Fixed Point DSP Software Library

USER'S GUIDE

Copyright

Texas Instruments
13905 University Boulevard
Sugar Land, TX 77479
http://www.ti.com/c2000

Revision Information

This is version 1.20.00.00 of this document, last updated on Mon May 27 06:57:24 CDT 2019.

Table of Contents

1 Introduction

The Texas Instruments TMS320C28x Fixed Point DSP Library is a 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.

Chapter 2 provides a host of resources on the C28x in general, as well as training material.

Chapter 3 describes the directory structure of the package.

Chapter 4 provides step-by-step instructions on how to integrate the library into a project and use any of the math routines.

Chapter 5 describes the structures and programming interface for this library

Chapter 6 lists the revision history of the library.

Examples have been provided for each library routine. They can be found in the *examples_ccsv5* directory. For the current revision, all examples have been written for the *F2833x* and tested on a *controlCard* platform. Each example has a script **"SetupDebugEnv.js"** that can be launched from the *Scripting Console* in CCS. These scripts will set-up the watch variables for the example. In some examples graphs (.graphProp) are provided; these can be imported into CCS during debug.

2 Other Resources

The user can get answers to F2833x frequently asked questions(FAQ) from the processors wiki page. Links to other references such as training videos will be posted here as well. http://processors.wiki.ti.com/index.php/Main_Page

Also check out the TI Delfino page: http://www.ti.com/delfino

And don't forget the TI community website: http://e2e.ti.com

In order to build the library and examples you will require **Codegen Tools v6.2.5 or later**. The tools can be obtained either through a CCS update or through this website: http://processors.wiki.ti.com/index.php/Compiler_Releases

The examples require the **F2833x** device support files in C2000Ware, which may be found at "~/C2000Ware_X_XX_XX_XX/device_support/f2833x"

3 Library Structure

As installed, the C28x Fixed Point Library is partitioned into a well-defined directory structure. The library and source code is installed into the controlSUITE directory,

```
C:\ti\c2000\C2000Ware_X_XX_XX_XX\libraries\dsp\FixedPoint\c28
```

Figure. 3.2 shows the directory structure while the subsequent table 3.1 provides a description for each folder.

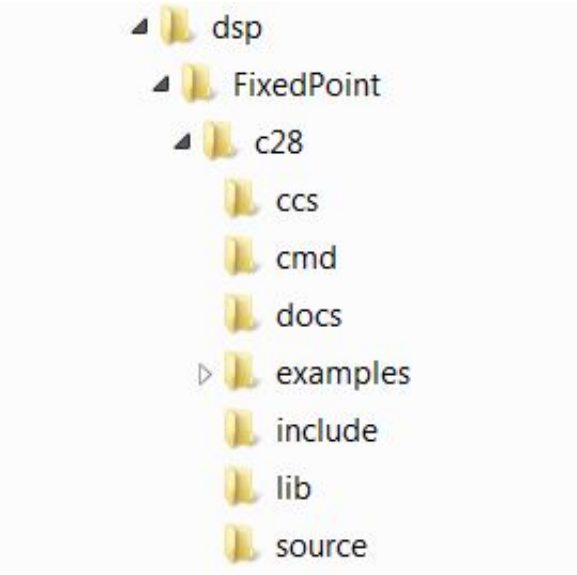


Figure 3.1: Directory Structure of the Fixed Point DSP Library

Folder	Description
<base>	Base install directory. By default this is C:/ti/c2000/C2000Ware_X_XX_XX_XX/libraries/dsp/FixedPoint/c28. For the rest of this document <base> will be omitted from the directory names.
<base>/ccs	Project files for the library. Allows the user to reconfigure, modify and re-build the library to suit their particular needs.
<base>/cmd	Linker command files used in the examples.
<base>/docs	Documentation for the current revision of the library including revision history.
<base>/examples	Examples that illustrate the library functions.
<base>/examples/matlab	MATLAB reference code. These are useful as they provide a standard input/output reference that the user can check against while debugging.
<base>/include	Header files for the Fixed Point library. These include function prototypes and structure definitions.
<base>/lib	Pre-built Fixed Point libraries.
<base>/source	Source files for the library.
<base>/test	Test framework for the library routines.

Table 3.1: Fixed Point Library Directory Structure Description

3.1 Build Options used to build the library

The CCS project for the library was built with C28x Codegen Tools v6.2.5 and has two build configurations

1. ISA_C2800, standard build configuration
2. ISA_C28FPU32, this variant of the library must be used in projects that have the **fpu32** support turned on. This does not mean that the library routines use floating point numbers, rather, it gives the user the option to use fixed point routines alongside floating point routines.

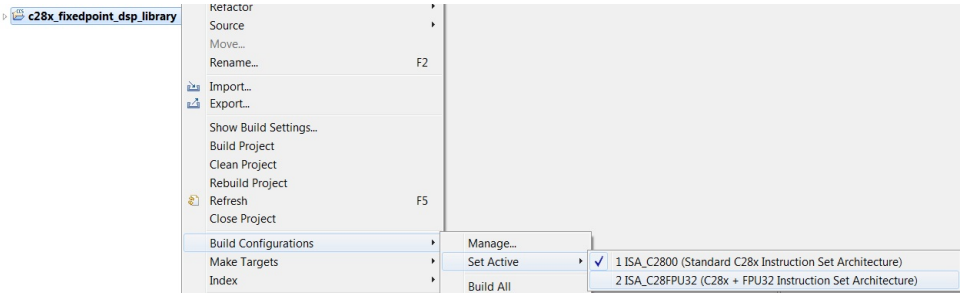


Figure 3.2: Build Configurations for the Fixed Point DSP Library

NOTE:IF THE FPU32 OPTION IS ENABLED AND THE USER ATTEMPTS TO USE THE STANDARD LIBRARY BUILD CONFIGURATION IT WILL RESULT IN A COMPILER ERROR ABOUT MISMATCHING INSTRUCTION SET ARCHITECTURES

The **ISA_C2800** build configuration was built with the following options:

```
-v28 -mt -ml -g --diag_warning=225
```

while the **ISA_C28FPU32** build configuration was built with these options:

```
-v28 -mt -ml -g --diag_warning=225 --fpu_support=fpu32
```

4 Using the Fixed Point Library

The source code and project for the Fixed Point library are provided. The user may import the project into CCSv5 (or later) and be able to view and modify the source code for all routines and lookup tables (see Fig. 4.1)

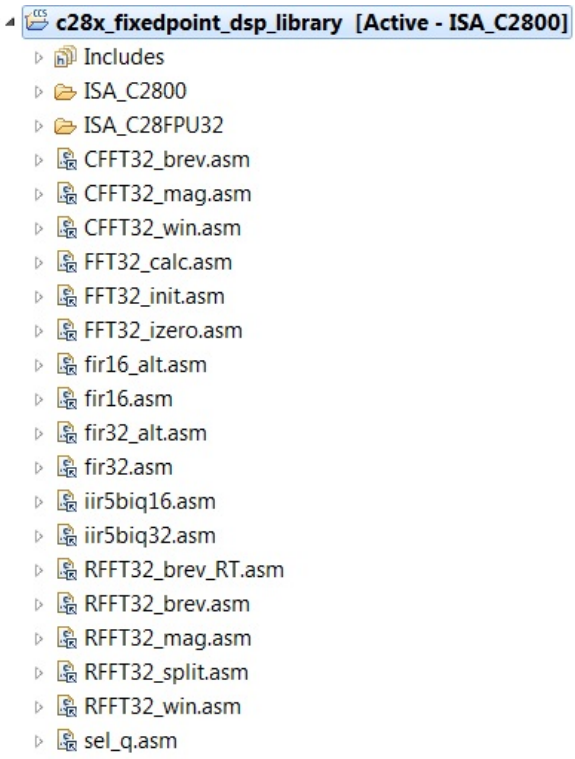


Figure 4.1: Fixed Point Library Project View

4.1 Library Build Configurations

The current version of the library has two build configurations (Fig. 4.2): **ISA_C2800**, henceforth referred to as the standard build or standard library, and **ISA_C28FPU32**, the alternate build or alternate library. The **ISA_C28FPU32** configuration is built with the **–float_support=fpu32** run-time support option set to fpu32, allowing you to integrate this library into a project where hardware floating point support is enabled. This does not mean that the library routines use floating point numbers, rather, it gives the user the option to use Fixed point routines alongside floating point routines. Running a build on the **ISA_C2800** configuration will generate the **c28x_fixedpoint_dsp_library.lib** while the **ISA_C28FPU32** configuration will generate the **c28x_fixedpoint_dsp_library_fpu32.lib**; both libraries (.lib) will be output to the “lib” folder.

NOTE: ATTEMPTING TO LINK IN THE _fpu32 LIBRARY INTO A PROJECT THAT DOES NOT HAVE THE FLOAT_SUPPORT ENABLED WILL RESULT IN A COMPILER ERROR ABOUT MISMATCHING INSTRUCTION SET ARCHITECTURES. THE SAME IS TRUE WHEN LINKING THE STANDARD BUILD INTO A PROJECT WITH FLOATING POINT SUPPORT TURNED ON.

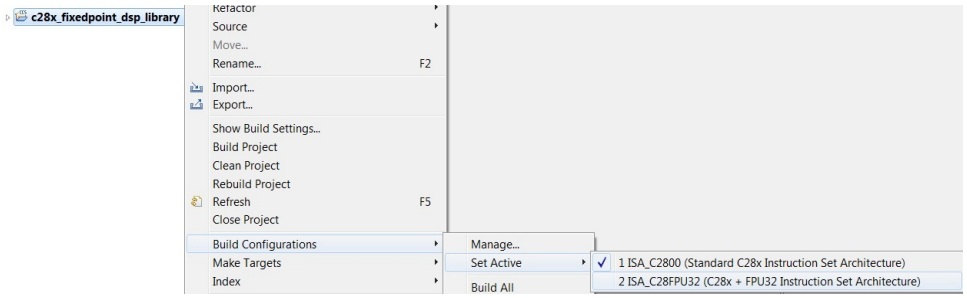


Figure 4.2: Library Build Configurations

4.2 Integrating the Library into your Project

To begin integrating the library into your project follow these easy steps:

1. Go to the **Project Properties->Build->Variables(Tab)** and add a new variable (see Fig. 4.3), INSTALLROOT_TO_FIXEDPTLIB, and point it to the root directory of the Fixed Point library in C2000Ware.

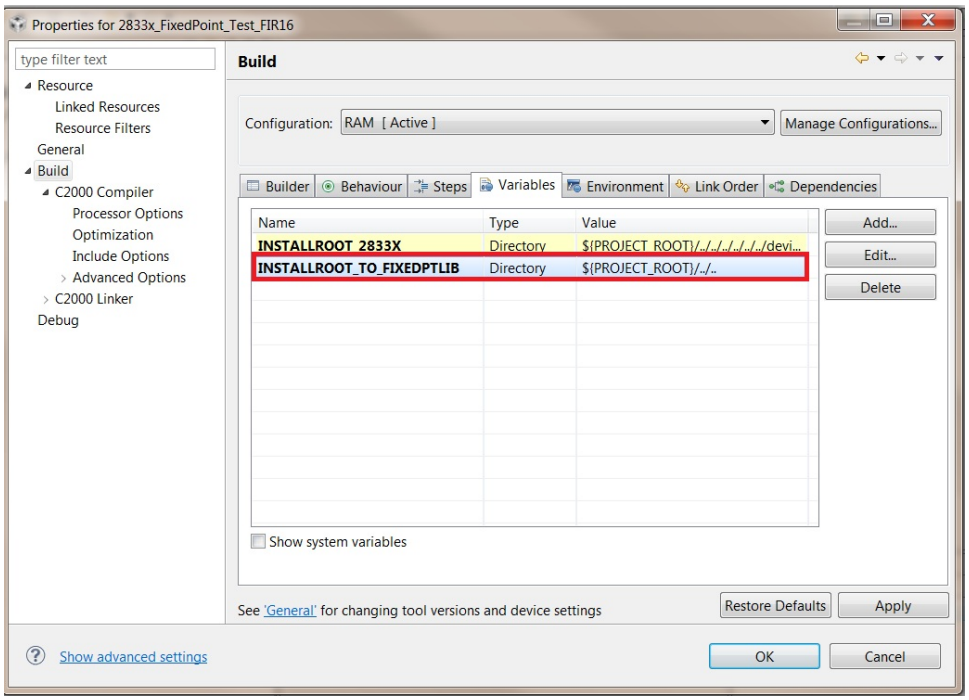


Figure 4.3: Creating a new build variable

Add the new path, **INSTALLROOT_TO_FIXEDPTLIB/include**, to the *Include Options* section of the project properties (Fig. 4.4). This option tells the compiler where to find the library header files.

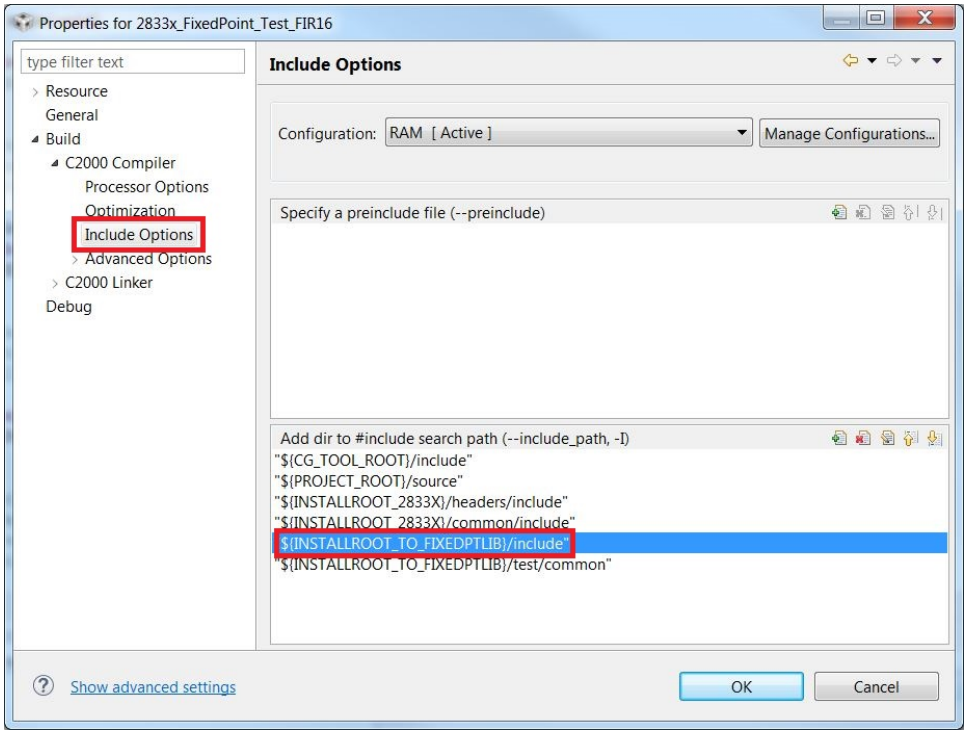


Figure 4.4: Adding the Library Header Path to the Include Options

2. When using the standard library be sure the **–float_support** option, in the **Runtime Model Options**, is disabled. If you intend to use the fixed point library in a project with the **–float_support** option set to fpu32 (Fig. 4.5), then include the alternate library in your project instead.

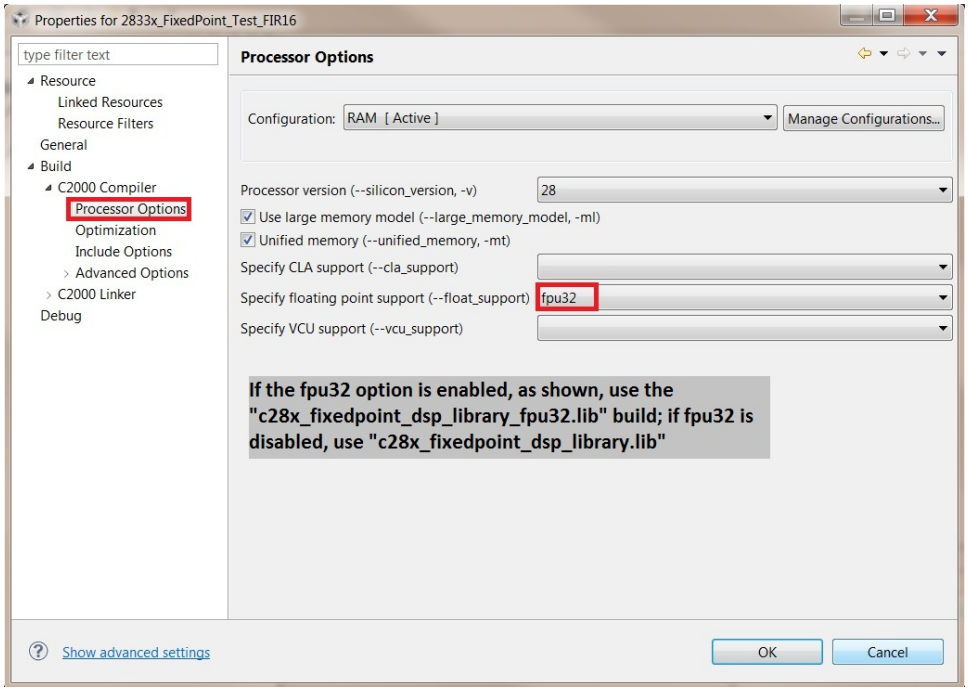


Figure 4.5: Runtime Support Options

3. Add the name of the library and its location to the **File Search Path** as shown in Fig. 4.6.
   NOTE: BE SURE TO USE THE APPROPRIATE LIBRARY, STANDARD OR ALTERNATE, DEPENDING ON THE FLOATING POINT SUPPORT OPTION

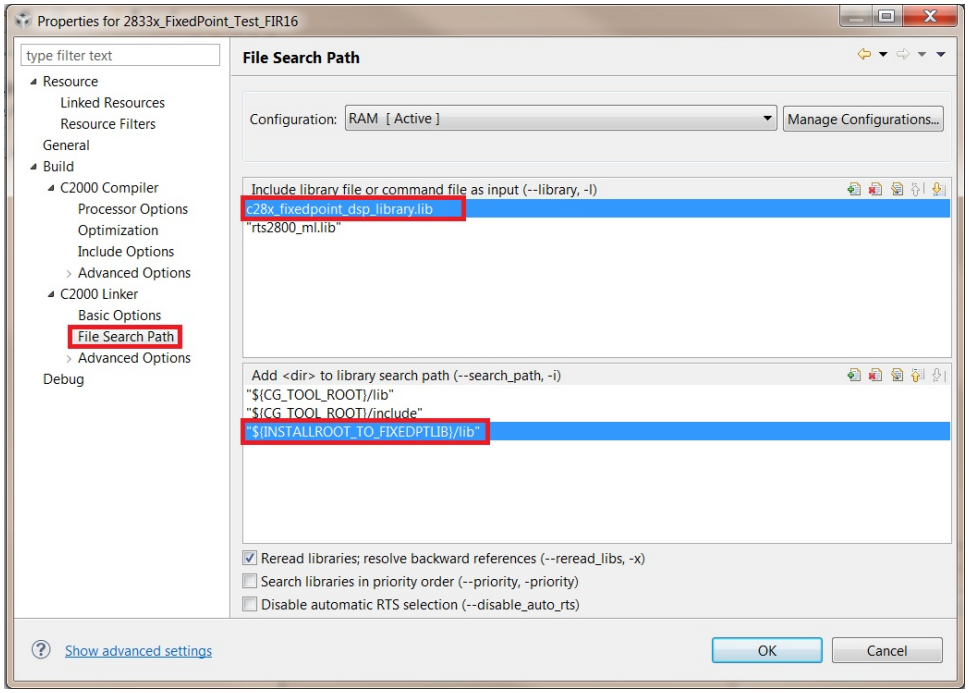


Figure 4.6: Adding the library and location to the file search path

4.3 Choosing a Q representation for the FFT routines

The operation and execution time of the FFT routine, FFT32_calc(), is dependent on the Q format selected in the file **"sel_q.asm"**. The macro **TF_QFMAT** can be set to either Q30 or Q31; the choice of the Q format decides which twiddle factor table will be used in the calculations.

When changing the value of this macro, TF_QFMAT, the user must rebuild the library for the changes to take effect.

5 Application Programming Interface (Fixed Point Library)

5.1 Fixed Point DSP Module Summary

This release contains the FFT and Filter modules. Other modules may be added in future releases. The functions under the FFT, FIR and IIR categories are member functions of the structure defined for each module and should be invoked through the structure object as opposed to direct function calls. The exception are the bit reversal functions which can only be called directly.

Default macros are provided to aid the user in initializing the module object and assuring the right values are written to each of the object's elements. This is important, especially in the case of the Complex and Real FFT modules. They both use the complex FFT function, FFT32_calc(), with the difference that an N point real FFT is done by running an N/2 point complex FFT followed by a split operation. The user must, therefore, instantiate an RFFT object with the correct size of the FFT, twiddle factor skip ratio and the number of stages. The use of the initialization macros makes the task straightforward and error free. The following table lists the functions and their prototypes:

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 *);
CFFT32_mag	void CFFT32_mag(void *);
CFFT32_win	void CFFT32_win(void *);
RFFT32_split	void RFFT32_split(void *);
RFFT32_mag	void RFFT32_mag(void *);
RFFT32_win	void RFFT32_win(void *);
FIR Modules	
FIR16_init	void FIR16_init(void *);
FIR16_calc	void FIR16_calc(void *);
FIR16_Alt_init	void FIR16_Alt_init(void *);
FIR16_Alt_calc	void FIR16_Alt_calc(void *);
FIR32_init	void FIR32_init(void *);
FIR32_calc	void FIR32_calc(void *);
FIR32_Alt_init	void FIR32_Alt_init(void *);
FIR32_Alt_calc	void FIR32_Alt_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 *);

Table 5.1: Summary of Library Routines

5.2 Module Description

5.2.1 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.

Prototype:

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

Parameters:

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 the size of the complex FFT (a power of 2).

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

Header File:

fft.h

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, the complex-valued DFT algorithm can still be used. One simple approach is to create a complex sequence from the real sequence, that is, real data for the real components and zeros for the imaginary component, the complex FFT can then be applied directly. Moreover, the 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 are the same, then it performs

an in-place bit reversal.

In order to store the N-point real valued sequence (32-bit) in complex form, we need 2N long words. **The destination buffer must be aligned to a 4N word boundary (16-bit word length) or 2N long words, where N is the number of acquired samples (should be a power of 2).**

Figure 5.1 demonstrates the bit-reversal process for 8 real data samples. The real data samples Re(n) occupy the even locations of the source array (the real part) while the odd locations (the imaginary part), Im(n), are left untouched. In this particular example, the storage buffer must be aligned to 16 longs words in order to bit reverse the 8 real data samples (32-bit) in complex form.

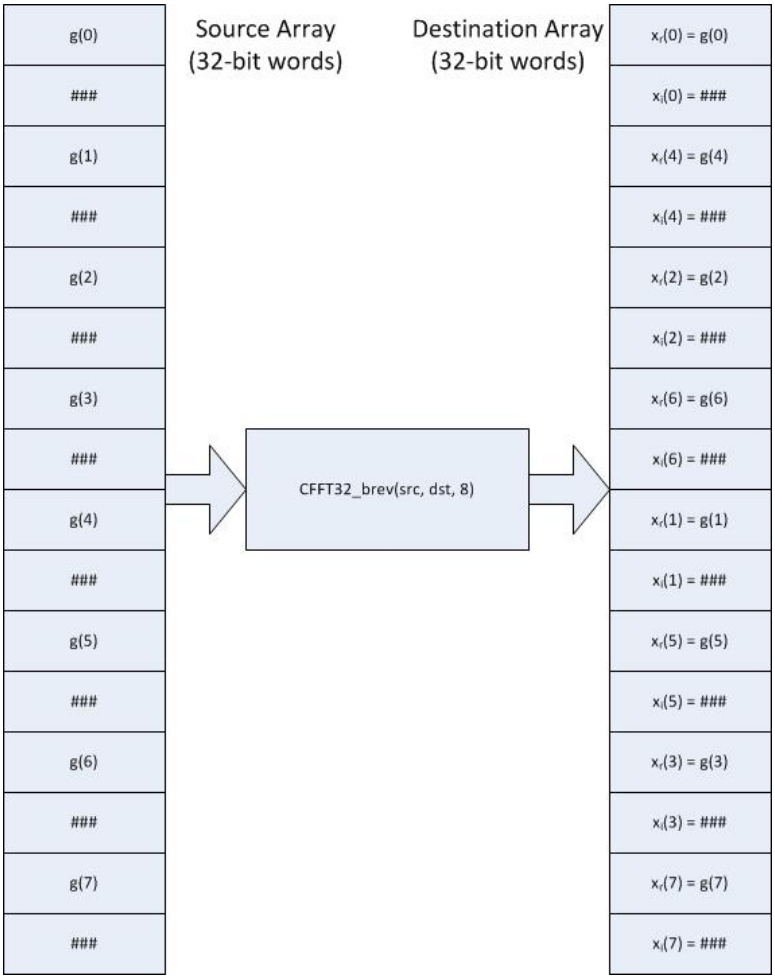


Figure 5.1: Bit Reversal Process

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:

All buffers and stack are placed in internal memory (zero-wait states in data space). Note that the CFFT32_brev must be called twice consecutively to first bit-reverse order the real part of the input buffer followed by the imaginary part.

Size	C-Callable ASM [1]
32	282 cycles
64	538 cycles
128	1050 cycles
256	2074 cycles
512	4122 cycles
1024	8218 cycles

[1] Execution cycles for a single invocation of the function

5.2.2 Real FFT Bit Reverse Function

Description:

This function reads N-point in-order real data samples (32-bit) stored in contiguous locations and writes them in bit-reversed order. It supports both in-place and off-place bit reversing.

Prototype:

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

Parameters:

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 2N word boundary (16-bit word length)** or N long words, where N is size of the real FFT (a power of 2).

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

Header File:

fft.h

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 RFFT32_brev function:

Item	Description	Format	Q-Values	Comment
src	Input buffer	Pointer to 32-bit integer array	Q31~Q0	Input data is stored in contiguous memory locations, i.e., there are no zeros between two data points.
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 power of 2

Background Information:

In many real 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 the imaginary components, The complex FFT can then be applied directly. However, this method is not efficient as it consumes *2N* memory locations (Real Imaginary) for *N* point sequence.

When input is purely real, their symmetric properties compute DFT very efficiently. One such optimized real FFT algorithm for *N*-point real data sequence is packing algorithm. The original *N*-point sequence is packed as $\frac{N}{2}$-point complex sequence and $\frac{N}{2}$-point complex FFT is performed on the complex sequence. Finally, the resulting $\frac{N}{2}$-point complex output is unpacked into another $\frac{N}{2}+1$ point complex sequence, which corresponds to spectral bin $[0, \frac{N}{2}]$ of *N*-point real input sequence. Spectral bin 0 to $\frac{N}{2}$ is sufficient, as the remaining bins $\frac{N}{2}+1$ to *N-1* are complex conjugates of spectral bins $\frac{N}{2}-1$ to 1. Notice that the bin 0 and $\frac{N}{2}$ do not have a matching point. Real part of bin 0 corresponds to DC offset which is average of all the time domain samples and the imaginary part will always be zero. The real part of bin

number $\frac{N}{2}$ corresponds to nyquist frequency and the imaginary part will always be zero.

The real FFT requires *N+2* memory locations to compute the FFT for *N*-point real valued sequence, which is highly preferable in contrast to the complex FFT that consumes *2N* locations for *N*-point real valued sequence.

The figure below demonstrates the concept of packing and how the bit reversal process works.

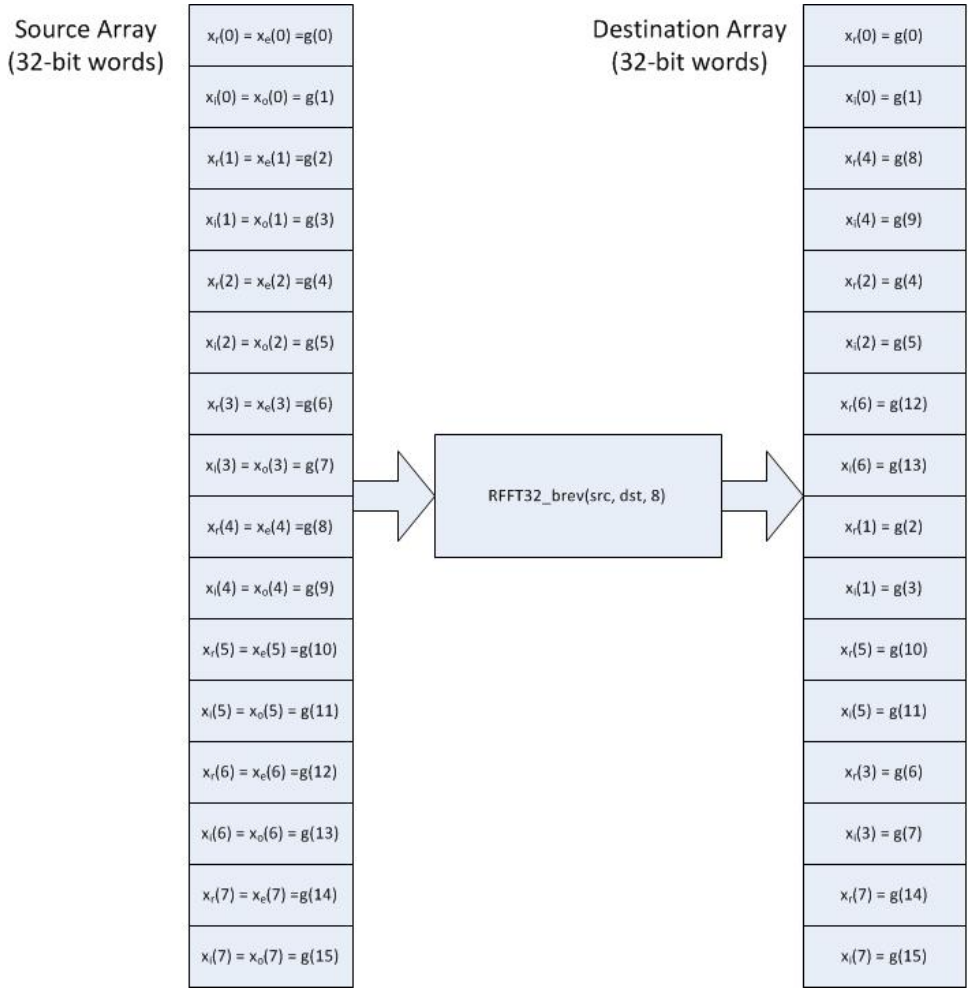


Figure 5.2: Bit Reversal Process

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:

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

Size	C-Callable ASM
32	169 cycles
64	313 cycles
128	601 cycles
256	1177 cycles
512	2329 cycles
1024	4633 cycles
2048	9241 cycles

5.2.3 Real Input Point by Point Bit Reverse Function

Description:

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

Header File:

fft.h

Availability:

C-Callable Assembly (CcA)

Object Definition:

The structure of the RFFT32_ACQ object is as follows

```
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 $\frac{N}{2}$ point complex data sequence in bit reversed order. The even data samples occupy the real part and odd data samples occupy imaginary part. Real FFTacquisition 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.

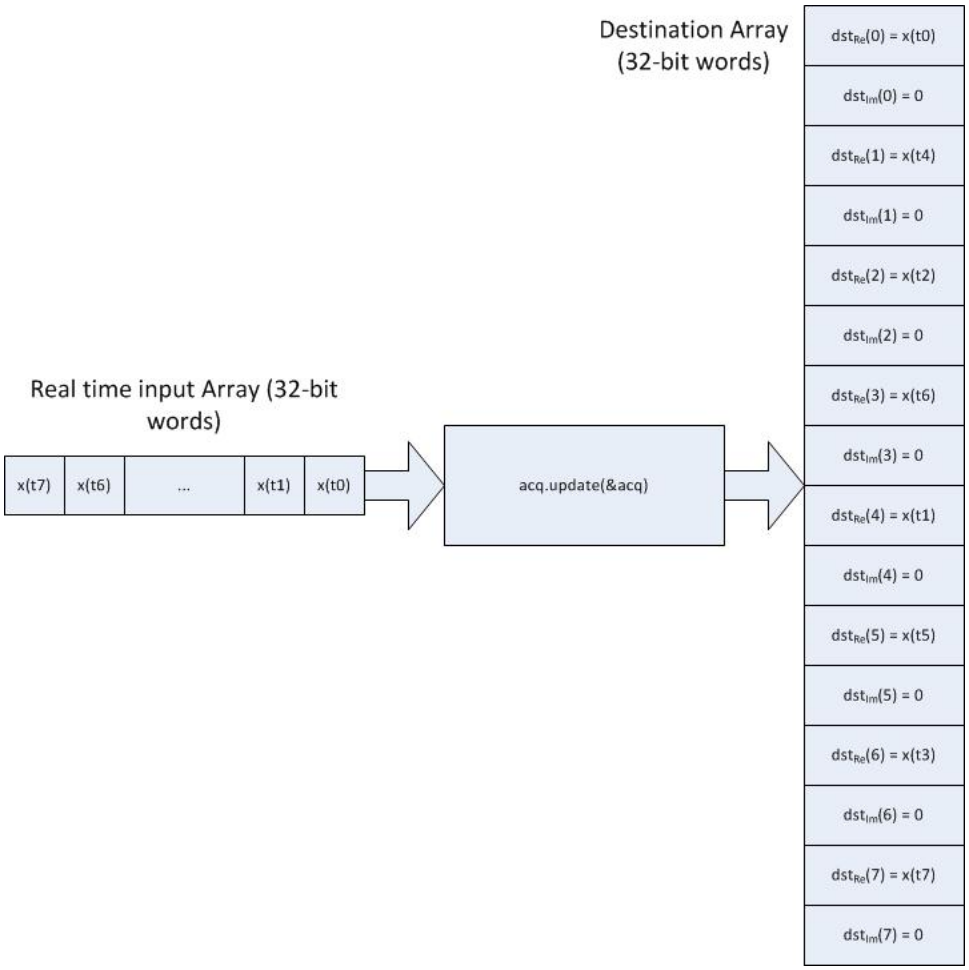


Figure 5.3: Bit Reversal Process

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:

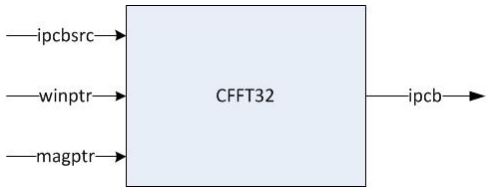
All buffers and stack are placed in internal memory (zero-wait states in data space).

Size	C-Callable ASM
1 Block	29 cycles

5.2.4 32 Bit Complex Fast Fourier Transform Module

Description:

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



Header File:

fft.h

Availability:

C-Callable Assembly (CcA)

Object Definition:

The structure of the CFFT32 object is as follows

```
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 2N word (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	$nrstages = log_2(size)$
magptr	Magnitude output	Pointer to 32-bit integer array	Q30	Magnitude buffer pointer

Continued on next page

Table 5.2 – continued from previous page

Item	Description	Format	Q-values	Comment
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=8, 16, 32, 64, 128, 256, 512, 1024 2048 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.

TF_QFMAT The Q-format for the twiddle factors is determined by the constant TF_QFMAT which can be found in the source file "sel_q.asm". The user has the option to set the constant to either **Q30** or **Q31**. By default, the library was built with the constant set to Q30. When altering the value of this constant, the user must rebuild the library for the change to take effect. The FFT32_init function, on the basis of the TF_QFMAT value, will set the object twiddle factor pointer to either the Q31 table (TF_QFMAT = Q31) or the Q30 table (TF_QFMAT = Q30).

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 3072 entries (32-bit words) to facilitate complex FFT computation of up to 4096 points. The table has $\frac{3N}{4}$ entries, where N is the maximum supported FFT. The

entries are as follows:

$$
\begin{aligned}
sin(2 * \pi * 0/4096) \\
sin(2 * \pi * 1/4096) \\
... \\
sin(2 * \pi * 1024/4096)\ or\ cos(2 * \pi * 0/4096) \\
sin(2 * \pi * 1025/4096)\ or\ cos(2 * \pi * 1/4096) \\
... \\
sin(2 * \pi * 2047/4096)\ or\ cos(2 * \pi * 1023/4096) \\
sin(2 * \pi * 2048/4096)\ or\ cos(2 * \pi * 1024/4096) \\
... \\
sin(2 * \pi * 3071/4096)\ or\ cos(2 * \pi * 2047/4096)
\end{aligned}
$$

Therefore, the total space (in words) that needs to be allocated for the twiddle factor table is 6144 or 0x1800.

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 $\frac{1}{N}$ 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 applies a window, pointed to by the *winptr* element of CFFT32 object, to the bit reversed data sequence (in Q31 format) in the computation buffer. The aim is to reduce the spectral leakage that occurs when the sampling rate is not an integer multiple of the constituent frequencies of the input waveform. The window is only applied to the real part of this buffer; the assumption being that the user acquires N **real** data points from the ADC, stores them at alternate locations (real part) of the input buffer and then zeros out the imaginary part, before running the complex FFT. If, however, the user is sampling complex data, i.e. in-phase and quadrature phase data, and wishes to window both the real and imaginary parts, they can use the CFFT32_win_dual() instead - prior to calling the window function, set the **"win"** function pointer of the CFFT32 object to point to the CFFT32_win_dual().

```
cfft.win = CFFT32_win_dual;
cfft.win(&cfft);
```

The size of the window coefficient array is $\frac{N}{2}$ long words (1/2 of the FFT length). Note that the windowing function should be invoked once the input data has been reordered in bit-reversed format. The window coefficients, for 32 to 4096 point FFTs, are provided for the following windows:

1. barthannwin
2. bartlett
3. blackman
4. blackmanharris
5. bohmanwin
6. chebwin

7. flattopwin
8. gausswin
9. hamming
10. hann
11. kaiser
12. nuttallwin
13. parzenwin
14. rectwin
15. taylorwin
16. triang
17. tukeywin

Each window has its own header file in the include folder, of the format: *fft_<window>_Q31.h*. The MATLAB script, "C28xFixedPointLib_Window_Generator.m", used to generate these files is included under examples_ccsv5\2833x_FixedPoint_Win\matlab; the script generates the windows using their default arguments, the user may choose to modify the script to generate specific windows with non-default arguments.

A fairly simple MATLAB script to generate a single window of a particular size can be accomplished with the following code snippet:

```
fid = fopen('output.txt','W');          % Open the output file
%*************************************************************************
% Hamming 32 pt.                                                         *
%*************************************************************************
N=32;                                   % Window length
string='Hamming32';                     % Header string
%-------------------
x=hamming(N);                           % Create the window
x=x(1:N/2);                             % Only need 1st half of data
xQ31=round(x*(2^31));                   % Round and put in Q31 format
fprintf(fid,'%s\n',string);             % Write header information
fprintf(fid,'%u,',xQ31);                % Write the output to the file
fprintf(fid,'\n\n');                    % Insert a couple of linefeeds
%*************************************************************************
fclose(fid);                            % Close the output file
```

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).

$$\begin{aligned} X(k) &= X_r(k) + jX_i(k) \\ |X(k)|^2 &= X_r(k)^2 + X_i(k)^2 \end{aligned}$$

Alignment Requirements:

The computation buffer should be aligned to 4N words (16-bit words size) boundary or 2N 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 512 words for 128 point FFT)

```
    /* computation buffer */
    FFTipcb     ALIGN(512) : { }  > RAML4,    PAGE 1
    FFTipcbsrc             : { }  > 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 *))CFFT32_mag,
             (void (*)(void *))CFFT32_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:

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

Samples (N-pt FFT)	Execution Time Q31 (cycles)	Execution Time Q30 (cycles
32-bit Complex FFT (FFT32_calc())		
32	1930	1994
64	4703	4895
128	11156	11668
256	25897	27177
512	59070	62142
1024	132819	139987
32-bit Complex FFT (CFFT32_mag())		
32	743	
64	1441	
128	2961	
256	4274	
512	11709	
1024	21728	
32-bit Complex FFT (CFFT32_win())		
32	306	
64	594	
128	1170	
256	2322	
512	4626	
1024	9234	
32-bit Complex FFT (CFFT32_dual_win())		
32	495	
64	971	
128	1931	
256	3855	
512	7695	
1024	15375	

Table 5.3: Complex FFT routines

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

The section “FFTtf” holds 3072 twiddle factors, each 32-bits or 2 words wide. A total of 6144 (0x1800) 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 (at least 0x1800) 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_SIZE(_FFTtfLoadSize),
        RUN_START(_FFTtfRunStart)
```

In the main C file, declare the following variables

```
extern uint16_t FFTtfLoadStart, FFTtfLoadSize, FFTtfRunStart;
```

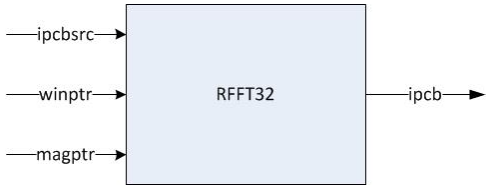
Finally use the memcpy function to copy over the section from FLASH to RAM

```
memcpy((uint32_t *)&FFTtfRunStart, (uint32_t *)&FFTtfLoadStart,
         (uint32_t)&FFTtfLoadSize);
```

5.2.5 32 Bit Real Fast Fourier Transform Module

Description:

This real FFT module computes the FFT of N -point real sequence using $\frac{N}{2}$ -point complex FFT routine.



Header File:

fft.h

Availability:

C-Callable Assembly (CcA)

Object Definition:

The structure of the RFFT32 object is as follows

```
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 (*split)(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 2N word (32-bit word size) boundary for N point real 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	$nrstages = log_2(size)$
magptr	Magnitude output	Pointer to 32-bit integer array	Q30	Magnitude buffer pointer

Continued on next page

Table 5.4 – continued from previous page

Item	Description	Format	Q-values	Comment
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
split	Member function	Function pointer	N/A	Split function computation
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= 32, 64, 128, 256, 512, 1024, 2048 4096 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 split(RFFT32_Handle);
void mag(RFFT32_Handle); (OPTIONAL)
void win(RFFT32_Handle);  (OPTIONAL)
```

void init(RFFT32_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 3072 entries (32-bit words) to facilitate complex FFT computation of up to 4096 points. The table has $\frac{3N}{4}$ entries, where N is the maximum supported FFT. The

entries are as follows:

$$
\begin{aligned}
sin(2 * \pi * 0/4096) \\
sin(2 * \pi * 1/4096) \\
... \\
sin(2 * \pi * 1024/4096)\ or\ cos(2 * \pi * 0/4096) \\
sin(2 * \pi * 1025/4096)\ or\ cos(2 * \pi * 1/4096) \\
... \\
sin(2 * \pi * 2047/4096)\ or\ cos(2 * \pi * 1023/4096) \\
sin(2 * \pi * 2048/4096)\ or\ cos(2 * \pi * 1024/4096) \\
... \\
sin(2 * \pi * 3071/4096)\ or\ cos(2 * \pi * 2047/4096)
\end{aligned}
$$

Therefore, the total space (in words) that needs to be allocated for the twiddle factor table is 6144 or 0x1800.

void calc(RFFT32_Handle); This routine performs radix 2, $\frac{N}{2}$ point in-place complex 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 $\frac{1}{N}$ scaling of FFT is distributed across the stages. Note that the input and output data are in Q31 format.

void split(RFFT32_Handle); Split function obtains the first $\frac{N}{2} + 1$ complex spectral bins of the *N*-point real valued input sequence from the output of $\frac{N}{2}$-point complex FFT. Hence, the size of the computation buffer is *N + 2* long words, to store the $\frac{N}{2} + 1$ spectral bins in complex form.

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 $\frac{N}{2}$ 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. The size of the window coefficient array is $\frac{N}{2}$ long words (1/2 of the FFT length). Note that the windowing function should be invoked once the input data has been reordered in bit-reversed format. The window coefficients, for 32 to 4096 point FFTs, are provided for the following windows:

1. barthannwin
2. bartlett
3. blackman
4. blackmanharris
5. bohmanwin
6. chebwin
7. flattopwin
8. gausswin
9. hamming
10. hann
11. kaiser

12. nuttallwin
13. parzenwin
14. rectwin
15. taylorwin
16. triang
17. tukeywin

Each window has its own header file in the include folder, of the format: *fft_<window>_Q31.h*. The MATLAB script, "C28xFixedPointLib_Window_Generator.m", used to generate these files is included under examples_ccsv5\2833x_FixedPoint_Win\matlab; the script generates the windows using their default arguments, the user may choose to modify the script to generate specific windows with non-default arguments.

A fairly simple MATLAB script to generate a single window of a particular size can be accomplished with the following code snippet:

```
fid = fopen('output.txt','W');          % Open the output file
%*********************************************************************
% Hamming 32 pt.                                                     *
%*********************************************************************
N=32;                                   % Window length
string='Hamming32';                     % Header string
%-------------------
x=hamming(N);                           % Create the window
x=x(1:N/2);                             % Only need 1st half of data
xQ31=round(x*(2^31));                   % Round and put in Q31 format
fprintf(fid,'%s\n',string);             % Write header information
fprintf(fid,'%u,',xQ31);                % Write the output to the file
fprintf(fid,'\n\n');                    % Insert a couple of linefeeds
%*********************************************************************
fclose(fid);                            % Close the output file
```

void mag(RFFT32_Handle); (OPTIONAL) This routine obtains the Magnitude Square of $\frac{N}{2}+1$ 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 $\frac{N}{2}+1$ magnitude outputs is equal to $\frac{N}{2}+1$ long words (1/2 of the real FFT length + 1).

$$\begin{aligned} G(k) &= G_r(k) + jG_i(k) \\ |G(k)|^2 &= G_r(k)^2 + G_i(k)^2 \end{aligned}$$

Alignment Requirements:

The computation buffer should be aligned to 2N words (16-bit words size) boundary or 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 256 words for 128 point FFT)

```
/* computation buffer */
FFTipcb     ALIGN(256) : { }  > RAML4,    PAGE 1
FFTipcbsrc             : { }  > 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. Note the values for the object elements; a $\frac{N}{2}$ complex FFT is used to do an *N* point real FFT and the object elements must be initialized appropriately

```
#define RFFT32_32P_DEFAULTS {
             (int32 *)NULL,
             (int32 *)NULL,
             16,
             4,
             (int32 *)NULL,
             (int32 *)NULL,
             0,
             0,
             256,
             (void (*)(void *))FFT32_init,
             (void (*)(void *))FFT32_calc,
             (void (*)(void *))RFFT32_split,
             (void (*)(void *))RFFT32_mag,
             (void (*)(void *))RFFT32_win
           }
```

The following sample code obtains the real FFT value.

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

  /* Create an Instance of FFT module */
  RFFT32 fft=RFFT32_32P_DEFAULTS;
  main()
  {
    .........
    // Generate sample waveforms:
    for(i=0; i < (N); i++)
    {
      ipcbsrc[i] =(long)real[i] //Q31
    }
    .........
    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 */
    fft.split(&fft);  /* perform the split operation to get the N/2 + 1 spectrum
  }
```

Benchmark Information:

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

Samples (N-pt FFT)	Execution Time Q31 (cycles)	Execution Time Q30 (cycles
32-bit Real FFT (FFT32_calc())		
32	765	781
64	1930	1994
128	4703	4895
256	11156	11668
512	25897	27177
1024	59070	62142
2048	132819	139987
32-bit Real FFT (RFFT32_split())		
32	407	
64	799	
128	1583	
256	3151	
512	6287	
1024	12559	
2048	25103	
32-bit Real FFT (RFFT32_mag())		
32	319	
64	582	
128	1108	
256	2160	
512	4264	
1024	8472	
2048	16888	
32-bit Real FFT (RFFT32_win())		
32	258	
64	498	
128	978	
256	1938	
512	3858	
1024	7698	
2048	15378	

Table 5.5: Real FFT routines

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 3072 twiddle factors, each 32-bits or 2 words wide. A total of 6144 (0x1800) 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 (at least 0x1800) 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_t FFTtfLoadStart, FFTtfLoadSize, FFTtfRunStart;
```

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

```
memcpy((uint32_t *)&FFTtfRunStart, (uint32_t *)&FFTtfLoadStart,
         (uint32_t)&FFTtfLoadSize);
```

5.2.6 16 Bit FIR Filter Module

Description:

This module implements 1 block (point by point) FIR Filter using DMAC instructions that effectively executes 2 filter taps in a cycle. There are two functions in this module, **FIR16_calc** can support up to 255^{th} order FIR filter, while **FIR16_Alt_calc** can support up to a 65535^{th} order filter.

Header File:

filter.h

Availability:

C-Callable Assembly (CcA)

Object Definition:

The structure of the FIR16 object is as follows

```
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); it serves as the wraparound point for the delay buffer whereas **FIR16_Alt_init** will zero it out.

Continued on next page

Table 5.6 – continued from previous page

Item	Description	Format	Q-values	Comment
order	Order of the filter	int16	Q0	Filter order, one less than the number of taps. Note that if the order is odd, **FIR16_Alt_init** will increase the order by 1 to make it even; this is done to facilitate the computation routine and is not an increase in the true order of the filter.
input	Input to the filter	int16	Q15	N/A
output	Output of the filter	int16	Q15	N/A
init	Member function	Function pointer	N/A	This initialization function initializes cbindex and zeros out the 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.

FIR16_ALT_DEFAULTS Structure symbolic constant is to initialize FIR16 Module in order to run the alternate 16-bit FIR routine. This provides the initial values to the terminal variables as well as method pointers.

Methods:

```
void FIR16_init(FIR16_Handle);
void FIR16_calc(FIR16_Handle);
void FIR16_Alt_init(FIR16_Handle);
void FIR16_Alt_calc(FIR16_Handle);
```

Each FIR implementation has two functions, an initialization and computation function. The input argument to these functions is the module handle.

Note:

1. **The delay buffer is assigned to a section, "firldb", which should be aligned to 256 word boundary in the data memory (RAM) for the FIR16 module. The FIR16_Alt module does not require any alignment**
2. **The coefficients are placed in a section, "firfilt", that 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.

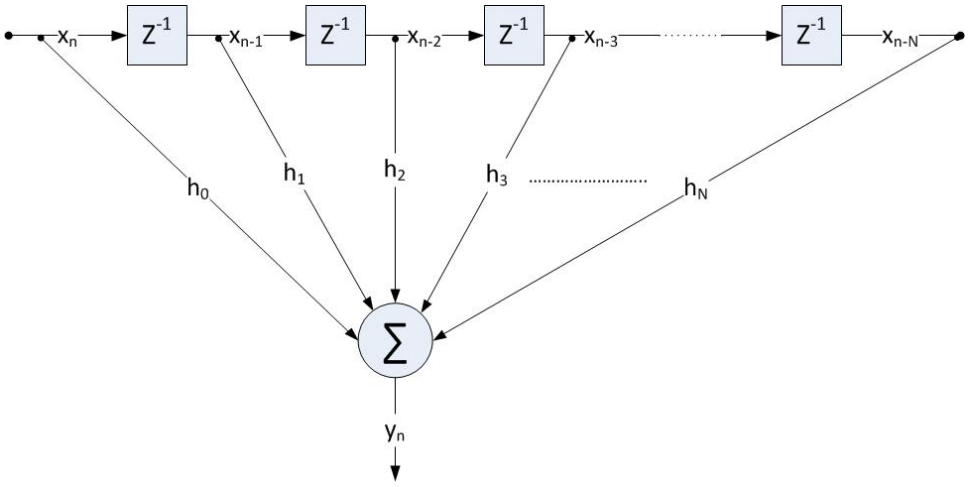


Figure 5.4: FIR Filter

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 the FIR16 module uses the C28x circular buffer addressing mode which requires the delay buffer to be aligned to a 256 words boundary and therefore, the delay buffer length is restricted to 256 words. Hence, this FIR filter module allows filter implementation of up to the 255^{th} order. The FIR16_Alt module, however, uses the C2xLP circular addressing mode which can support a buffer of up to 65536 words and allows a filter implementation of up to the 65535^{th} order.

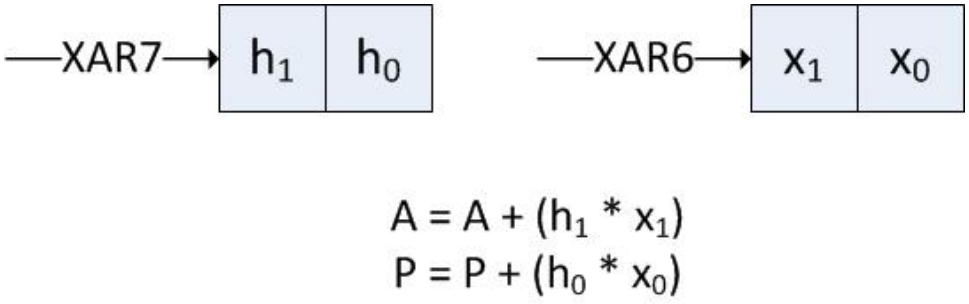


Figure 5.5: **DMAC ACC:P, *XAR6%++, XAR7++**

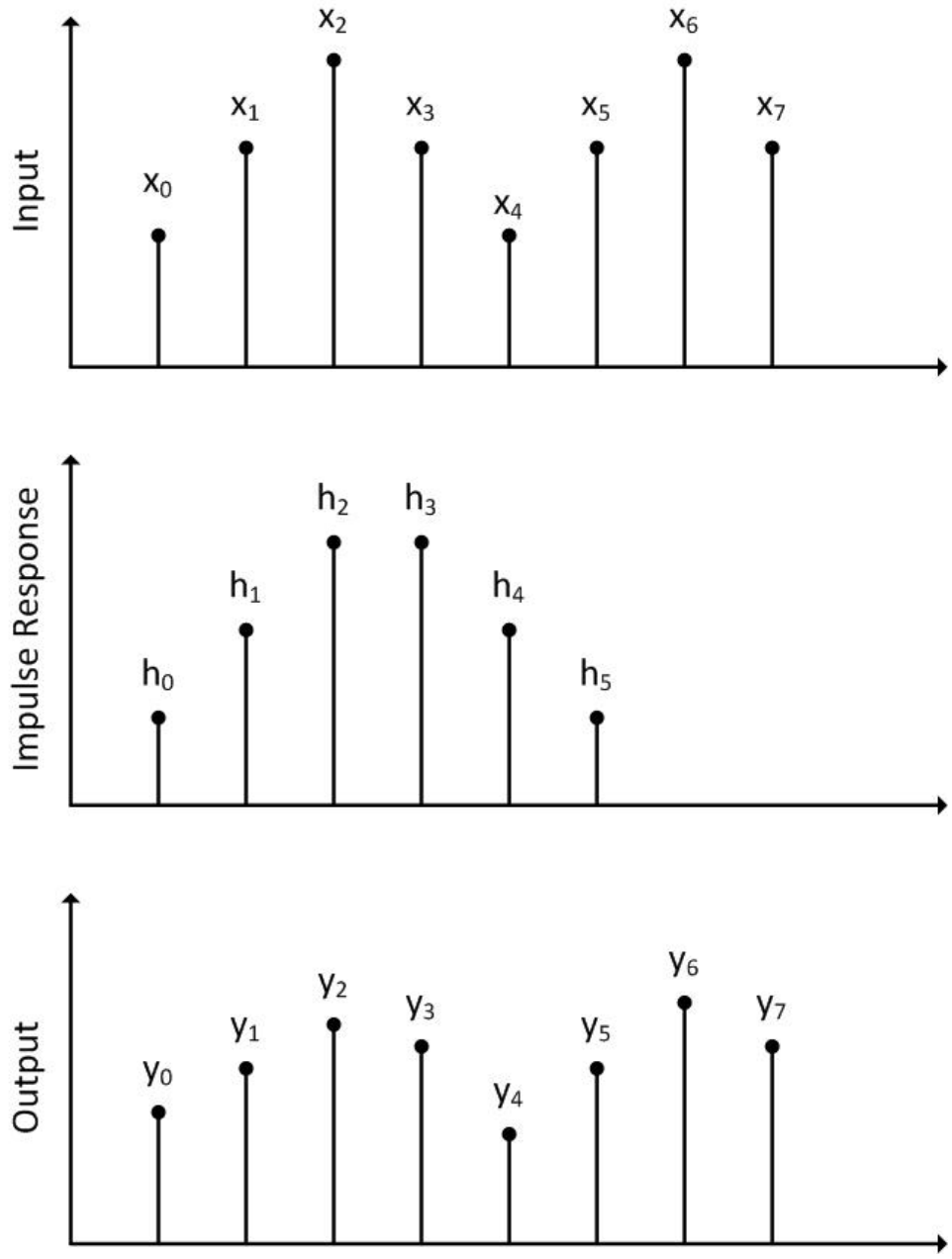


Figure 5.6: **Filter Input/Output**

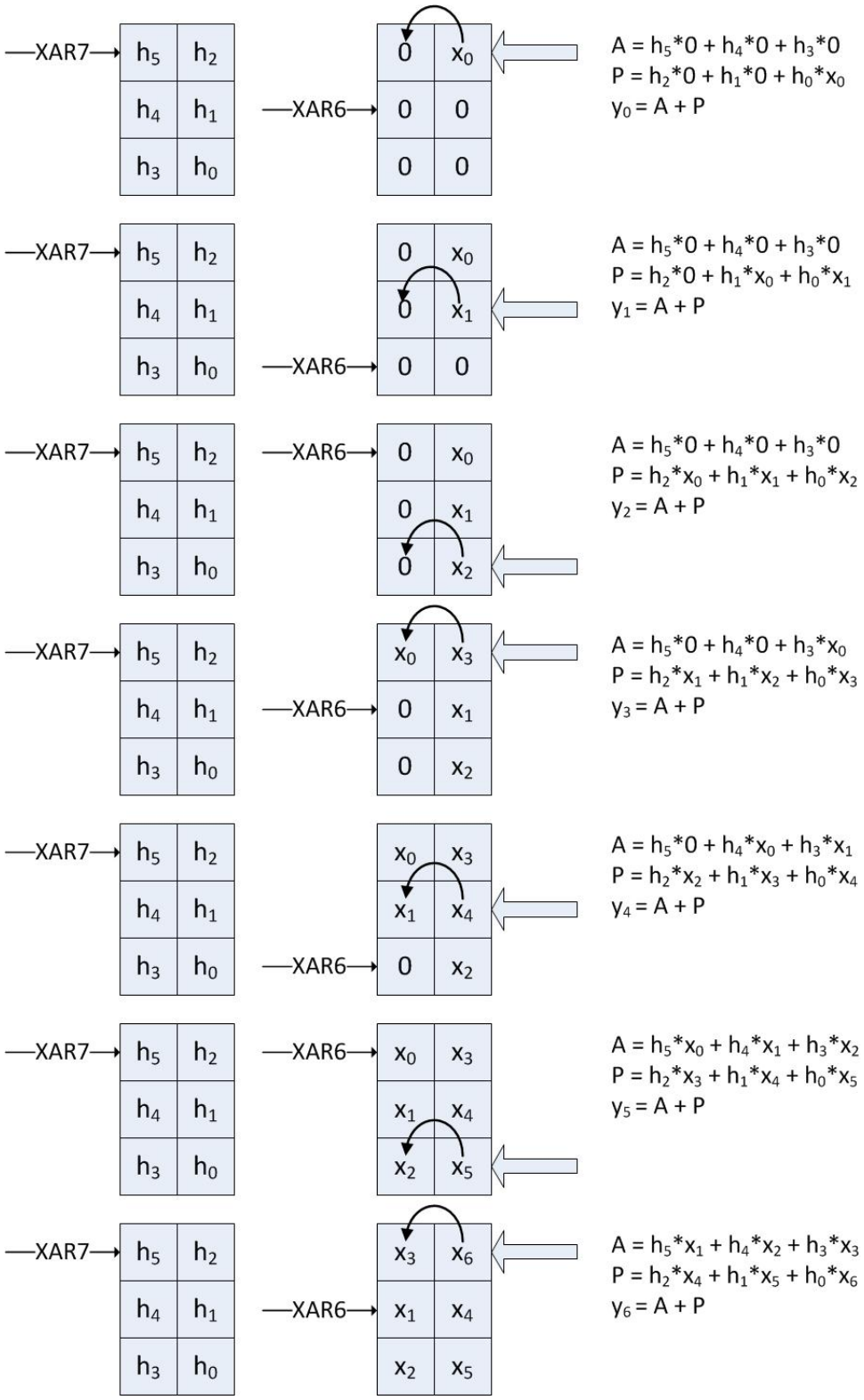


Figure 5.7: **Filtering Process**

Coefficients generation:

The FIR coefficients can be generated by MATLAB Filter Design and Analysis Tool (FDATool) in fixed point format and written to a header file, which the user can then include in their CCS project. The coefficients generated by MATLAB may also be copied directly into **"fir.h"** replacing the contents of FIR16_LPF32_TEST allowing the user to easily test their filter with the provided CCS example.

MATLAB will generate FIR_ORDER+1 coefficients as signed 16-bit integers with fractional 15 bits i.e., Q15 format. If the order is even, the size of the delay line buffer, **dbuffer**, must be (FIR_ORDER + 3)/ 2. If the order is odd, the size should be (FIR_ORDER + 1)/2

```
// Define the Delay buffer for the "FIR_ORDER"th order filter
// and place it in "firldb" section. Since we define it as int32_t,
// The size of the buffer is:
// FIR_ORDER even -> (FIR_ORDER+1)/2+1 e.g. FIR_ORDER = 32, size = 17 dwords
//           odd  -> (FIR_ORDER+1)/2   e.g. FIR_ORDER = 31, size = 16 dwords
// The delay line buffer must be aligned to a 256 word boundary
// if using the FIR16_calc()
#ifndef __cplusplus
#pragma DATA_SECTION(dbuffer,"firldb");
#else
#pragma DATA_SECTION("firfilt");
#endif
int32_t dbuffer[(FIR_ORDER+3)/2];
```

The calculation routines use the DMAC instruction, it does two multiply-accumulates in a single cycle; it requires the coefficients be reordered so that the output of the filter is generated in the right order. The C code to do this is as follows:

```
#if(FIR_ORDER & 0x01)   // odd
#define FIR_ORDER_REV   (FIR_ORDER + 1) // even
#else
#define FIR_ORDER_REV   (FIR_ORDER + 2) // even
#endif

// Define Constant Coefficient Array  and place it in the "coefffilt"
// section. You can either reorder the coefficients at run time (done
// in this example) or just store them reordered.
// Index       LSW       MSW
//         +---------+---------+
//  0      | h(L-1)  | h(L/2-1)|
//  2      | h(L-2)  | h(L/2-2)|
//  4      | h(L-3)  | h(L/2-3)|
//  6      | h(L-4)  | h(L/2-4)|
//  ...    | ...     | ...     |
//  L/2-3  | h(L/2+2)| h(2)    |
//  L/2-2  | h(L/2+1)| h(1)    |
//  L/2-1  | h(L/2)  | h(0)    |
//         +---------+---------+
// The size of the array is:
// FIR_ORDER even -> (FIR_ORDER+1)+1 e.g. FIR_ORDER = 32, size = 34 words
//           odd  -> (FIR_ORDER+1)   e.g. FIR_ORDER = 31, size = 32 words
// The reason being that we use the DMAC operation which will multiply
```

```
// the coefficients in 32-bit chunks, we don't want an odd size coefficient
// buffer and risk corrupting the delay line
#ifndef __cplusplus
#pragma DATA_SECTION(coeff, "coefffilt");
#else
#pragma DATA_SECTION("coefffilt");
#endif
int16_t coeff[FIR_ORDER+2];

#ifndef __cplusplus
#pragma DATA_SECTION(revCoeff, "coefffilt");
#else
#pragma DATA_SECTION("coefffilt");
#endif
int16_t revCoeff[FIR_ORDER+2];

// Reorder the coefficients
// FIR_ORDER even -> (FIR_ORDER+1)+1 e.g. FIR_ORDER = 32, size = 34 words
//           odd  -> (FIR_ORDER+1)   e.g. FIR_ORDER = 31, size = 32 words
for(i = 0; i < FIR_ORDER + 2; i = i + 2){
    revCoeff[FIR_ORDER_REV-i-1] = coeff[i/2+FIR_ORDER_REV/2];
    revCoeff[FIR_ORDER_REV-i-2] = coeff[i/2];
}
```

1. 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 (5.8).

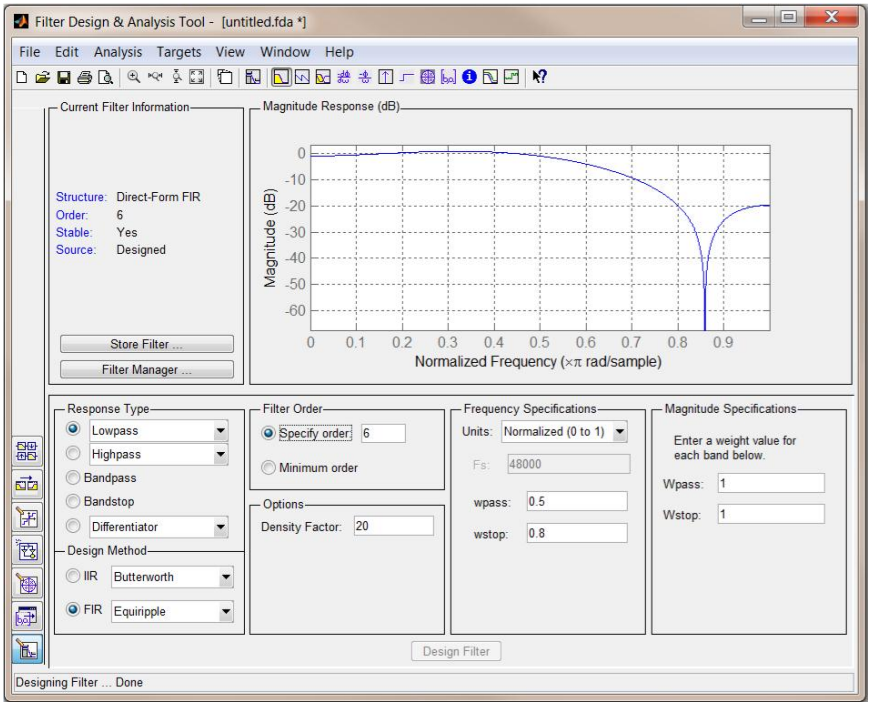


Figure 5.8: **1)**

2. Go to Targets-> Generate C header (5.9).

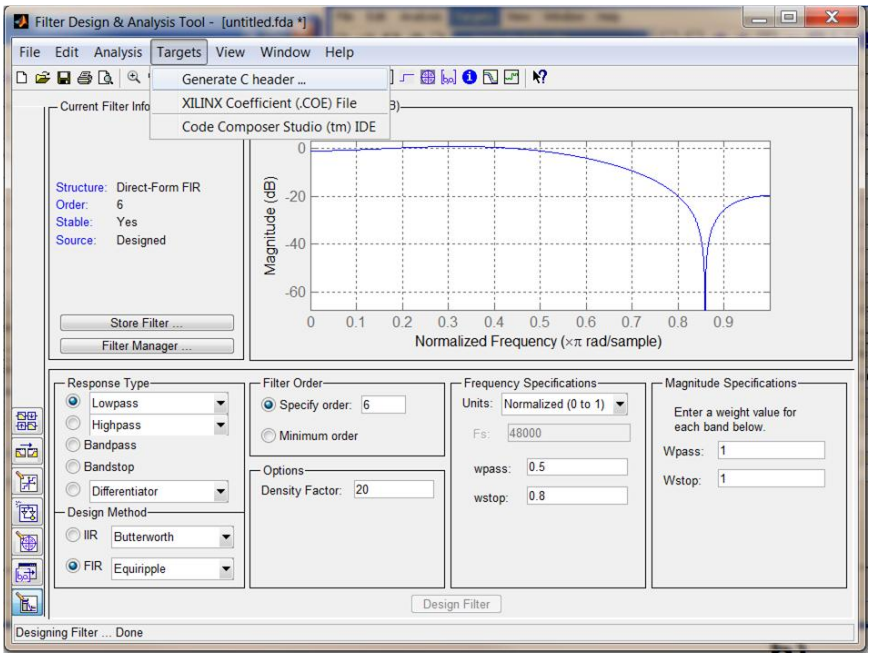


Figure 5.9: **2)**

3. After 2), 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’ (5.10).

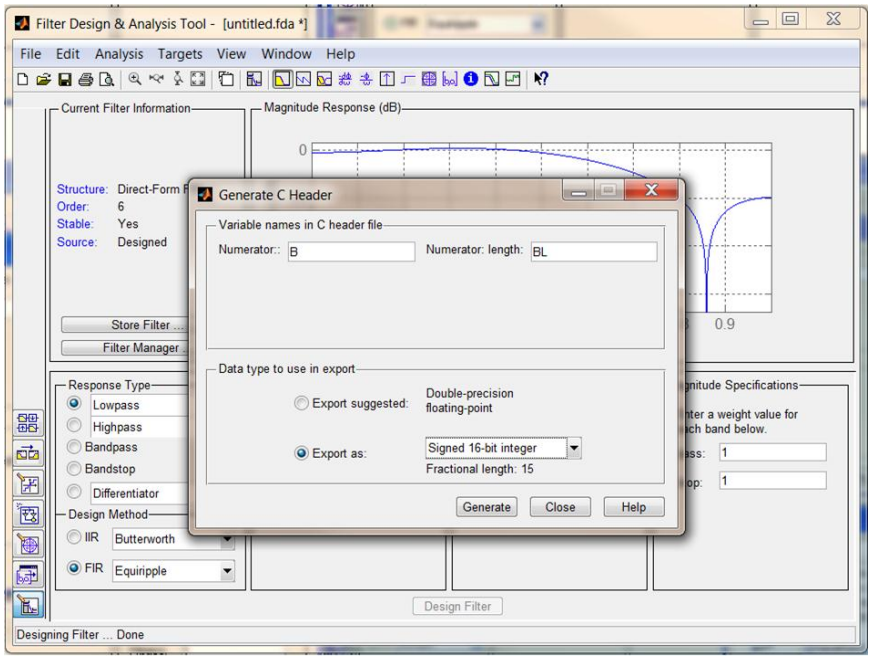


Figure 5.10: **3)**

4. Save the header file. The coefficient can directly port to 'fir.h' (5.11).

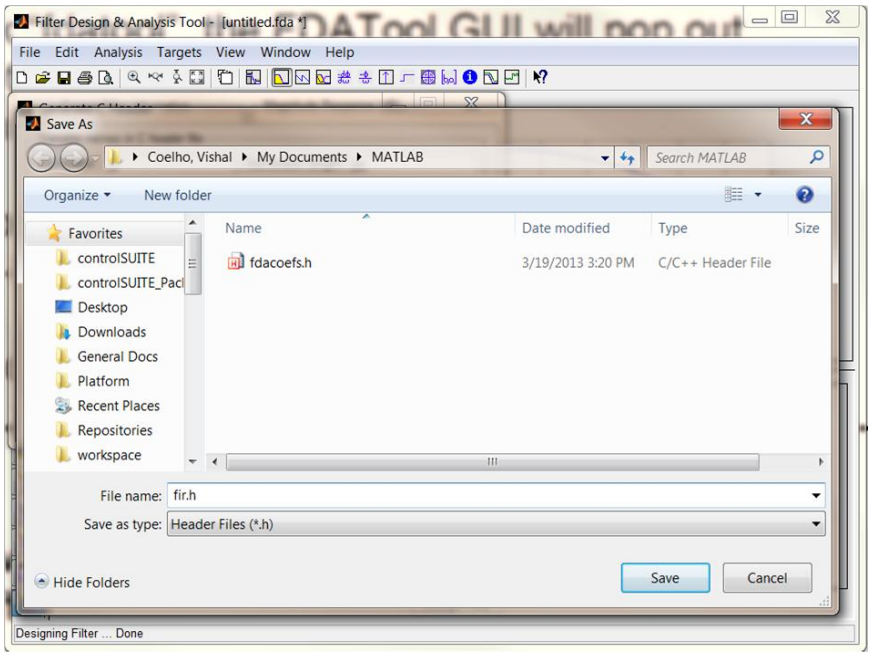


Figure 5.11: **4)**

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.

```
#include <filter.h>

// Filter Symbolic Constants
#define FIR_ORDER       32
#define SIGNAL_LENGTH   128

#if(FIR_ORDER & 0x01)   // odd
#define FIR_ORDER_REV   (FIR_ORDER + 1) // even
#else
#define FIR_ORDER_REV   (FIR_ORDER + 2) // even
#endif

// If using the alternate FIR filter, replace with
// FIR16 fir= FIR16_ALT_DEFAULTS;
FIR16 fir= FIR16_DEFAULTS;

int32_t dbuffer[(FIR_ORDER+3)/2];
int16_t sigIn[SIGNAL_LENGTH];
int16_t sigOut[SIGNAL_LENGTH];
int16_t coeff[FIR_ORDER+2];
int16_t revCoeff[FIR_ORDER+2];

// Start of main()
int main(void)
{
    .........
    if((FIR_ORDER & 0x01) == 0){
      revCoeff[FIR_ORDER_REV-1] = 0;
    }
    // Reorder the coefficients
    for(i = 0; i < FIR_ORDER + 2; i = i + 2){
        revCoeff[FIR_ORDER_REV-i-1] = coeff[i/2+FIR_ORDER_REV/2];
        revCoeff[FIR_ORDER_REV-i-2] = coeff[i/2];
    }
    .........
    // Initialize FIR16 object
fir.order       = FIR_ORDER;
fir.dbuffer_ptr = &dbuffer[0];
fir.coeff_ptr   = (int32_t *)&revCoeff[0];
fir.init(&fir);

    // FIR calculation
    for(i = 0; i < SIGNAL_LENGTH; i++){
      fir.input  = xn; // Q15 format
      fir.calc(&fir);
      yn         = fir.output;
    }
}
```

Benchmark Information:

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

FIR Taps (1 block)	C-Callable ASM
FIR16_calc	
8	47 cycles
16	51 cycles
32	59 cycles
64	75 cycles
128	107 cycles
256	171 cycles
FIR16_Alt_calc	
8	49 cycles
16	53 cycles
32	61 cycles
64	77 cycles
128	108 cycles
256	172 cycles
512	300 cycles
1024	556 cycles

Table 5.7: 16-bit FIR Routines

5.2.7 32 Bit FIR Filter Module

Description:

This module implements FIR Filter using QMACL instructions that effectively executes 1 filter tap in a cycle. There are two functions in this module, **FIR32_calc** can support up to 127^{th} order FIR filter, while **FIR32_Alt_calc** can support up to a 32767^{th} order filter.

Header File:

filter.h

Availability:

C-Callable Assembly (CcA)

Object Definition:

The structure of the FIR32 object is as follows

```
typedef struct {
  int32 *coeff_ptr;
  int32 * dbuffer_ptr;
  int16 cbindex;
  int16 order;
  int16 input;
  int16 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-bit 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); it serves as the wraparound point for the delay buffer whereas **FIR32_Alt_init** will zero it out.
order	Order of the filter	int16	Q0	Filter order, one less than the number of taps.
input	Input to the filter	int16	Q31	N/A
output	Output of the filter	int16	Q30	N/A

Continued on next page

Table 5.8 – continued from previous page

Item	Description	Format	Q-values	Comment
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.

FIR32_DEFAULTS Structure symbolic constant is to initialize FIR32 Module in order to run the alternate 32-bit FIR routines. This provides the initial values to the terminal variables as well as method pointers.

Methods:

```
void FIR32_init(FIR32_Handle);
void FIR32_calc(FIR32_Handle);
void FIR32_Alt_init(FIR32_Handle);
void FIR32_Alt_calc(FIR32_Handle);
```

Each FIR implementation has two functions, an initialization and computation function. The input argument to these functions is the module handle.

Coefficients generation:

The method to generate the 32 bit FIR filter coefficients is the same as that described under the 16 bit FIR Module description, with the only difference that in step 3, the user chooses to generate the coefficients as signed 32-bit integers with fractional length of 31.

For the **FIR32_calc** function, the coefficients need no be reordered but for the **FIR32_Alt_calc** routine they need to be in reverse order.

Note:

1. **The delay buffer is assigned to a section, "firldb", which should be aligned to 256 word boundary in the data memory (RAM) for the FIR16 module. The FIR16_Alt module does not require any alignment**
2. **The coefficients are placed in a section, "firfilt", that can be placed anywhere in the data memory (RAM)**

Example:

The following sample code runs a 32-bit FIR filter:

```
#include <filter.h>

#define FIR_ORDER       43
#define SIGNAL_LENGTH   128

FIR32 fir= FIR32_DEFAULTS;
int32_t dbuffer[FIR_ORDER+1];
```

```
int32_t sigIn[SIGNAL_LENGTH];
int32_t sigOut[SIGNAL_LENGTH];
const int32_t coeff[FIR_ORDER+1] = FIR32_LPF32_TEST;
// Alternate FIR32 requires reordering of the coefficients
// int32_t revCoeff[FIR_ORDER+1];
int main(void)
{
    .........
    // Alternate FIR32 requires reordering of the coefficients
    // for(i = 0; i < FIR_ORDER_REV + 1; i++){
    //     revCoeff[FIR_ORDER_REV-i] = coeff[i];
    // }
    .........
    // Initialize FIR32 object
    fir.order       = FIR_ORDER;
    fir.dbuffer_ptr = dbuffer;
    fir.coeff_ptr   =(int32_t *)coeff;
    fir.init(&fir);


    for(i=0; i < SIGNAL_LENGTH; i++){
      fir.input = xn;             //Input data
      fir.calc(&fir);             //FIR convolution operation
      yn        = fir.output;     //Output data
    }

}
```

Benchmark Information:

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

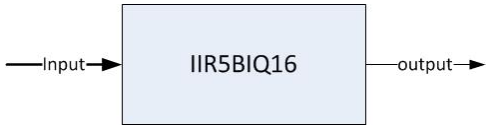
FIR Taps (1 block)	C-Callable ASM
FIR32_calc	
8	45 cycles
16	53 cycles
32	69 cycles
64	101 cycles
128	165 cycles
FIR32_Alt_calc	
8	46 cycles
16	54 cycles
32	70 cycles
64	102 cycles
128	166 cycles
256	294 cycles
512	550 cycles
1024	1062 cycles

Table 5.9: 32-bit FIR Routines

5.2.8 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.



Header File:

filter.h

Availability:

C-Callable Assembly (CcA)

Object Definition:

The structure of the IIR5BIQ16 object is as follows

```
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 biquad elements	int16	Q0	Number of bi-quad (SOS) elements. Calculated by eziir16.m
input	Input data	int16	Q15	Should be normalized to Q15 format.

Continued on next page

Table 5.10 – continued from previous page

Item	Description	Format	Q-values	Comment
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 IIR5BIQ16 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 init(IIR5BIQ16_handle);
void calc(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 found in Appendix A.

Note:

For generating IIR coefficients, IIR5BIQ16 module cannot work independently from Matlab scripts "eziir16.m" (<base>\ examples_ccsv5\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

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

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.

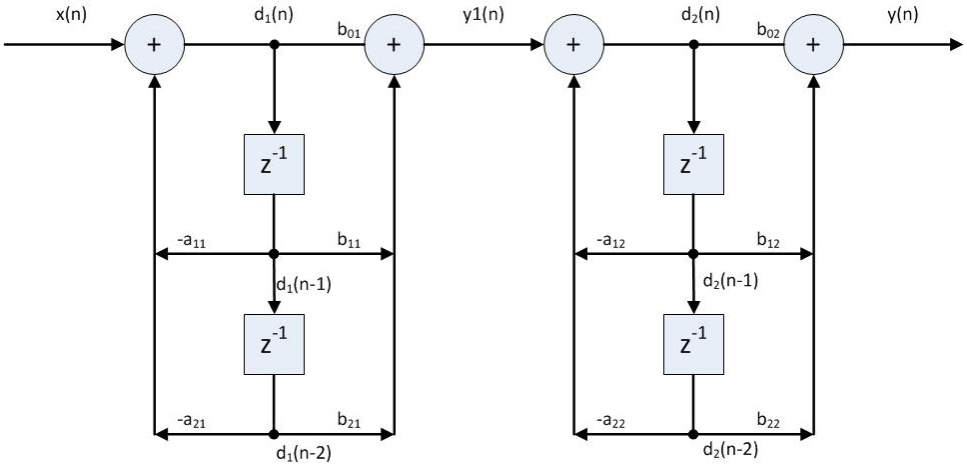


Figure 5.12: Second Order Stage (SOS) or Biquad

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.

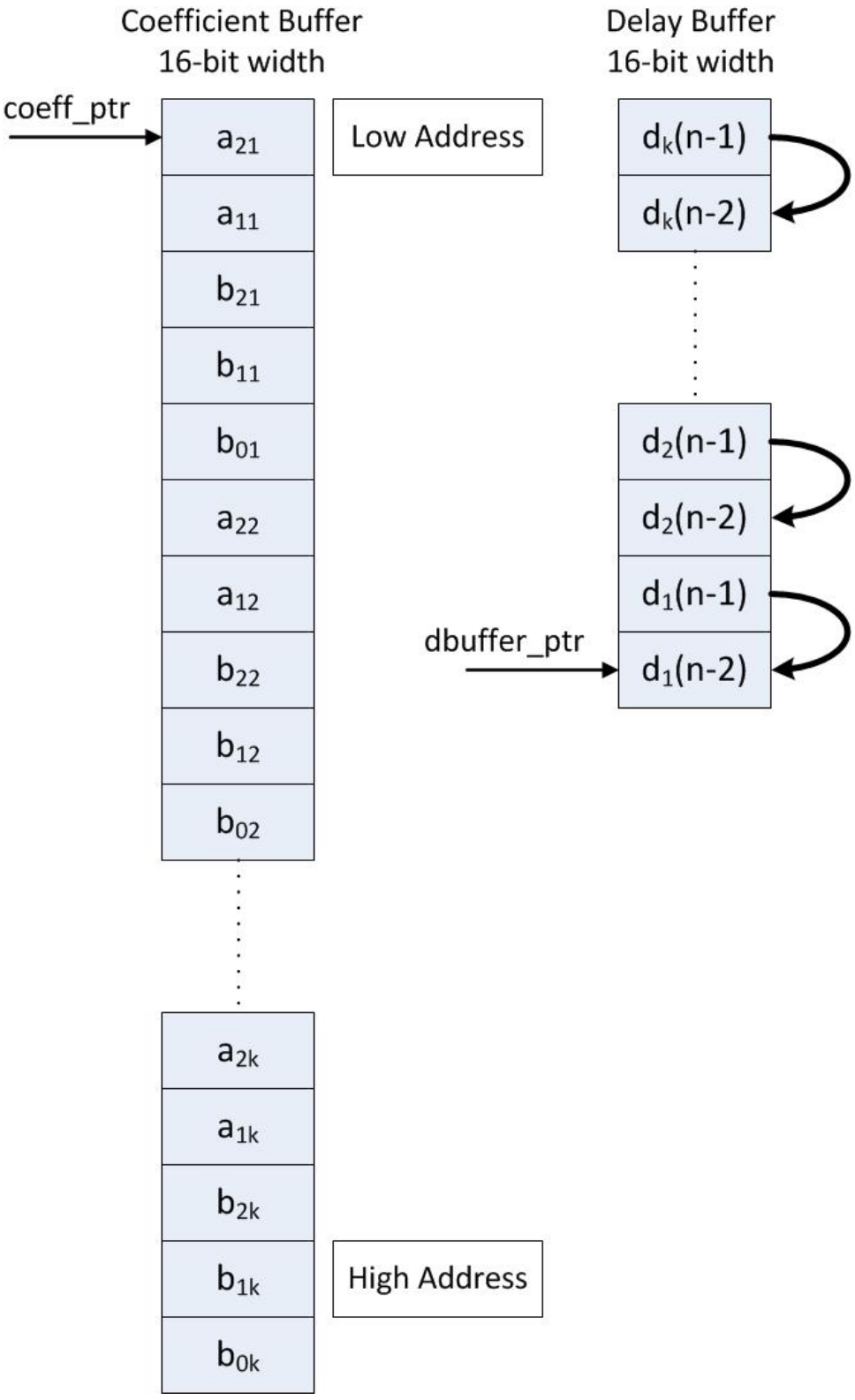


Figure 5.13: Coefficient and Delay buffer Storage

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:

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

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

5.2.9 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.

Header File:

filter.h

Availability:

C-Callable Assembly (CcA)

Object Definition:

The structure of the IIR5BIQ32 object is as follows

```
typedef struct {
  void (*init)(void *);
  void (*calc)(void *);
  int32 *coeff_ptr;
  int32 *dbuffer_ptr;
  int16 nbiq;
  int32 input;
  int16 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	Pointer to the Filter coefficient array. Arrangement can be referred to “Background Information” section.
dbuffer_ptr	Delay buffer	Pointer to 32-bit integer array	Q31	Delay buffer arrangement can be referred to “Background Information” section.
nbiq	Number of biquad elements	int32	Q0	Number of bi-quad (SOS) elements. Calculated by eziir32.m
input	Input data	int32	Q31	Should be normalized to Q31 format.

Continued on next page

Table 5.11 – continued from previous page

Item	Description	Format	Q-values	Comment
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	Q30	Output data in Q30 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 IIR5BIQ32 Filter module. To create multiple instances of the module simply declare variables of type IIR5BIQ32

IIR5BIQ32_handle User defined Data type of pointer to IIR5BIQ32 Module

IIR5BIQ32_DEFAULTS Structure symbolic constant is to initialize IIR5BIQ32 Module. This provides the initial values to the terminal variables as well as method pointers.

Methods:

```
void init(IIR5BIQ32_handle);
void calc(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 found in Appendix A.

Note:

For generating IIR coefficients, IIR5BIQ32 module cannot work independently from Matlab scripts "eziir32.m" (<base>\ examples_ccsv5\2833x_FixedPoint_IIR32\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

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

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 ex-

tremely 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.

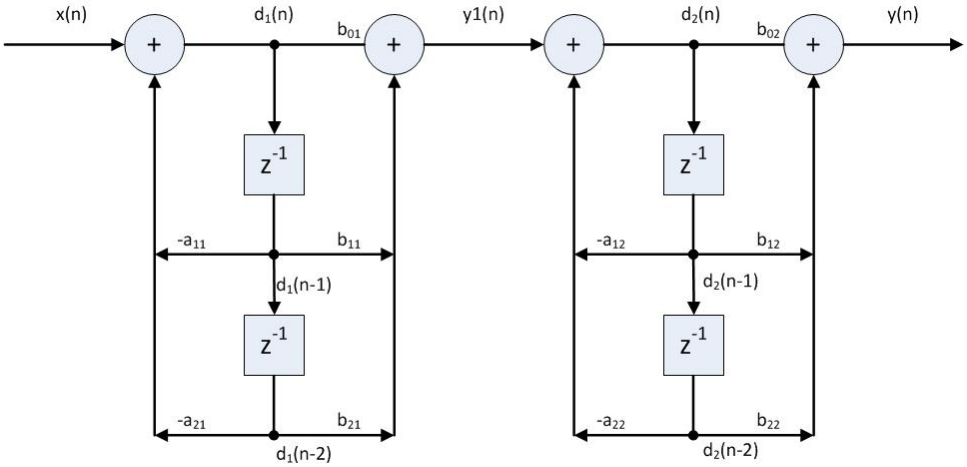


Figure 5.14: Second Order Stage (SOS) or Biquad

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 eziir16 filter design scrip to generate filter co-efficient for this module.

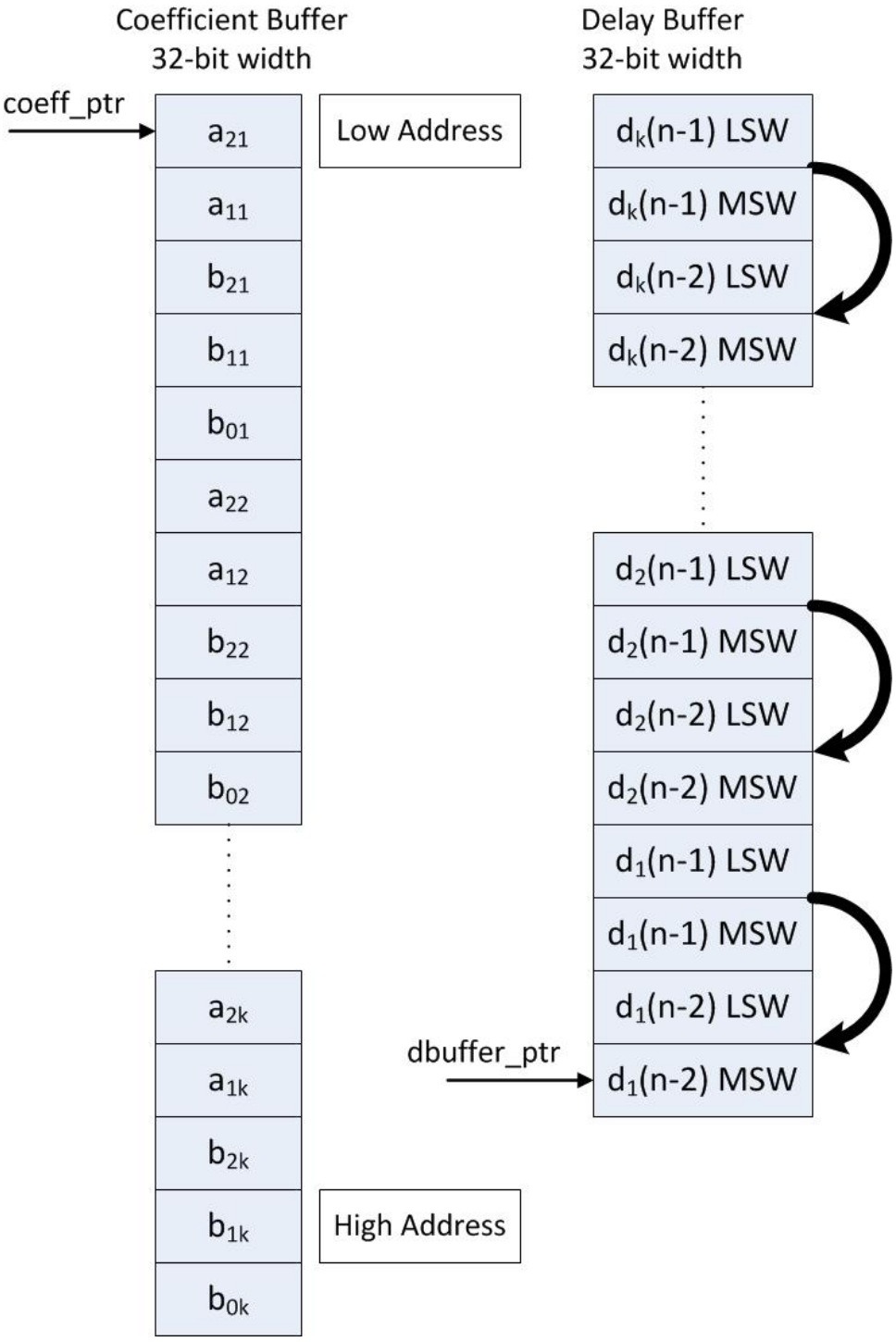


Figure 5.15: Coefficient and Delay buffer Storage

Example:

The following sample code obtains the filtered result with 16 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:

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

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

6 Revision History

V1.20.00.00: Moderate Update, October 23, 2014

- Revised documentation.
- Re-factored all library and example projects to use CGT v6.2.5.
- Updated all examples to work with CCS v5.
- FIR16/32: Added alternate routines that uses C2xLP circular addressing allowing the filters to exceed the 256(128) tap limit of the current FIR routines. Added examples and matlab scripts to demonstrate how these routines work
- fft.h: Corrected the errors in the RFFT32_<n>P_DEFAULT macros.
- Added 17 different windowing tables for FFTs up to 4096 points, each in their own header file. Also added the MATLAB script to generate them.
- Added RFFT windowing function - applies the window to both the even and odd entries of the bit reversed input buffer.
- Fixed data overrun issue with RFFT32_brev caused by not zeroing out AH prior to using a shifted ACC (AL) as a loop counter.
- Fixed issue with FIR16_calc where the loop count and wrap around address were incorrectly set to 1 less than the proper value - this was causing the filter to not use the last coefficient as well as the last element of the delay line buffer.
- Fixed issue with FIR32_init which would either not zero out the entire delay line (odd order filter) or zero out an extra word (even order filter). Also calculated the wraparound offset “cbindex” one time instead of FIR_calc calculating it on each invocation.
- Corrected size of input (32-bits) in the IIR5BIQ32 structure.
- Fixed issue with incorrect loop count in the FIR32_calc routine.
- RFFT32_brev will now bit-reverse order both even and odd locations
- Added split function that derives the complex spectrum of a 2N point real sequence, after it is run through a complex N point FFT, to get the real FFT
- Added new magnitude function for the real FFT, that computes the magnitude of the spectrum from 0 Hz to the Nyquist frequency, about which point the magnitude plot is symmetric.

V1.01: Minor Update, January 10, 2011

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

V1.00: First Release, November 1, 2010

Official release. Includes the Real FFT and real FFT and 16 bit and 32 bit FIR/IIR fixed point filter library.

Beta 1: Test Release, May 7, 2002

Includes the Real, real FFT, 16 bit FIR, 16 and 32 bit IIR fixed point filter library.

A IIR Filter Design Package User's Guide

A.1 IIR Filter Specifications

F_N	Nyquist frequency is $\frac{1}{2}$ of the Sampling Frequency.
F_P	Passband corner frequency is a scalar (F_P) or a two-element vector ($[F_{P1}, F_{P2}]$) with values between 0 and the Nyquist Frequency (F_N).
F_S	Stopband corner frequency is a scalar (F_S) or a two-element vector ($[F_{S1}, F_{S2}]$) with values between 0 and the Nyquist frequency (F_N).
R_P	Maximum permissible passband loss in decibels.
R_S	Minimum Stop-band attenuation, in decibels.

Table A.1: Description of Stop band and Pass band Filter Parameters

Filter Type	Stopband and Passband Conditions	Stopband	Passband
LPF	$F_P < F_S$, Both Scalar	$[F_S, F_N]$	[0, F_P]
HPF	$F_P < F_S$, Both Scalar	$[0, F_S]$	[F_P, F_N]
BPF	$F_{S1} < F_{P1} < F_{P2} < F_{S2}$, $F_P = [F_{P1}, F_{P2}]$ & $F_S = [F_{S1}, F_{S2}]$ are Vector	$[0, F_{S1}]$ & $[F_{S2}, F_N]$	$[F_{P1}, F_{P2}]$
BSF	$F_{P1} < F_{S1} < F_{S2} < F_{P2}$, $F_P = [F_{P1}, F_{P2}]$ & $F_S = [F_{S1}, F_{S2}]$ are Vector	$[F_{S1}, F_{S2}]$	$[0, F_{P1}]$ & $[F_{P2}, F_N]$

Table A.2: Filter Type Stopband and Passband Specifications

The information presented in the table is illustrated in the following diagrams:

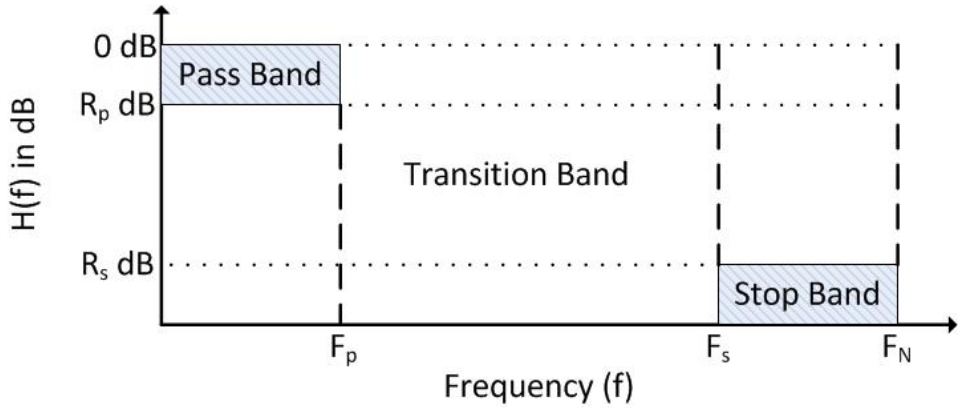


Figure A.1: LPF Filter Specification

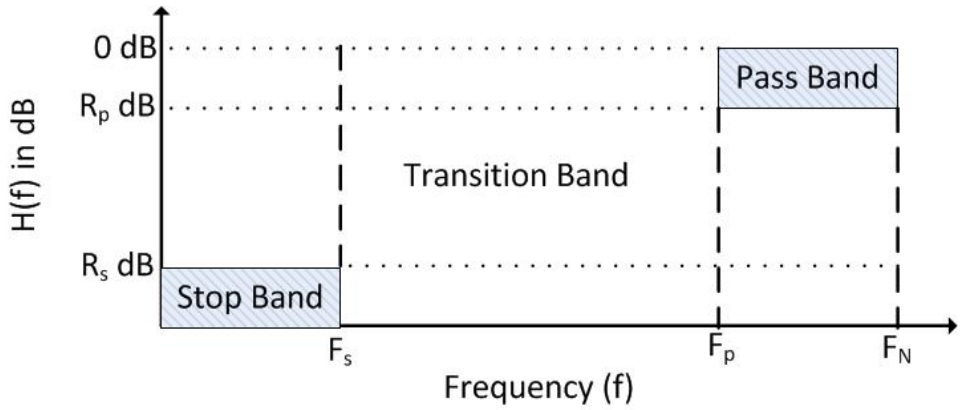


Figure A.2: HPF Filter Specification

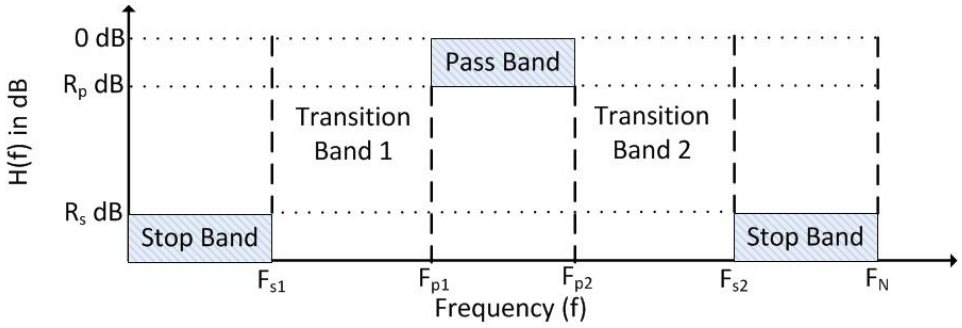


Figure A.3: BPF Filter Specification

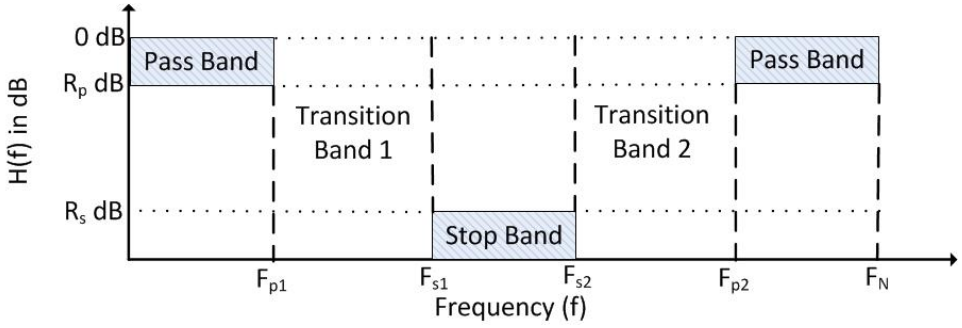


Figure A.4: BSF Filter Specification

A.2 ezIIR Filter Design Script Usage

ezIIR filter design script facilitates the user to design cascade IIR filter using Second Order Section (SOS) without any overflow issues in the internal nodes of the filter.

Step 1:

Invoke the MATLAB software and modify the current working directory to **C:\ti\c2000\C2000Ware_X_XX_XX_XX\libraries\dsp\FixedPoint\c28\examples\2833x_FixedPoint_IIR$ \matlab (here $ = 16 or 32)**

```
>> cd C:\ti\c2000\C2000Ware\_X\_XX\_XX\_XX\libraries\dsp\FixedPoint\c28\
examples\2833x_FixedPoint_IIR$\matlab (here $=16 or 32)
```

Step 2:

Execute the **eziir16** or **eziir32** script and input the required filter response parameters

Note: eziir16 script generates filter co-efficients for IIR5BIQ16 module
eziir32 script generates filter co-efficients for IIR5BIQ32 module

The script requests the user to provide following informationŠs for filter design viz.,

1. Type of Filter
2. Type of Response
3. Sampling frequency in Hz
4. Pass Band Ripples in Decibels
5. Stop Band Attenuation in Decibels
6. Pass Band Frequency in Hz
7. Stop Band Frequency in Hz
8. Name of the file to store the outputs

```
» C:\ti\c2000\C2000Ware_X_XX_XX_XX\libraries \dsp\FixedPoint\c28
\examples_ccsv5\2833x_FixedPoint_IIR16 \matlab
» eziir16


ezIIR FILTER DESIGN SCRIPT
Butterworth       : 1
Chebyshev(Type 1) : 2
Chebyshev(Type 2) : 3
Elliptic          : 4
Select Any one of the above IIR Filter Type : 1
Low pass          : 1
High Pass         : 2
Band Pass         : 3
Band Stop         : 4
Select Any one of the above Response         : 1
Enter the Sampling frequency                 : 20000
Enter the Pass band Ripples in dB(RP)        : 1
Enter the stop band Rippled in dB(RS)        : 20
Enter the pass band corner frequency(FP)     : 2000
Enter the stop band corner frequency(FS)     : 3000
Enter the name of the file for coeff storage : filter.dat
```

```
Q format of the IIR filter coefficients:
  13
Input Scaling value:
  0.5369
Number of Biquads
  4
```

Step 3:

ezIIR Filter design script outputs the following information viz.,

1. Set of SOS coefficients
2. Input Scaling Factor required to avoid the overflow in the first biquad
3. Number of Biquad required to obtain the specified filter characteristics
4. Q Format used to represent the scaled SOS filter coefficients
5. Displays the filter response in figure window 1 and window 2 (Fig A.2 A.2)
   Figure window 1, displays the magnitude response in normal scale
   Figure window 2, displays the magnitude response in logarithmic scale

FILTER.DAT

```
#define IIR16_COEFF {\
                0,3794,0,307,307,\
                -2159,7894,668,1335,668,\
                -3483,8904,541,1082,541,\
                -6131,10923,13364,26729,13364}

#define IIR16_ISF     4398
#define IIR16_NBIQ    4
#define IIR16_QFMAT   13
```

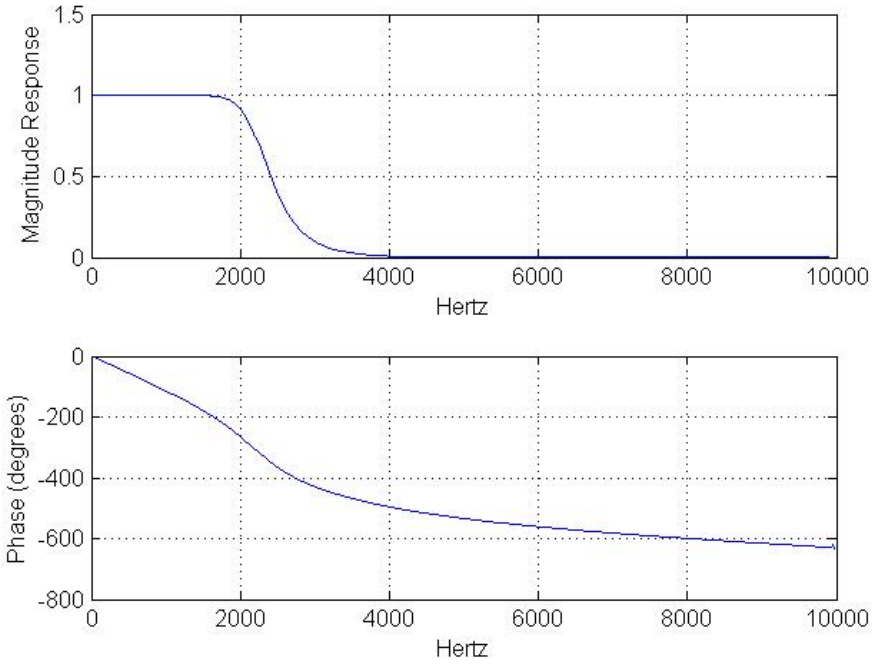


Figure A.5: Filter response using normal scale

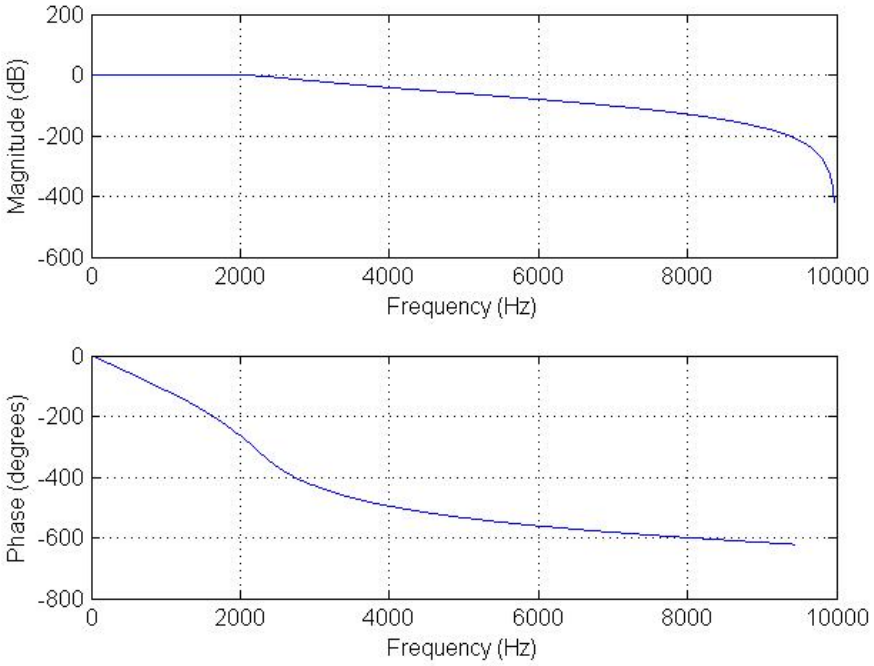


Figure A.6: Filter response using logarithmic scale

Step 4:

Rename the symbolic constants uniquely as required and copy it to the IIRCOEF file in order to initialize the filter object in the main system file.

A.3 ezIIR IIR Filter Design Examples

A.3.1 LPF Design

LPF Specification:

Filter Type	: Butterworth
Pass Band cutoff frequency (FP)	: 3000 Hz
Stop Band cutoff frequency (FS)	: 4000 Hz
Sampling Frequency	: 20 KHz
Pass Band Attenuation (RP)	: 1 dB
Stop Band Attenuation (RS)	: 30 dB

LPF Design using ezIIR script:

```
>> eziir16
ezIIR FILTER DESIGN SCRIPT
Butterworth       : 1
Chebyshev(Type 1) : 2
Chebyshev(Type 2) : 3
Elliptic          : 4
Select Any one of the above IIR Filter Type : 1
Low pass          : 1
High Pass         : 2
Band Pass         : 3
Band Stop         : 4
Select Any one of the above Response         : 1
Enter the Sampling frequency                 : 20000
Enter the Pass band Ripples in dB (RP)       : 1
Enter the stop band Rippled in dB (RS)       : 30
Enter the pass band corner frequency (FP)    : 3000
Enter the stop band corner frequency (FS)    : 4000
Enter the name of the file for coeff storage : lpf.dat

Q format of the IIR filter coefficients:
  12

Input Scaling value:
  0.4995

Number of Biquads:
  6
```

LPF.DAT

```
#define IIR16_COEFF {\
                 -373,2423,132,264,132,\
                 -516,2500,561,1121,561,\
                 -819,2665,603,1207,603,\
```

```
                    -1324,2938,624,1248,624,\
                    -2103,3361,472,944,472,\
                    -3286,4003,8606,17212,8606}

#define IIR16_ISF     2046
#define IIR16_NBIQ    6
#define IIR16_QFMAT   12
```

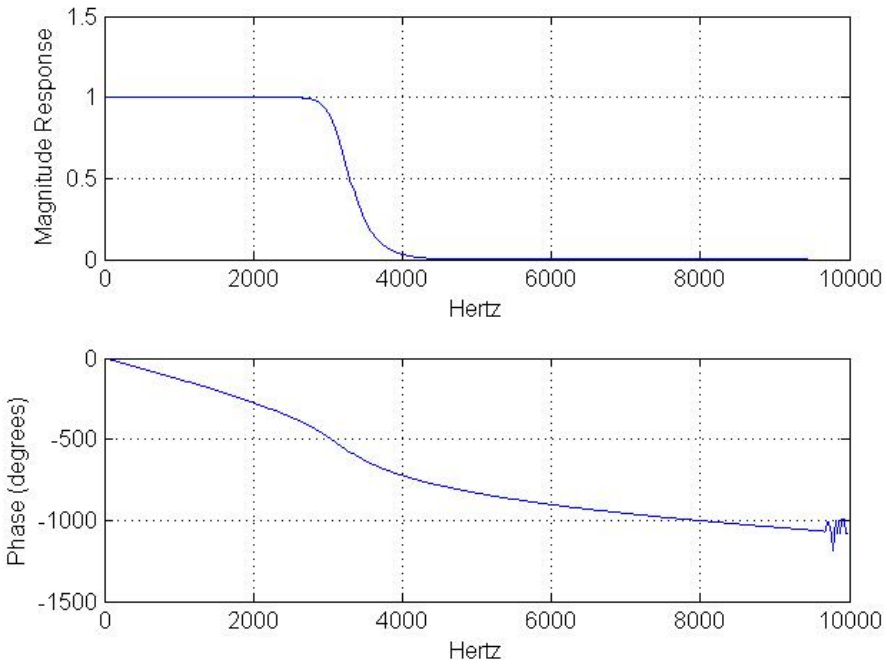


Figure A.7: Low Pass Filter response using normal scale

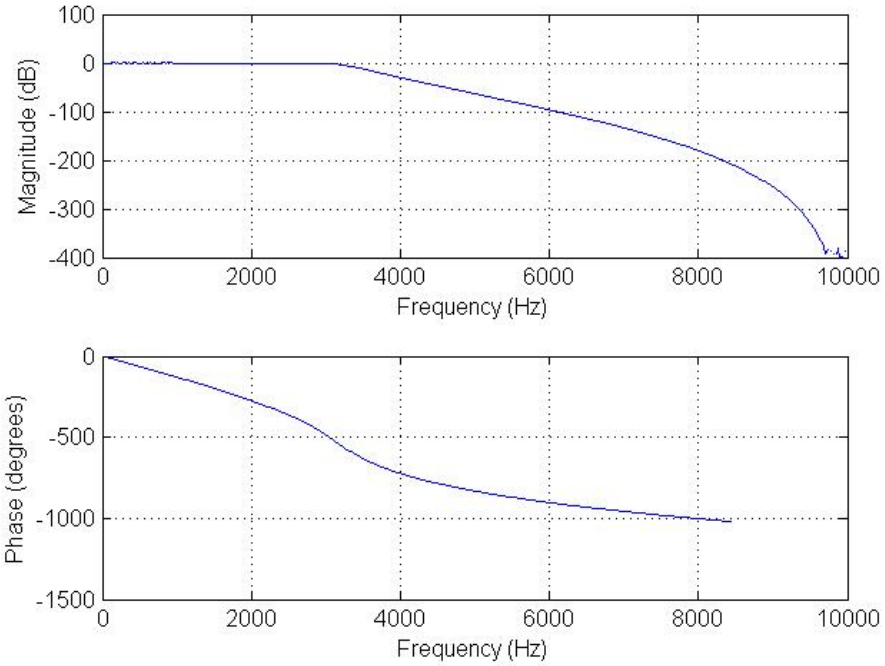


Figure A.8: Low Pass Filter response using logarithmic scale

A.3.2 HPF Design

HPF Specification:

Filter Type	: Chebyshev Type I
Pass Band cutoff frequency (FP)	: 4000Hz
Stop Band cutoff frequency (FS)	: 3000Hz
Sampling Frequency	: 20KHz
Pass Band Attenuation (RP)	: 0.1 dB
Stop Band Attenuation (RS)	: 30 dB

HPF Design using ezIIR script:

```
>> eziir16
ezIIR FILTER DESIGN SCRIPT
Butterworth       : 1
Chebyshev(Type 1) : 2
Chebyshev(Type 2) : 3
Elliptic          : 4
Select Any one of the above IIR Filter Type : 2
Low pass          : 1
High Pass         : 2
Band Pass         : 3
```

```
Band Stop          : 4
Select Any one of the above Response         : 2
Enter the Sampling frequency                 : 20000
Enter the Pass band Ripples in dB(RP)        : 0.1
Enter the stop band Rippled in dB(RS)        : 30
Enter the pass band corner frequency(FP)     : 4000
Enter the stop band corner frequency(FS)     : 3000
Enter the name of the file for coeff storage : hpf.dat

Q format of the IIR filter coefficients:
  12

Input Scaling value:
  0.6830

Number of Biquads:
  4
```

HPF.DAT

```
#define IIR16_COEFF {\
                 0,-1298,0,-456,456,\
                 -1106,-1198,1486,-2971,1486,\
                 -2372,1138,1356,-2711,1356,\
                 -3523,2655,10912,-21824,10912}

#define IIR16_ISF     2798
#define IIR16_NBIQ    4
#define IIR16_QFMAT   12
```

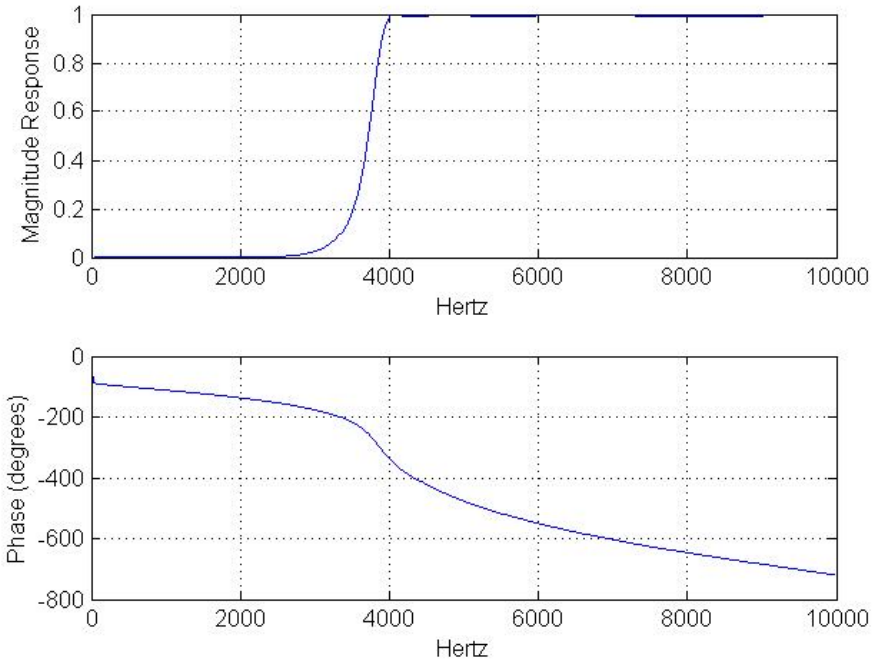


Figure A.9: High Pass Filter response using normal scale

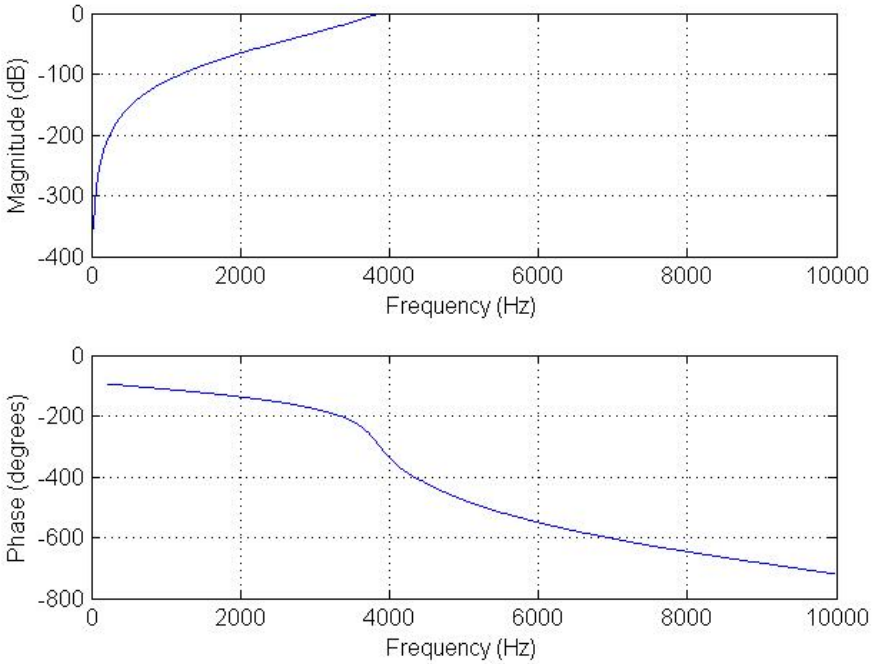


Figure A.10: High Pass Filter response using logarithmic scale

A.3.3 BPF Design

BPF Specification:

Filter Type	: Chebyshev Type II
Pass Band cutoff frequency (FP)	: [3000, 4000]Hz
Stop Band cutoff frequency (FS)	: [2500, 4500]Hz
Sampling Frequency	: 20KHz
Pass Band Attenuation (RP)	: 0.1 dB
Stop Band Attenuation (RS)	: 30 dB

BPF Design using ezIIR script:

```
>> eziir16
ezIIR FILTER DESIGN SCRIPT
Butterworth       : 1
Chebyshev(Type 1) : 2
Chebyshev(Type 2) : 3
Elliptic          : 4
Select Any one of the above IIR Filter Type : 3
Low pass          : 1
High Pass         : 2
Band Pass         : 3
Band Stop         : 4
Select Any one of the above Response         : 3
Enter the Sampling frequency                 : 20000
Enter the Pass band Ripples in dB(RP)        : 0.1
Enter the stop band Rippled in dB(RS)        : 30
Enter the pass band corner frequency(FP)     : [3000,4000]
Enter the stop band corner frequency(FS)     : [2500,4500]
Enter the name of the file for coeff storage : bpf.dat

Q format of the IIR filter coefficients:
  11

Input Scaling value:
  0.3849

Number of Biquads:
  5
```

BPF.DAT

```
#define IIR16_COEFF {\
                 -986,1448,-408,0,408,\
                 -1323,889,648,96,648,\
                 -1452,2263,672,-1082,672,\
                 -1830,952,950,-264,950,\
                 -1877,2581,17695,-25335,17695}
```

```
#define IIR16_ISF     788
#define IIR16_NBIQ    5
#define IIR16_QFMAT   11
```

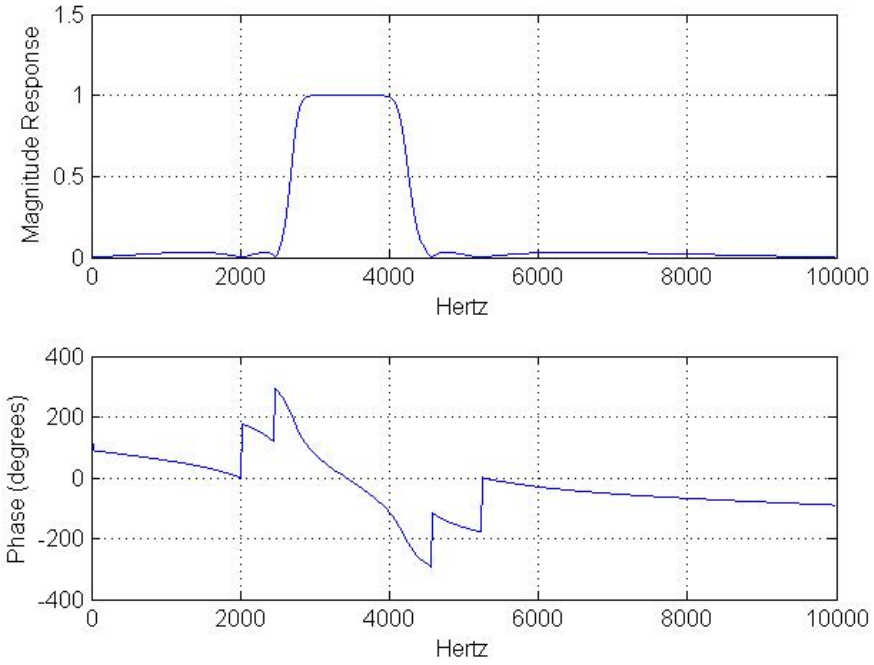


Figure A.11: Band Pass Filter response using normal scale

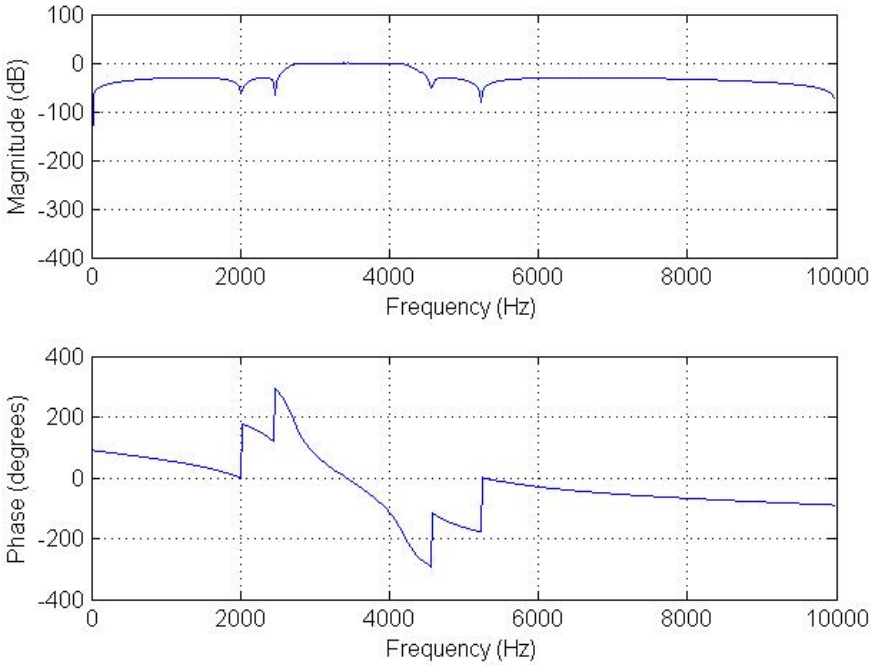


Figure A.12: Band Pass Filter response using logarithmic scale

A.3.4 BSF Design

BSF Specification:

Filter Type	: Elliptic
Pass Band cutoff frequency (FP)	: [3000, 4000]Hz
Stop Band cutoff frequency (FS)	: [3400, 3600]Hz
Sampling Frequency	: 20KHz
Pass Band Attenuation (RP)	: 0.1 dB
Stop Band Attenuation (RS)	: 30 dB

BSF Design using ezIIR script:

```
>> eziir16
ezIIR FILTER DESIGN SCRIPT
Butterworth       : 1
Chebyshev(Type 1) : 2
Chebyshev(Type 2) : 3
Elliptic          : 4
Select Any one of the above IIR Filter Type : 4
Low pass          : 1
High Pass         : 2
Band Pass         : 3
```

```
Band Stop          : 4
Select Any one of the above Response         : 4
Enter the Sampling frequency                 : 20000
Enter the Pass band Ripples in dB(RP)        : 0.1
Enter the stop band Rippled in dB(RS)        : 30
Enter the pass band corner frequency(FP)     : [3000,4000]
Enter the stop band corner frequency(FS)     : [3400,3600]
Enter the name of the file for coeff storage : bsf.dat

Q format of the IIR filter coefficients:
  11

Input Scaling value:
  0.1848

Number of Biquads:
  3
```

BSF.DAT

```
#define IIR16_COEFF {\
                 -1515,1638,859,-790,859,\
                 -1883,1378,1998,-1631,1998,\
                 -1901,2198,21576,-21934,21576}

#define IIR16_ISF     379
#define IIR16_NBIQ    3
#define IIR16_QFMAT   11
```

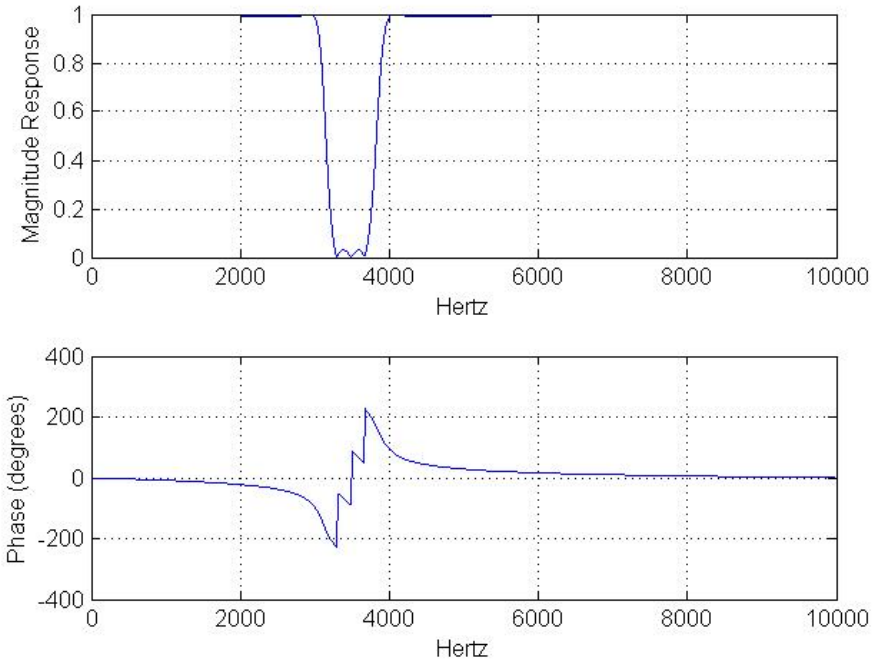


Figure A.13: Band Stop Filter response using normal scale

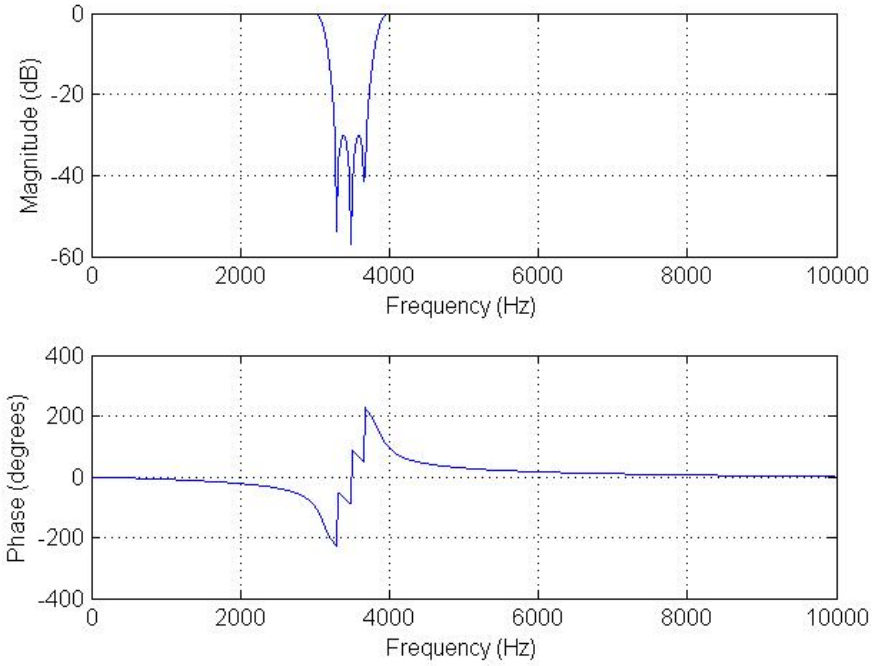


Figure A.14: Band Stop Filter response using logarithmic scale

A.4 Test coefficients for IIR filter

To demonstrate the I**R5BIQ16 & IIR5BIQ16** filter modules, we have generated filter co-efficient for LPF, HPF, BPF and BSF responses using **eziir16 & eziir32** script and placed it in IIR.H header file.

These test co-efficients are generated using the same filter specification given in filter design examples in previous section.

IIR.H : Test Co-efficients for IIR5BIQ16 Module

```
{\bf /* LPF co-efficients for IIR16 module */}
#define IIR16_LPF_COEFF {\
                        -746,4846,1056,2111,1056,\
                        -1032,5001,1120,2239,1120,\
                        -1639,5330,1192,2385,1192,\
                        -2647,5877,1211,2422,1211,\
                        -4206,6722,872,1745,872,\
                        -6573,8005,4861,9722,4861}
#define IIR16_LPF_ISF   4092
#define IIR16_LPF_NBIQ  6
#define IIR16_LPF_QFMAT 13

{\bf /* HPF co-efficients for IIR16 module */}
#define IIR16_HPF_COEFF {\
                         0,-2597,0,-3340,3340,\
                         -2211,-2396,1746,-3492,1746,\
                         -4745,2276,2007,-4014,2007,\
                         -7046,5310,13685,-27370,13685}
#define IIR16_HPF_ISF    5595
#define IIR16_HPF_NBIQ   4
#define IIR16_HPF_QFMAT  13

{\bf /* BPF co-efficients for IIR16 module */}
#define IIR16_BPF_COEFF {\
                         -1078,1437,-367,0,367,\
                         -1395,935,713,43,713,\
                         -1496,2176,594,-917,594,\
                         -1855,994,1022,-329,1022,\
                         -1890,2462,18610,-25359,18610}
#define IIR16_BPF_ISF    721
#define IIR16_BPF_NBIQ   5
#define IIR16_BPF_QFMAT  11

{\bf /* BSF co-efficients for IIR16 module */}
#define IIR16_BSF_COEFF {\
                         -1532,1626,859,-781,859,\
                         -1889,1374,2032,-1644,2032,\
                         -1906,2168,22098,-22158,22098}
#define IIR16_BSF_ISF    366
#define IIR16_BSF_NBIQ   3
#define IIR16_BSF_QFMAT  11
```

IIR.H : Test Co-efficients for IIR5BIQ32 Module

```
{\bf /* LPF co-efficients for IIR32 module */}
#define IIR32_LPF_COEFF {\
    -24444800,158794151,8647611,17295223,8647611,\
    -33805581,163869390,36741777,73483554,36741777,\
    -53695266,174653202,39535955,79071910,39535955,\
    -86750921,192575355,40880726,81761451,40880726,\
    -137806611,220256787,30931379,61862757,30931379,\
    -215373186,262311922,564004144,1128008289,564004144}
#define IIR32_LPF_ISF    134086103
#define IIR32_LPF_NBIQ   6
#define IIR32_LPF_QFMAT  28

{\bf /* HPF co-efficients for IIR32 module */}
#define IIR32_HPF_COEFF {\
    0,-85096979,0,-29857202,29857202,\
    -72466417,-78522171,97363917,-194727833,97363917,\
    -155480100,74571693,88837952,-177675903,88837952,\
    -230891969,173985995,715131301,-1430262602,715131301}
#define IIR32_HPF_ISF    183338477
#define IIR32_HPF_NBIQ   4
#define IIR32_HPF_QFMAT  28

{\bf /* BPF co-efficients for IIR32 module */}
#define IIR32_BPF_COEFF {\
    -70620977,94154018,-24080177,0,24080177,\
    -91416523,61304763,46717868,2819319,46717868,\
    -98072227,142585344,38933432,-60070942,38933432,\
    -121545500,65135203,66986143,-21532217,66986143,\
    -123895267,161346553,1219596683,-1661914443,1219596683}
#define IIR32_BPF_ISF    47247113
#define IIR32_BPF_NBIQ   5
#define IIR32_BPF_QFMAT  27

{\bf /* BSF co-efficients for IIR32 module */}
#define IIR32_BSF_COEFF {\
    -100408845,106578277,56314524,-51161340,56314524,\
    -123814194,90035192,133182068,-107730821,133182068,\
    -124894576,142053666,1448196385,-1452129490,1448196385}
#define IIR32_BSF_ISF    23999390
#define IIR32_BSF_NBIQ   3
#define IIR32_BSF_QFMAT  27
```

Bibliography

[1] TI Application Report, *: Overflow Avoidance Techniques in Cascaded IIR Filter Implementations on TMS320 DSP's*. SPRA509.

[2] Alan V. Oppenheim, Ronald W. Schafer, *Scaling in Fixed Point Implementation of IIR systems (p. 359-370)*. Discrete Time Signal Processing, Prentice Hall Signal Processing Series, 1st edition, 1989.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI 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. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

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