32 points CFFT user can use following templates to define the header structure.

```
#define CFFT32_32P_DEFAULTS   {
                (int32 *)NULL,
                (int32 *)NULL,
                32,
                5,
                (int32 *)NULL,
                (int32 *)NULL,
                0,
                0,
                128,
                (void (*)(void *))FFT32_init,
                (void (*)(void *))FFT32_izero,
                (void (*)(void *))FFT32_calc,
                (void (*)(void *))FFT32_mag,
                (void (*)(void *))FFT32_win
        }
```

The following sample code obtains the complex FFT value.

```
#include fft.h
#define    N  32
/* Data buffer alignment                              */
#pragma DATA_SECTION(ipcb, "FFTipcb");
#pragma DATA_SECTION(ipcbsrc, "FFTipcbsrc");
long ipcbsrc[2*N];
long ipcb[2*N];
/* Header structure initialization                    */
CFFT32  fft=CFFT32_32P_DEFAULTS;


main()
{
       ……….
       // Generate sample waveforms:
       for(i=0; i < (N*2); i=i+2)
       {
             ipcbsrc[i]  =(long)real[i]       //Q31
             ipcbsrc[i+1] = (long)imag[i];    //Q31
        }
       ……….
       /* Real part bit reversing          */
       CFFT32_brev(ipcbsrc, ipcb, N);
       /* Imaginary part bit reversing     */
       CFFT32_brev(&ipcbsrc[1], &ipcb[1], N);
       fft.ipcbptr=ipcb; /* FFT computation buffer   */
       fft.init(&fft);/* Twiddle factor pointer init */
       fft.calc(&fft);   /* Compute the FFT          */
}
```

Benchmark Information:

32-bit Complex FFT		
FFT size	**Execution Cycles**	
	Case 1 : TF(Q31)	**Case 2 : TF(Q30)**
128	11159	11671
256	25901	27181
512	59075	62146
1024	132823	139991

Notes:
CASE 1: Twiddle factor is in Q31 format and placed in internal memory (Zero wait state access)
CASE 2: Twiddle factor is in Q30 format and placed in internal memory (Zero wait state access)

The section "**FFTtf**" holds 4096 twiddle factors, each 32-bits or 2 words wide. A total of 8192 (0x2000) contiguous words need to be allotted this section in memory. When running in emulation mode it may be necessary to allocate an entire RAM block in the linker command file. For e.g.

```
FFTtf       >       RAML7,      PAGE = 1
```

When running out of FLASH in either emulation or standalone mode the twiddle factors must be stored in flash. Define a flash section of sufficient size (atleast 0x2000) in either page 0 or 1 of the memory map and allocate "**FFTtf**" to it as follows

```
FFTtf       >       FLASHC,     PAGE = 0
```

For better performance or time-critical code, you can optionally copy over the twiddle factors from FLASH to RAM at runtime by defining separate load and run addresses.

In the linker command file define the section **FFTtf** as follows

```
FFTtf       :       LOAD = FLASHC,   PAGE = 0
                    RUN =  RAML7,    PAGE = 1
                    LOAD_START(_FFTtfLoadStart),
                    LOAD_END(_FFTtfLoadEnd),
                    RUN_START(_FFTtfRunStart)
```

In the main C file, declare the following variables

```
extern Uint16 FFTtfLoadStart, FFTtfLoadEnd, FFTtfRunStart;
```

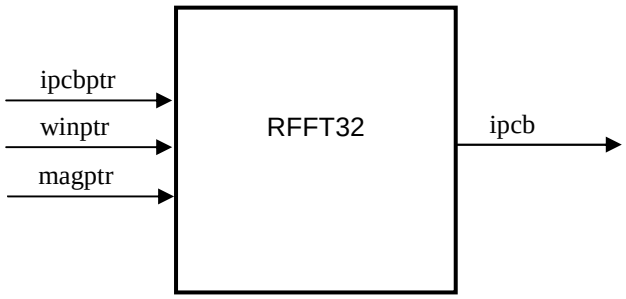
Finally use the MemCopy function to copy over the section from FLASH to RAM

```
MemCopy(&FFTtfLoadStart, &FFTtfLoadEnd, &FFTtfRunStart);
```

RFFT32	*32 Bit Real Fast Fourier Transform Module*

Description

This real FFT module computes the FFT of N -point real sequence using $N/2$ -point complex FFT routine.



Availability

C-Callable Assembly (CcA)

Object Definition

The structure of RFFT32 object is defined by following structure definition

```
typedef struct {
        int32     *ipcbptr;
        int32     *tfptr;
        int16     size;
        int16     nrstage;
        int32     *magptr;
        int32     *winptr;
        int32     peakmag;
        int16     peakfrq;
        int16     ratio;
        void      (*init)(void *);
        void      (*calc)(void *);
        void      (*mag)(void *);
        void      (*win)(void *);
}RFFT32;
```

Item	Description	Format	Q-values	Comment
ipcbptr	Input data	Pointer to 32-bit integer array	***Q31***	Computation buffer pointer, this buffer must be aligned to 2Nword (32-bit word size) boundary for N point complex FFT
tfptr	Twiddle factor pointer	Pointer to 32-bit integer array	Q30 or Q31	Not initialized by user. It will assigned by FFT32_init function
size	FFT sample size	Int16	Q0	Must be a power of 2.
nrstage	Number of stages	Int16	Q0	nrstages=log2(size)

magptr	Magnitude output	Pointer to 32-bit integer array	Q30	Magnitude buffer pointer
winptr	Window input	Pointer to 32-bit integer array	Q31	Window coefficient pointer for windowed FFT, only used for real FFT.
peakmag	Peak magnitude	32-bit integer	Not used	Not used.
ratio	Twiddle factor search step	Int16	Q0	ratio=4096/size
init	Member function	Function pointer	N/A	Twiddle factor initialization function
calc	Member function	Function pointer	N/A	FFT calculation function
mag	Member function	Function pointer	N/A	Magnitude calculation function.
win	Member function	Function pointer	N/A	Not used

Special Constants and Data types

RFFT32
The module definition is created as a data type. This makes it convenient to instance an interface to the FFT module. To create multiple instances of the module simply declare variables of type **RFFT32**

RFFT32_handle
User defined Data type of pointer to **RFFT32** Module

RFFT32_xxxP_DEFAULTS: xxx=128, 256, 512 & 1024
Structure symbolic constant to initialize **RFFT32** Module to compute **"xxx"** point real FFT. This provides the initial values to the terminal variables as well as method pointers.

Methods

void init(**RFFT32_handle**);
void calc(**RFFT32_handle**);
void mag(**RFFT32_handle**); (OPTIONAL)
void win(**RFFT32_handle**); (OPTIONAL)

void init(RFFT32_handle);
The FFT initialisation routine updates the twiddle factor pointer with the address of twiddle factor table. Twiddle factors are assembled into "FFTtf" section and contains 768 entries (32-bit) to fecilitate real FFT computation of upto 1024 points.

void calc(RFFT32_handle);
This routine performs radix 2, $N/2$-point in-place FFT computation on the bit-reversed data sequence (in Q31 format) pointed by the *ipcbptr* of the FFT module and produce in-order data (in Q31 format) representing frequency domain information. The $1/N$ scaling of FFT is distributed across the stages. Note that the input and output data are in Q31 format.

void win(RFFT32_handle); (OPTIONAL)

This function window the bit-reversed complex data sequence (in Q31 format) in the computation buffer using the window coefficients (in Q31 format) pointed by the *winptr* element of FFT module to reduce the leakage effect. Size of the window coefficient array is $N/2$ long words (1/2 of the real FFT length). Note that the window function should be invoked only if the computation buffer contains the data sequence in bit reversed order. Window coefficients for hamming, hanning & blackman window to cater to 128, 256, 512, 1024-point complex FFT can be calculated by Matlab. For example, following command will generate length of N=32 hamming window and store the first N/2 of them in win[N/2] array since this window is symmetric.

```
s=floor(2^31*hamming(N));
win=s(1:N/2);
```

Window function is not necessary to be hamming, it can be blackman, hann, kaiser, etc. Please refer “help window” in Matlab command window, user can select whatever they prefer.

void mag(RFFT32_handle); (OPTIONAL)

This routine obtains the Magnitude Square of $N/2+1$ complex spectral bin obtained from split operation and stores back the result (in Q30 format) either in the computation buffer or in a dedicated array as commanded by the *magptr* element of FFT module. Note that the magnitude output is stored in Q30 format. The size of the array to hold $N/2+1$ magnitude outputs is equal to $N/2+1$ long words (1/2 of the real FFT length + 1).

$$G(k) = G_r(K) + jG_i(k)$$

$$|G(k)|^2 = G_r(k)^2 + G_i(k)^2$$

Alignment Requirements:

The computation buffer should be aligned to $4 \times N$ ***words (16-bit words size) boundary*** or $2 \times N$ long words (32-bit word size), in order to get the samples in bit-reversed order by using the bit-reversal utility RFFT32_brev

Linker Command File (Align computation buffer to 512words for 128 point FFT)

```
FFTipcb ALIGN(512) : { } >       RAML4  PAGE 1
FFTipcbsrc      ALIGH(512)   { } >      RAML5  PAGE 1
FFTmag                        >       RAML6  PAGE 1
FFTtf                         >       RAML7  PAGE 1
```

Example:

For 32 points RFFT user can use following templates to define the header structure.

```
#define RFFT32_32P_DEFAULTS    {
      (int32 *)NULL,
      (int32 *)NULL,
      32,
      5,
      (int32 *)NULL,
      (int32 *)NULL,
      0,
```

```
        0,
        128,
        (void (*)(void *))FFT32_init,
        (void (*)(void *))FFT32_calc,
        (void (*)(void *))FFT32_mag,
        (void (*)(void *))FFT32_win}
```

The following sample code obtains the real FFT value.

```
#define     N   32      //FFT size
//Buffer alignment
#pragma DATA_SECTION(ipcb, "FFTipcb");
#pragma DATA_SECTION(ipcbsrc, "FFTipcbsrc");
long ipcbsrc[2*N];
long ipcb[2*N];

/* Create an Instance of FFT module                 */
RFFT32  fft=RFFT32_32P_DEFAULTS;

main()
{
    ……….
     // Generate sample waveforms:
    for(i=0; i < (N*2); i=i+2)
    {
      ipcbsrc[i]  =(long)real[i] //Q31
      ipcbsrc[i+1] = 0;
    }
    ……….
    RFFT32_brev(ipcbsrc, ipcb, N); /*Bit reverse       */

    fft.ipcbptr=ipcb;   /* FFT computation buffer      */
    fft.init(&fft);     /* Twiddle factor pointer init */
    fft.calc(&fft);     /* Compute the FFT             */
}
```

Benchmark Information:

32-bit Real FFT		
FFT size	**Execution Cycles**	
	Case 1 : TF(Q31)	**Case 2 : TF(Q30)**
128	6509	6763
256	14756	15394
512	33081	34615
1024	73422	77004

Notes:
CASE 1: Twiddle factor is in Q31 format and placed in internal memory (Zero wait state access)
CASE 2: Twiddle factor is in Q30 format and placed in internal memory (Zero wait state access)

The section "**FFTtf**" holds 4096 twiddle factors, each 32-bits or 2 words wide. A total of 8192 (0x2000) contiguous words need to be allotted this section in memory. When running in emulation mode it may be necessary to allocate an entire RAM block in the linker command file. For e.g.

```
FFTtf       >       RAML7,      PAGE = 1
```

When running out of FLASH in either emulation or standalone mode the twiddle factors must be stored in flash. Define a flash section of sufficient size (atleast 0x2000) in either page 0 or 1 of the memory map and allocate "**FFTtf**" to it as follows

```
FFTtf       >       FLASHC,     PAGE = 0
```

For better performance or time-critical code, you can optionally copy over the twiddle factors from FLASH to RAM at runtime by defining separate load and run addresses.

In the linker command file define the section **FFTtf** as follows

```
FFTtf       :       LOAD = FLASHC,   PAGE = 0
                    RUN =  RAML7,    PAGE = 1
                    LOAD_START(_FFTtfLoadStart),
                    LOAD_END(_FFTtfLoadEnd),
                    RUN_START(_FFTtfRunStart)
```

In the main C file, declare the following variables

```
extern Uint16 FFTtfLoadStart, FFTtfLoadEnd, FFTtfRunStart;
```

Finally use the MemCopy function to copy over the section from FLASH to RAM

```
MemCopy(&FFTtfLoadStart, &FFTtfLoadEnd, &FFTtfRunStart);
```

Description

This module implements 1 block (point by point) FIR Filter using DMAC instructions that effectively executes 2 filter taps in a cycle. This module can support up to 255th order FIR filter.

Availability

C-Callable Assembly (CcA)

Header File

`filter.h`

Object Definition

The structure of FIR16 object is defined by following structure definition

```
typedef struct {
        int32   *coeff_ptr;
        int32   * dbuffer_ptr;
        int16   cbindex;
        int16   order;
        int16   input;
        int16   output;
        void    (*init)(void *);
        void    (*calc)(void *);
}FIR16;
```

Item	Description	Format	Q-values	Comment
coeff_ptr	Coefficient pointer	Pointer to 32-bit integer buffer	Q15	Pointer to the Filter coefficient array. Please notice the FIR filter coefficients in this buffer are not arranged in natural order .
dbuffer_ptr	Delay buffer	Pointer to 32-bit integer buffer	Q15	Pointer to the Delay buffer. Please notice the elements in this buffer are not arranged in natural order.
cbindex	Circular buffer index	int16	Q0	Calculated by FIR16_init function in terms of the order (range: 0x00~FE)
order	Order of the filter	int16	Q0	Number of FIR taps, order has to be even number.
input	Input to the filter	int16	Q15	N/A
output	Output of the filter	int16	Q14	N/A

init	Member function	Function pointer	N/A	This initialization function initializes cbindex and clean up delay buffer.
calc	Member function	Function pointer	N/A	This module calculates the FIR filtering using dual MAC instruction DMAC.

Special Constants and Data types

FIR16
The module definition is created as a data type. This makes it convenient to instance an interface to the FIR16 Filter module. To create multiple instances of the module simply declare variables of type FIR16

FIR16_handle
User defined Data type of pointer to FIR16 Module

FIR16_DEFAULTS
Structure symbolic constant is to initialize FIR16 Module. This provides the initial values to the terminal variables as well as method pointers.

Methods

void init(FIR16_handle);
void calc(FIR16_handle);
These definitions implements two methods viz., the initialization and FIR filter computation function. The input argument to these functions is the module handle

Note:

1. **"firldb" section should be aligned to 256 word boundary in the data memory (RAM)**
2. **"firfilt" section can be placed anywhere in the data memory (RAM)**

Background Information

In general, an Nth order FIR filter is described by the difference equation

$$y(n) = \sum_{k=0}^{N} h_k \times x(n-k)$$

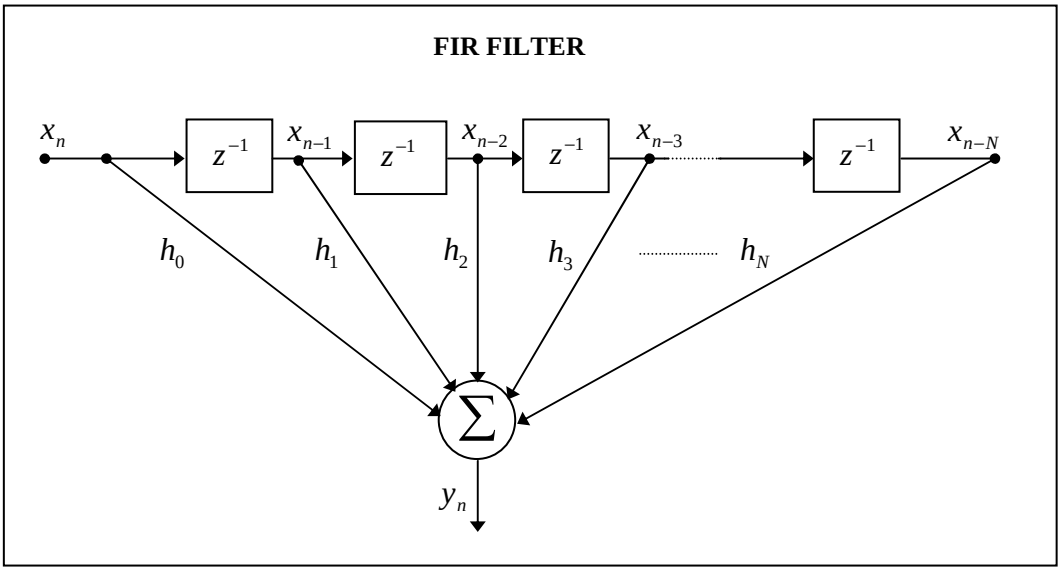
or, equivalently, by the system function

$$H(z) = \sum_{k=0}^{N} h_k \times z^{-k}$$

Furthermore, the unit sample response of the FIR system is identical to the coefficients $h(k)$, that is

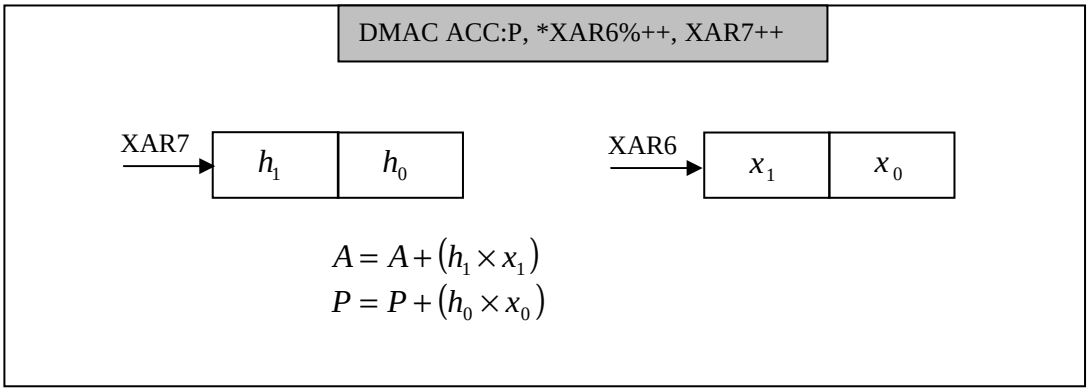
$$h(n) = \begin{cases} h_n, & 0 < n < N \\ 0, & otherwise \end{cases}$$

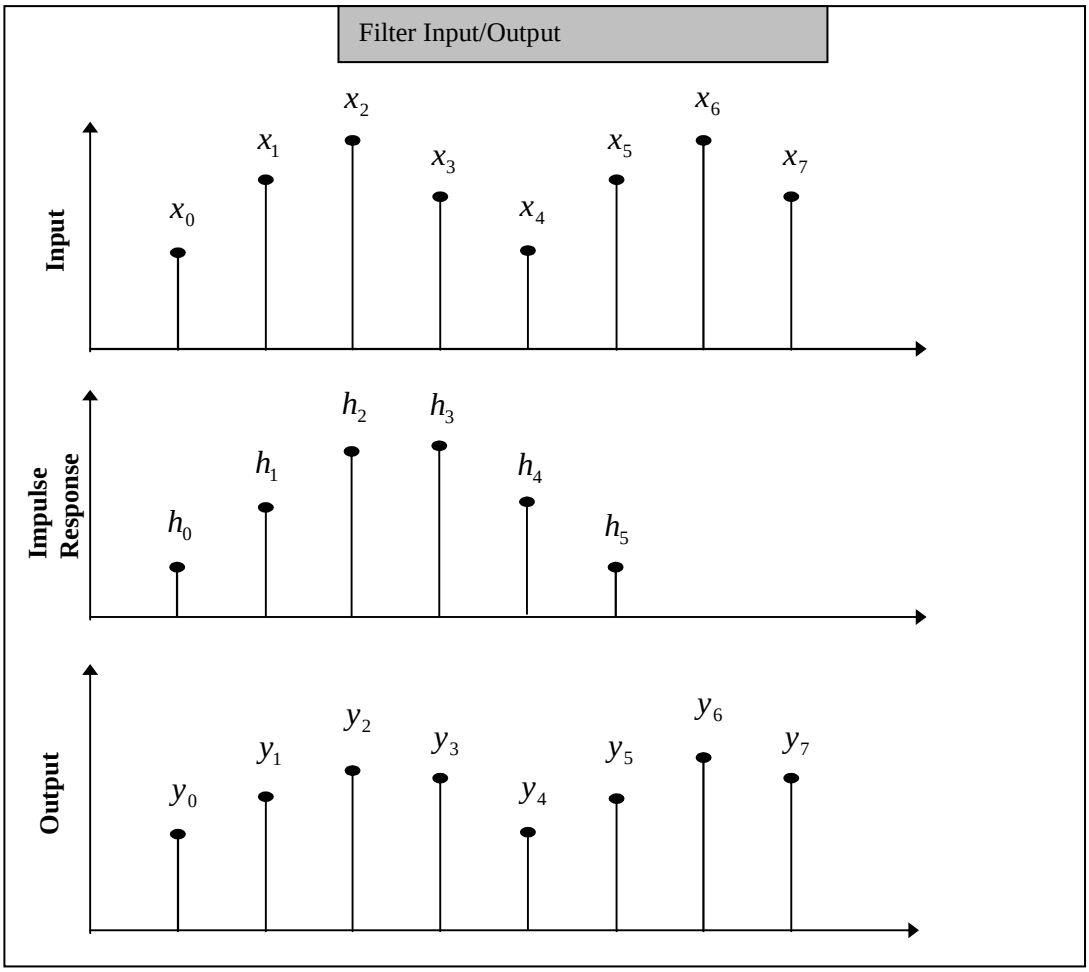
The signal flow diagram of the FIR filter, representing the above mentioned difference equation is given below. The following Tapped delay implementation is canonical, which means that the number of storage element in the structure is equal to the order of the Filter.

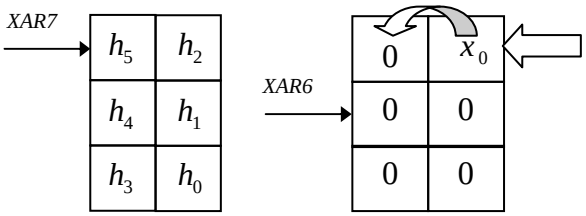


The FIR filter is essentially a sum of product operating on an array of values maintained in a delay line. Note that the number of filter co-efficients will always be greater then the order of the filter by 1.

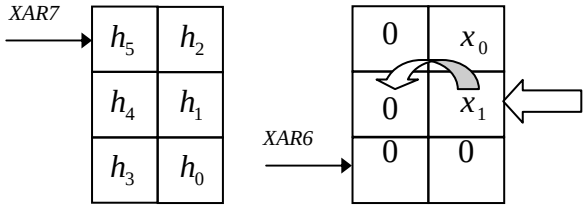
The following diagram depicts the filter tap computation using DMAC instruction, computation proceeds from the oldest values to the newest and the DMAC instruction computes 2 filter taps per cycle. Delay buffer is addressed using circular addressing scheme that will wrap around the pointer to the beginning of the pointer when the pointer reaches the last location of delay buffer. Note that C28x circular buffer addressing requires the delay buffer to be aligned to 256words location and the delay buffer length is restricted to 256words. Hence, this FIR filter module allows FIR filter implementation of up to 255th order.



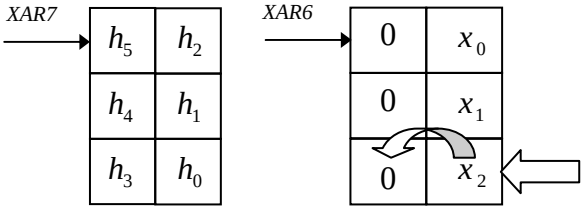




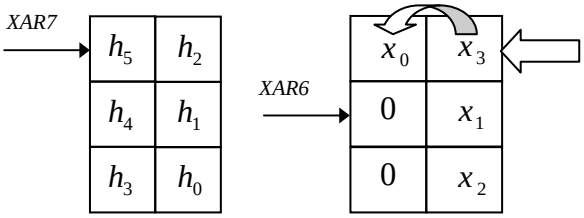
$$A = h_5 \times 0 + h_4 \times 0 + h_3 \times 0$$
$$P = h_2 \times 0 + h_1 \times 0 + h_0 \times x_0$$
$$y_0 = A + P$$



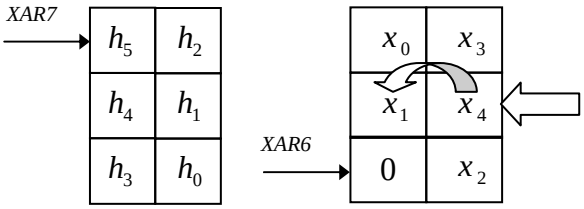
$$A = h_5 \times 0 + h_4 \times 0 + h_3 \times 0$$
$$P = h_2 \times 0 + h_1 \times x_0 + h_0 \times x_1$$
$$y_1 = A + P$$



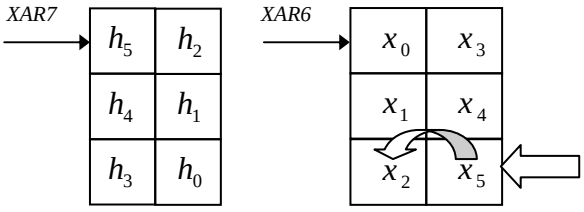
$$A = h_5 \times 0 + h_4 \times 0 + h_3 \times 0$$
$$P = h_2 \times x_0 + h_1 \times x_1 + h_0 \times x_2$$
$$y_2 = A + P$$



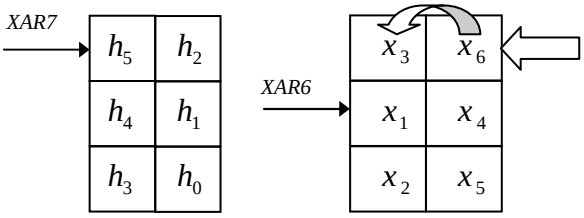
$$A = h_5 \times 0 + h_4 \times 0 + h_3 \times x_0$$
$$P = h_2 \times x_1 + h_1 \times x_2 + h_0 \times x_3$$
$$y_3 = A + P$$



$$A = h_5 \times 0 + h_4 \times x_0 + h_3 \times x_1$$
$$P = h_2 \times x_2 + h_1 \times x_3 + h_0 \times x_4$$
$$y_4 = A + P$$



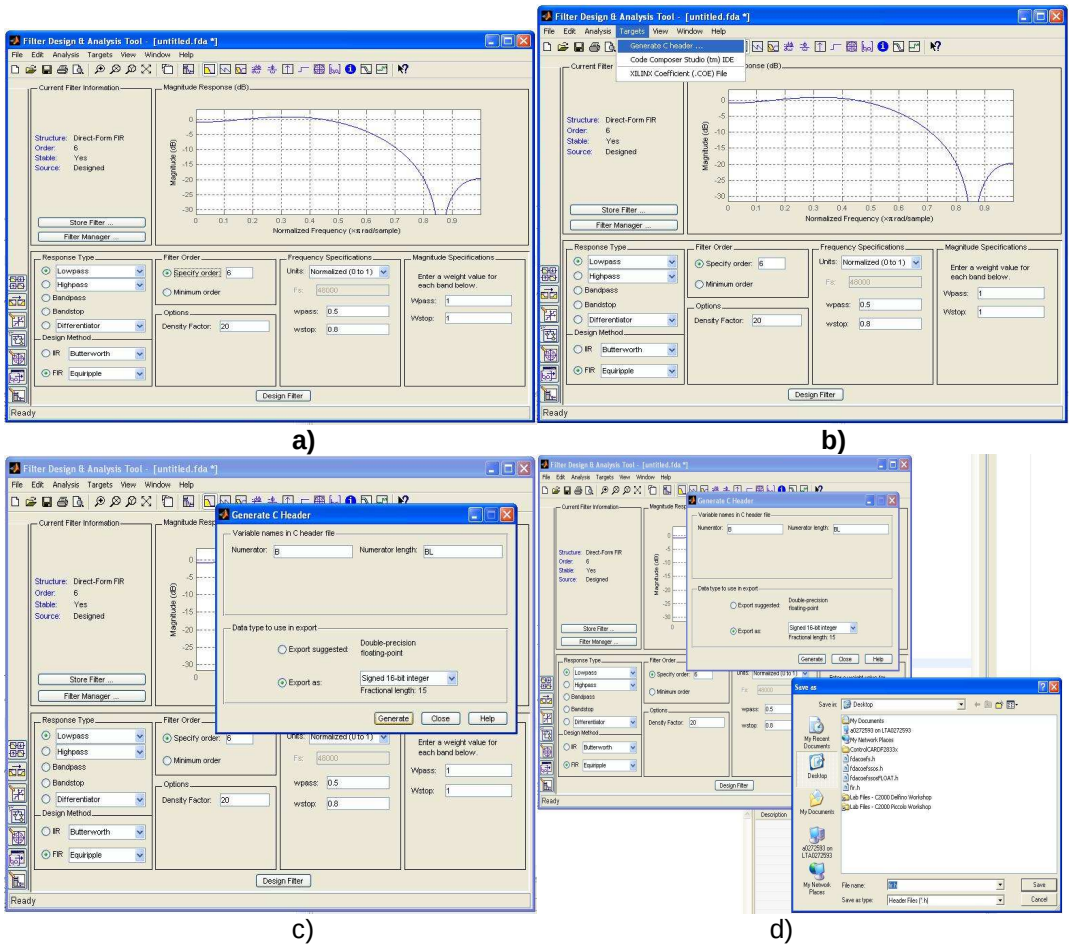
$$A = h_5 \times x_0 + h_4 \times x_1 + h_3 \times x_2$$
$$P = h_2 \times x_3 + h_1 \times x_4 + h_0 \times x_5$$
$$y_5 = A + P$$



$$A = h_5 \times x_1 + h_4 \times x_2 + h_3 \times x_3$$
$$P = h_2 \times x_4 + h_1 \times x_5 + h_0 \times x_6$$
$$y_6 = A + P$$

Coefficients generation:

The FIR coefficients header file can be generated by MATLAB Filter Design and Analysis Tool (FDATool). Users can port these header files directly to the “fir.h”. There are four steps of design and generating user specified coefficients.



FIR coefficient header file generation

a) Open Matlab command window, click “fdatool”, the FDATool GUI will pop out. Select the parameters, filter type and other options which you prefer, click Design Filter. The design process is finished.
b) Go to Targets-> Generate C header;
c) After b), the Generate C header dialogue box pops out and in ‘Data type to use in export’, check ‘Export as’ Signed 16-bit or 32-bit integer (depends on 16 bit FIR or 32 bit FIR). Click ‘Generate’;
d) Save the header file. The coefficient can directly port to ‘fft.h’.

NOTE:
The module assumes the FIR taps number is even. If user wants to calculate odd taps FIR filter, one zero should be added to the FIR coefficients to make it become even taps FIR filter.

Example:

In the example project associated with FIR16_calc, users don't need to worry about the order of coefficient since one section of the code will arrange it for user.

The following sample code obtains the FFT of the real input.

```
#include <fir.h>
#define FIR_ORDER   32
#define SIGNAL_LENGTH 100
/* Header structure buffer alignment      */
#pragma DATA_SECTION(fir, "firfilt");
FIR16  fir= FIR16_DEFAULTS;
/* Delay buffer alignment                 */
#pragma DATA_SECTION(dbuffer,"firldb");
long dbuffer[(FIR_ORDER+3)/2];
……….
/* Coefficients buffer alignment          */
#pragma DATA_SECTION(coeff, "coefffilt");
long const coeff[(FIR_ORDER+3)/2];
/* Coefficients array                     */
int FIR16_LPF32_TEST[FIR_ORDER+1];


void main()
{
      /* FIR Generic Filter Initialisation    */
      fir.order=FIR_ORDER;
      fir.dbuffer_ptr=dbuffer;
      fir.coeff_ptr=(long *)coeff;
      fir.init(&fir);
      ……..
      /* Reorganize coefficients array        */
      for(i=0;i<FIR_ORDER_REV;i=i+2)
      {
      *(p+FIR_ORDER-i-2)=FIR16_LPF32_TEST[i/2];
      *(p+FIR_ORDER-i-1)=FIR16_LPF32_TEST[i/2+FIR_ORDER/2];
      }
      ……..
      /* FIR filter calculation               */
      for(i=0; i < SIGNAL_LENGTH; i++)
      {
            fir.input=xn
            fir.calc(&fir);
            yn=fir.output;
       }
}
```

Benchmark Information:

FIR Taps (1 block)	C-Callable ASM
8	54 cycles
16	58 cycles
32	66 cycles
64	82 cycles
128	114 cycles

Description This module implements FIR Filter using QMACL instructions that effectively executes 1 filter tap in a cycle. This module can support up to 255th order FIR filter.

Availability C-Callable Assembly (CcA)

Header File `filter.h`

Object Definition The structure of FIR32 object is defined by following structure definition

```
typedef struct {
        int32   *coeff_ptr;
        int32   * dbuffer_ptr;
        int16   cbindex;
        int16   order;
        int32   input;
        int32   output;
        void    (*init)(void *);
        void    (*calc)(void *);
}FIR32;
```

Item	Description	Format	Q-values	Comment
coeff_ptr	Coefficient pointer	Pointer to 32-bit integer buffer	Q31	Pointer to the Filter coefficient array. 32 bit FIR coefficients are arranged in natural order.
dbuffer_ptr	Delay buffer	Pointer to 32-bit integer buffer	Q31	Pointer to the Delay buffer. 32-b it elements in delay buffer are arranged in natural order.
cbindex	Circular buffer index	Int16	Q0	Calculated by FIR32_init function in terms of the order (range: 0x00~FE)
order	Order of the filter	Int16	Q0	Number of FIR taps, order not has to be even number.
input	Input to the filter	Int32	Q31	N/A
output	Output of the filter	Int32	Q30	N/A
init	Member function	Function pointer	N/A	This initialization function initializes cbindex and clean up delay buffer.
calc	Member function	Function pointer	N/A	This module calculates the FIR filtering using QMACL instruction.

Special Constants and Data types

FIR32
The module definition is created as a data type. This makes it convenient to instance an interface to the FIR32 Filter module. To create multiple instances of the module simply declare variables of type FIR32

FIR32_handle
User defined Data type of pointer to FIR32 Module

FIR32_DEFAULTS
Structure symbolic constant is to initialize FIR32 Module. This provides the initial values to the terminal variables as well as method pointers.

Methods

void init(FIR32_handle);
void calc(FIR32_handle);
These definitions implements two methods viz., the initialization and FIR filter computation function. The input argument to these functions is the module handle

Coefficients generation:
32 bit FIR filter coefficients generation is the same as 16 bit FIR. Generation method can be referred to 16 bit FIR filter Coefficients generation section.

Note:

1. **"firldb" section should be aligned to 256word boundary in the data memory (RAM)**
2. **"firfilt" section can be placed anywhere in the data memory (RAM)**

Example

The following sample code obtains the scaled FFT magnitude.

```
#include <fir.h>

/* Filter Symbolic Constants              */
#define FIR_ORDER   32
#define SIGNAL_LENGTH 1000
/* Header structure alignment             */
#pragma DATA_SECTION(fir, "firfilt");
FIR32  fir= FIR32_DEFAULTS;
/* Delay buffer alignment                 */
#pragma DATA_SECTION(dbuffer,"firldb");
long dbuffer[FIR_ORDER+1];
……..
/* Coefficient buffer alignment           */
#pragma DATA_SECTION(coeff, "coefffilt");
const long coeff[FIR_ORDER]=FIR32_LPF32_TEST;
```

```
void main()
{
      ……….
      /* FIR Generic Filter Initialisation    */
      fir.order=FIR_ORDER;
      fir.dbuffer_ptr=dbuffer;
      fir.coeff_ptr=(long *)coeff;
      fir.init(&fir);
      ……….
      /* FIR filter calculation               */
      for(i=0; i < SIGNAL_LENGTH; i++)
      {
            fir.input=xn;//0.5 Q15 format
            fir.calc(&fir);
            yn=fir.output;
       }
}
```

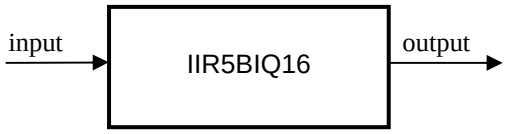
Benchmark Information:

FIR Taps (1 block)	C-Callable ASM
8	55 cycles
16	63 cycles
32	79 cycles
64	111 cycles

IIR5BIQ16	*16 Bit IIR Filter Module*

Description

This module implements Direct II Form cascade Second Order Sections (SOS) IIR filter structure using “biquad” with 16-bit delay line.



Availability

C-Callable Assembly (CcA)

Header File

`filter.h`

Object Definition

The structure of IIR5BIQ16 object is defined by following structure definition

```
typedef struct {
        void    (*init)(void *);
        void    (*calc)(void *);
        int32   *coeff_ptr;
        int32   *dbuffer_ptr;
        int16   nbiq;
        int16   input;
        int16   isf;
        int16   qfmat;
        int16   output;
}IIR5BIQ16;
```

Item	Description	Format	Q-values	Comment
init	Member function	Function pointer	N/A	Delay buffer clean up and initialization function
calc	Member function	Function pointer	N/A	IIR filter calculation function
`coeff_ptr`	Coefficients pointer	Pointer to 16-bit integer array	Q15	Pointer to the Filter coefficient array. Arrangement can be referred to “Background Information” section.
`dbuffer_ptr`	Delay buffer	Pointer to 16-bit integer array	Q15	Delay buffer arrangement can be referred to “Background Information” section.
`nbiq`	Number of bi-quad elements	int16	Q0	Number of bi-quad (SOS) elements. Calculated by eziir16.m
`input`	Input data	int16	Q15	Should be normalized to Q15 format.
`isf`	Input scaling coefficients	int16	Q15	Dynamic adjustment coefficients in order to prevent from out of dynamic range. Calculated by eziir16.m

qfmat	Q format value	int16	Q0	Q format chosen to be run in order to prevent from out of dynamic range. Calculated by eziir16.m
output	Output data	int16	Q14	Output data converted into Q14.

Special Constants and Data Types

IIR5BIQ16
The module definition is created as a data type. This makes it convenient to instance an interface to the IIR Filter module. To create multiple instances of the module simply declare variables of type IIR5BIQ16.

IIR5BIQ16_handle
User defined Data type of pointer to IIR5BIQ16 Module

IIR5BIQ16_DEFAULTS
Structure symbolic constant is to initialize IIR5BIQ16 Module. This provides the initial values to the terminal variables as well as method pointers.

Methods

void calc(IIR5BIQ16_handle)
void init(IIR5BIQ16_handle)
These definitions implements two methods viz., the initialization and IIR filter computation function. The input argument to these functions is the module handle

Coefficients generation:

The IIR coefficients are generated by 'eziir16.m'. Details can be referred to eziir_matlab.pdf.

Note:
For generating IIR coefficients, IIR5BIQ16 module cannot work independently from Matlab scripts "eziir16.m" (<base>\examples_ccsv4\2833x_FixedPoint_IIR16\matlab). User can run this script to get specified filter coefficients and copied it to the header file "iir.h".

Background Information

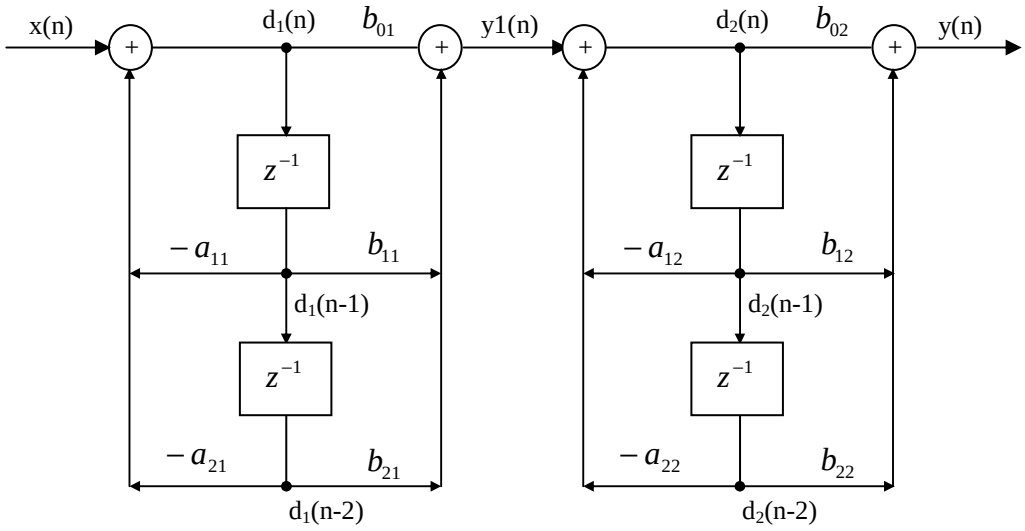
In general, an IIR filter is described by the difference equation

$$y(n) \quad = \quad -\sum_{k=1}^{N} a_k \times y(n-k) \quad + \quad \sum_{k=0}^{M} b_k x(n-k)$$

or, equivalently, by the system function

$$H(z) = \frac{\sum_{k=0}^{M} b_k \times z^{-k}}{1 - \sum_{k=1}^{N} a_k \times z^{-k}}$$

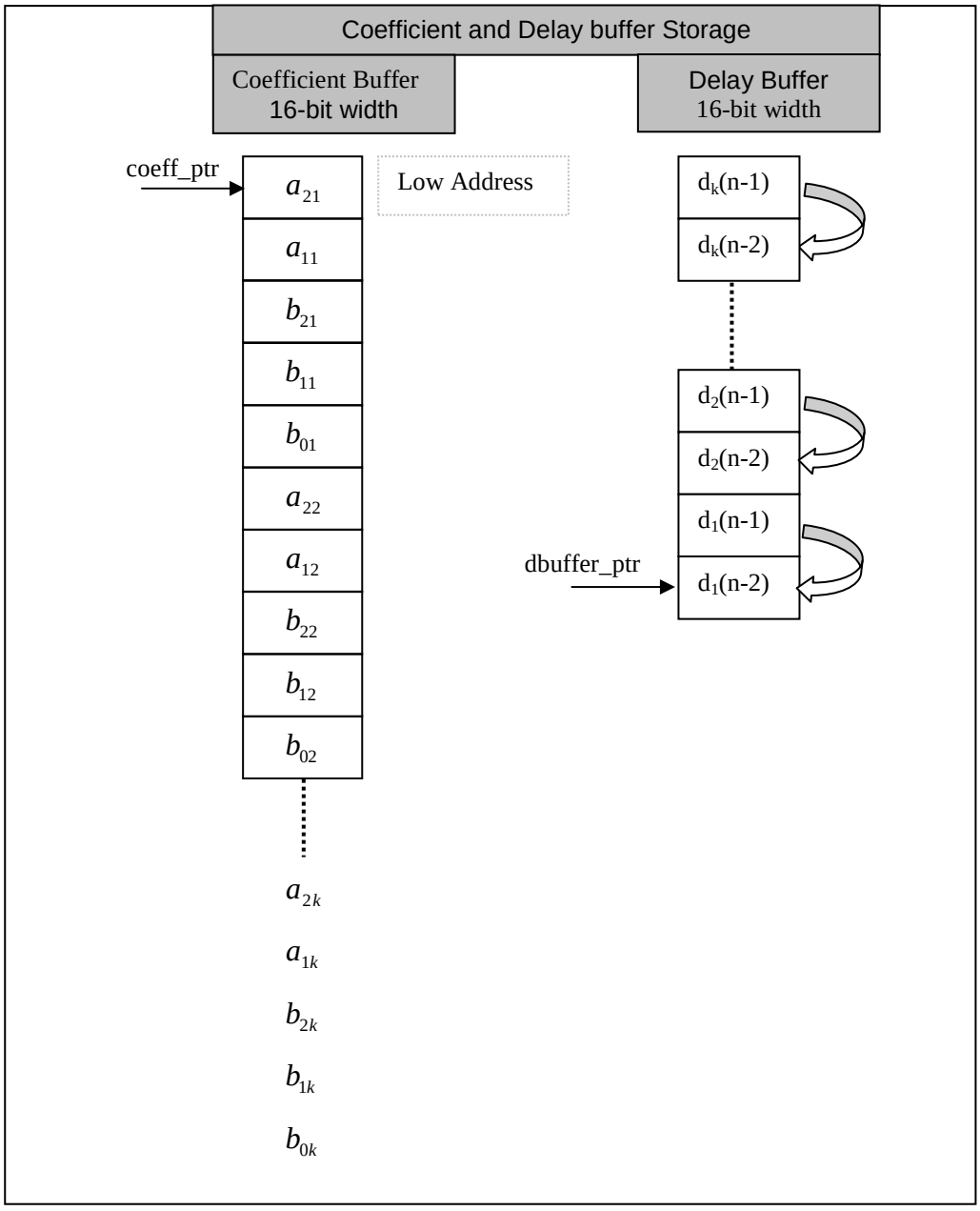
Problems of implementing the above system with finite precision arithmetic have motivated the development of various filter structures. Direct Form I & II implementation of IIR filter is extremely sensitive to parameter quantization, in general, and are not recommended in practical applications. It has been shown that breaking up the transfer function into lower-order sections and connecting these in cascade or parallel can reduce this sensitivity to coefficient quantization. Hence, we choose to use cascade configuration of direct form II structured Second order section (SOS) in our implementation. The SOS's are commonly referred to as Biquads. The following block diagram shows the Cascade implementation of 4th order IIR filter using two biquads.



The SOS coefficients generated by the MATLAB for given set of filter specification provides unity gain in the pass-band and attenuates the remaining frequency component. Though the input to output gain does not peak above unity, the intermediate node gain in the biquad sections would vary significantly depending on the filter characteristics. The user should devise a technique to limit the peak gain at all the intermediate nodes to unity in order to avoid the overflow.

The first step is to identify the node gains in each SOS with respect to the input and then scale the input of each subsection appropriately to avoid the overflow. Scaling the input to each section is equivalent to scaling the 'b' coefficients of the preceding section. We have developed the "MATLAB" script to design the IIR filter without overflow issues. The

script carry outs the node gain analysis and scales the signal level at various points by scaling the “b” coefficients to limit the gain at all the node in the filter to unity. The scaled SOS coefficients, Input Scaling factor, the Q format used to represent coefficients and number of biquad used to obtain the requested filter characteristics are stored in a file, to initialize the IIR5BIQ16 filter module. Input Scaling factor limits the gain at the first node of the first biquad to unity. The user must use the eziir16 filter design scrip to generate filter co-efficient for this module.



Example: The following sample code obtains the filtered result with 16 bit IIR filter.

```
#include <filter.h>

/* Filter Symbolic Constants          */
#define SIGNAL_LENGTH 1000

#pragma DATA_SECTION(iir, "iirfilt");
IIR5BIQ32  iir=IIR5BIQ16_DEFAULTS;

#pragma DATA_SECTION(dbuffer,"iirfilt");
long dbuffer[2*IIR16_LPF_NBIQ];
……..
const long coeff[5*IIR16_LPF_NBIQ]=IIR16_LPF_COEFF;

void main()
{
      ……..
      /* IIR Filter Initialisation  */
      iir.dbuffer_ptr=dbuffer;
      iir.coeff_ptr=(long *)coeff;
      iir.qfmat=IIR16_LPF_QFMAT;
      iir.nbiq=IIR16_LPF_NBIQ;
      iir.isf=IIR16_LPF_ISF;
      iir.init(&iir);

      /* IIR Filter calculation    */
      for(i=0; i < SIGNAL_LENGTH; i++)
      {
            iir.input=xn;
            iir.calc(&iir);
            yn=iir.output32;
       }
}
```

Benchmark Information:

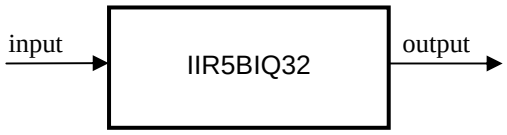
NBIQ	C-Callable ASM
1	23 cycles
2	49 cycles
4	103 cycles
8	190 cycles

Note: All buffers and stack are placed in internal memory (zero-wait states in data space).

IIR5BIQ32	*32 Bit IIR Filter Module*

Description

This module implements Direct II Form cascade Second Order Sections (SOS) IIR filter structure using “biquad” with 32-bit delay line.



Availability

C-Callable Assembly (CcA)

Header File

`filter.h`

Declaration

```
VOID IIR5BIQ32_calc (void *)
```

Object Definition

The structure of IIR5BIQ32 object is defined by following structure definition

```
typedef struct {
        void    (*init)(void *);
        void    (*calc)(void *);
        int32   *coeff_ptr;
        int32   *dbuffer_ptr;
        int32   nbiq;
        int32   input;
        int32   isf;
        int32   output32;
        int16   output16;
        int16   qfmat;
}IIR5BIQ32;
```

Item	Description	Format	Q-values	Comment
init	Member function	Function pointer	N/A	Delay buffer clean up and initialization function
calc	Member function	Function pointer	N/A	IIR filter calculation function
`coeff_ptr`	Coefficients pointer	Pointer to 32-bit integer array	Q31	Details are in “Background Information” section.
`dbuffer_ptr`	Delay buffer	Pointer to 32-bit integer array	Q31	Details are in “Background Information” section
`nbiq`	Number of bi-quad elements	int32	Q0	Number of bi-quad (SOS) elements. Calculated by eziir32.m

input	Input data	int32	Q31	Should be normalized to Q31 format.
isf	Input scaling coefficients	int32	Q31	Dynamic adjustment coefficients in order to prevent from out of dynamic range. Calculated by eziir32.m
qfmat	Q format value	int32	Q0	Q format chosen to be run in order to prevent from out of dynamic range. Calculated by eziir32.m
Output32	Output data	int32	Q31	Output data in Q31 format
Output16	Output data	int16	Q14	Output data in Q14 format

Special Constants and Data Types

IIR5BIQ32
The module definition is created as a data type. This makes it convenient to instance an interface to the IIR Filter module. To create multiple instances of the module simply declare variables of type IIR5BIQ32.

IIR5BIQ32_handle
User defined Data type of pointer to IIR5BIQ16 Module

IIR5BIQ32_DEFAULTS
Structure symbolic constant to initialize IIR5BIQ16 Module. This provides the initial values to the terminal variables as well as method pointers.

Methods

void calc(IIR5BIQ32_handle)
void init(IIR5BIQ32_handle)
These definitions implements two methods viz., the initialization and IIR filter computation function. The input argument to these functions is the module handle

Coefficients generation:

The IIR coefficients are generated by 'eziir32.m'. Details can be referred to eziir_matlab.pdf.

Note:
For generating IIR coefficients, IIR5BIQ32 module cannot work independently from Matlab scripts "eziir32.m" (<base>\examples_ccsv4\2833x_FixedPoint_IIR16\matlab).. User can run this script to get specified filter coefficients and copied it to the header file "iir.h".

Background Information

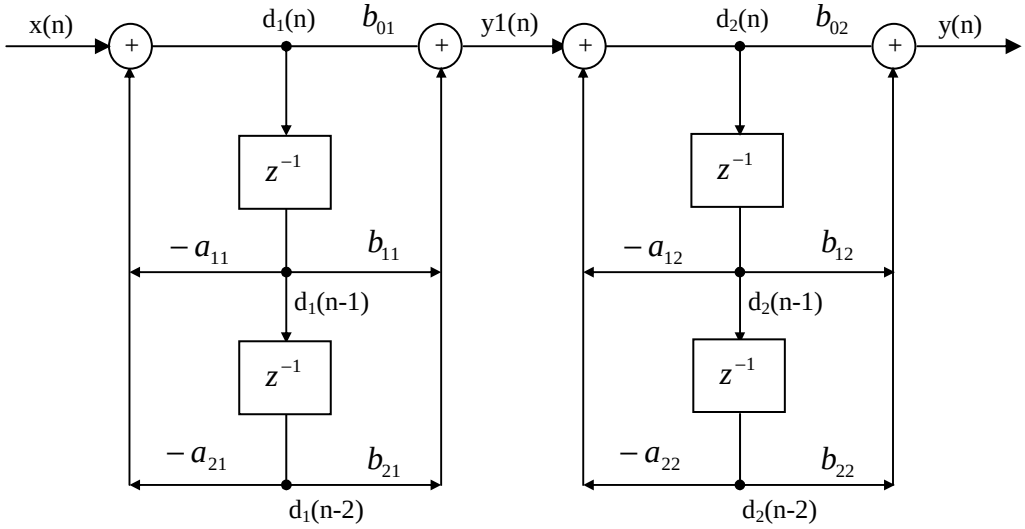
In general, an IIR filter is described by the difference equation

$$y(n) \quad = \quad -\sum_{k=1}^{N} a_k \times y(n-k) \quad + \quad \sum_{k=0}^{M} b_k x(n-k)$$

or, equivalently, by the system function

$$H(z) = \frac{\sum_{k=0}^{M} b_k \times z^{-k}}{1 - \sum_{k=1}^{N} a_k \times z^{-k}}$$

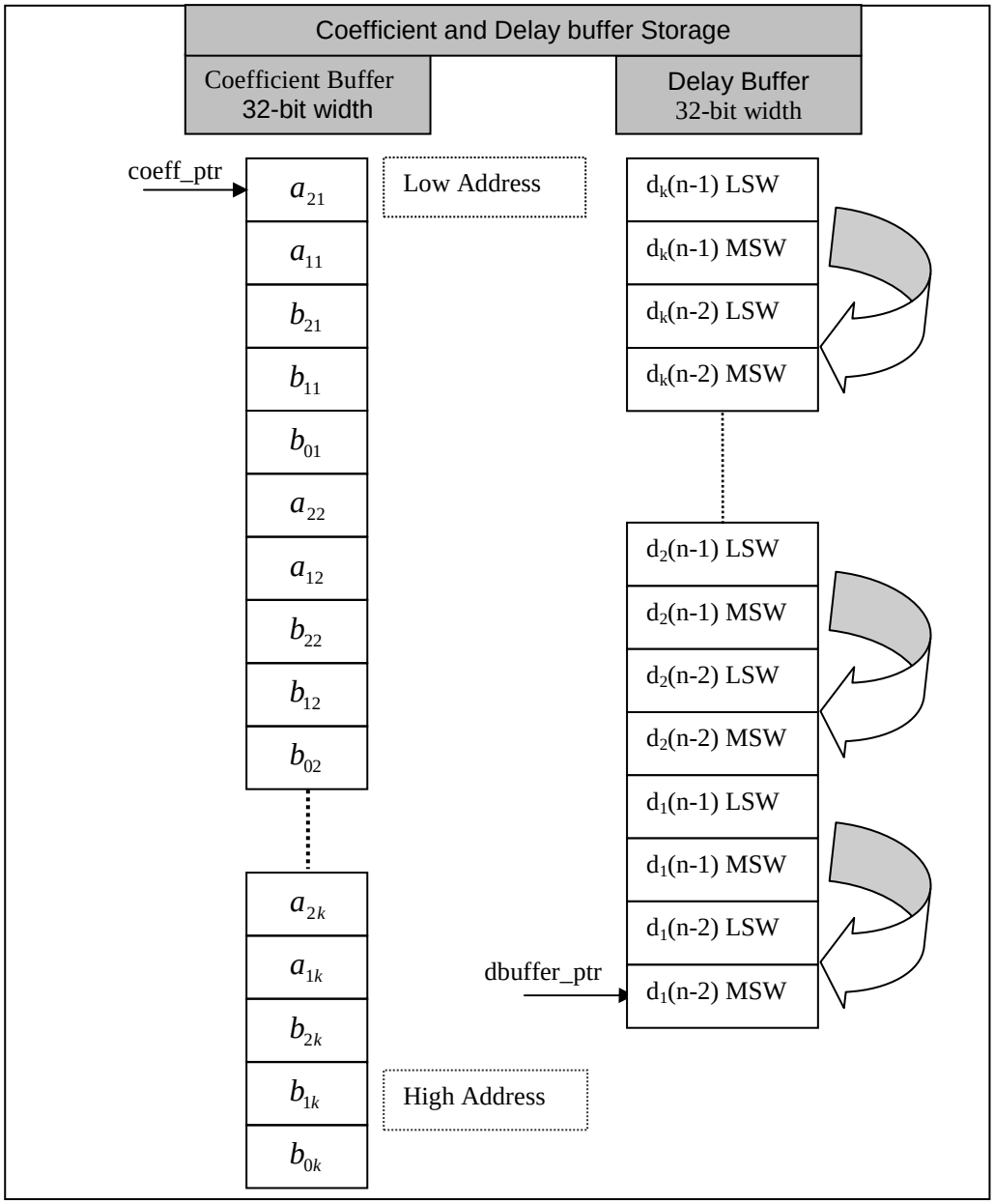
Problems of implementing the above system with finite precision arithmetic have motivated the development of various filter structures. Direct Form I & II implementation of IIR filter is extremely sensitive to parameter quantization, in general, and are not recommended in practical applications. It has been shown that breaking up the transfer function into lower-order sections and connecting these in cascade or parallel can reduce this sensitivity to coefficient quantization. Hence, we choose to use cascade configuration of direct form II structured Second order section (SOS) in our implementation. The SOS's are commonly referred to as Biquads. The following block diagram shows the Cascade implementation of 4th order IIR filter using two biquads.



The SOS coefficients generated by the MATLAB for given set of filter specification provides unity gain in the pass-band and attenuates the remaining frequency component. Though the input to output gain does not peak above unity, the intermediate node gain in the biquad sections would vary significantly depending on the filter characteristics. The user should devise a technique to limit the peak gain at all the intermediate nodes to unity in order to avoid the overflow.

The first step is to identify the node gains in each SOS with respect to the input and then scale the input of each subsection appropriately to avoid the overflow. Scaling the input to each section is equivalent to scaling the 'b' coefficients of the preceding section. We

have developed the “MATLAB” script to design the IIR filter without overflow issues. The script carry outs the node gain analysis and scales the signal level at various points by scaling the “b” coefficients to limit the gain at all the node in the filter to unity. The scaled SOS coefficients, Input Scaling factor, the Q format used to represent coefficients and number of biquad used to obtain the requested filter characteristics are stored in a file, to initialize the IIR5BIQ32 filter module. Input Scaling factor limits the gain at the first node of the first biquad to unity. The user must use the eziir32 filter design scrip to generate filter co-efficient for this module.



Example: The following sample code obtains the filtered result with 32 bit IIR filter.

```
#include <filter.h>

/* Filter Symbolic Constants                          */
#define SIGNAL_LENGTH 1000

/* Create an Instance of IIR5BIQD32 module and
place the object in "iirfilt" section    */
#pragma DATA_SECTION(iir, "iirfilt");
IIR5BIQ32  iir=IIR5BIQ32_DEFAULTS;

/* Define the Delay buffer for the cascaded 6 biquad IIR
filter and place it in "iirfilt" section */
#pragma DATA_SECTION(dbuffer,"iirfilt");
long dbuffer[2*IIR32_LPF_NBIQ];
……..
const long coeff[5*IIR32_LPF_NBIQ]=IIR32_LPF_COEFF;

void main()
{

      /* IIR Filter Initialisation      */
      iir.dbuffer_ptr=dbuffer;
      iir.coeff_ptr=(long *)coeff;
      iir.qfmat=IIR32_LPF_QFMAT;
      iir.nbiq=IIR32_LPF_NBIQ;
      iir.isf=IIR32_LPF_ISF;
      iir.init(&iir);

      /* Calculation section            */
      for(i=0; i < SIGNAL_LENGTH; i++)
      {
            iir.input=xn;
            iir.calc(&iir);
            yn=iir.output32;
      }
}
```

Benchmark Information:

NBIQ	C-Callable ASM
1	22 cycles
2	48 cycles
4	110 cycles
8	202 cycles

5. Revision History

Beta 1 – May 7, 2002

- First release. Includes the complex, real FFT, 16 bit FIR, 16 and 32 bit IIR fixed point filter library.

v1.00 – November 1, 2010

- Second release. Includes the complex FFT and real FFT and 16 bit and 32 bit FIR/IIR fixed point filter library.

v1.01 – January 10, 2010

- Minor revision. Added section on running FFTs out of RAM in standalone mode. Library re-compiled with fpu support.