

Accessing DSPLIB in Processor SDK RTOS

ABSTRACT

DSPLIB is part of the Processor SDK RTOS release. The library was optimized to run on different device cores. This application note details the usage of DSPLIB for a single C66 core. Multiple device families (such as C66xx, 66AK2Hxx, 66AK2Kxx, and AM57xx) use the C66 core as a DSP engine. This note details the use of some common functions, algorithm descriptions, and performance analysis in DSPLIB (version 3.4.0.0).

NOTE: Assume little-endian mode in this document.

Contents

1	Introduction	2
2	FFT Family of Functions	2
3	Using DSPLIB FFT Functions.....	5
4	Biquad Function.....	7
5	About FIR	8
6	References	8
Appendix A	DSPLIB Documentation in Release	9

Trademarks

Windows is a registered trademark of Microsoft Corporation.
 All other trademarks are the property of their respective owners.

1 Introduction

The Processor SDK RTOS release contains optimized libraries that were developed for specific cores. Not all libraries were optimized for all cores or devices. Each function has an optimized version for the specific core, as well as a standard ANSI C (natural C) version that can be compiled and used on any core or device.

This document details DSPLIB for devices with a C66 core (or multiple cores). Each function in the DSPLIB has its own directory (`dsplib_c66x_X_Y_Z_T\packages\ti\dsplib\src` where `dsplib_c66x_X_Y_Z` is where DSPLIB was installed). Each function has a unit test code with a postfix `_d`, a standard ANSI C code with postfix `_cn`, and an optimized version without a postfix. In addition, required include files are in the function directory as well as CCS projects that enable users to build and run the unit tests. Instructions for building and running unit tests are available in [1].

A high-level description of parameters and the return values of the library functions is part of the release. [Appendix A](#) details how to access the high-level documentation of each function.

The goal of this document is to add more information about the use of library functions such as the format of parameters, limitations of parameters, performance considerations, how to generate auxiliary data, and the minimum and maximum number of elements that can be processed.

2 FFT Family of Functions

Fast Fourier Transform (FFT) may be the most important signal processing tool that is used on many applications. DSPLIB contains many FFT routines. The C66 core version of the DSPLIB has 16 variations of the FFT and inverse FFT (IFFT) algorithm that cover fixed point, single-precision floating point, and double-precision floating point for FFT and IFFT cases.

In addition, a special FFT library (FFTLIB) is available for some devices. The focus of the FFTLIB is on multicore implementation of appropriate FFT for multicore devices. Using multiple cores in parallel speeds up the execution of large FFT substantially. The FFTLIB support speed-up techniques for special FFT cases. This document does not cover FFTLIB functions.

2.1 Decimation-in-Time (DIT) and Decimation-in-Frequency (DIF)

The classic Cooley–Tukey FFT algorithm is based on breaking up a block of N (number of) FFT elements into two blocks (each the size of $N/2$). The process then repeats until the block size is two. The breaking (decimation) of the larger block can be applied to the input vector; In that case, the algorithm is called decimation-in-time. (Fourier transform is considered to transform time domain into frequency domain, so the input to FFT is called time domain). The decimation can be done on the result vector where it is called Decimation in Frequency, or DIF. All FFT functions in DSPLIB support number of elements that is power of 2 in DIF algorithm. Note that FFTLIB has functions that support sizes that are not power of 2.

The decimation algorithm is made of structures called butterfly. Classic butterfly has two complex number as inputs, use unit circle complex values called twiddle factors to generate two new complex numbers written on top of the original two complex numbers (what is called “in place” algorithm). More efficient algorithms use butterflies that have four complex input and generate four complex output. These butterflies are called Radix 4 butterflies as opposed to Radix 2 butterflies where there are two input complex numbers and two outputs.

TI’s implementations of FFT uses only Radix 4 butterfly for number of elements that is power of 4. For FFT with number of elements not power of 4, (but power of 2) the algorithm uses Radix 4 until the last phase and Radix 2 in the last phase.

2.2 Bit Reversal

When the FFT algorithm is done in place, there is a specific permutation of the order between the input and the output vector. TI implementations of the FFT are done in place up to the last phase where the output is written to a different vector. During the last phase the output is re-ordered for the correct order. That is, TI implementations input and output data are not in bit-reversal format.

Note that the parameter unsigned char *brev that is part of the API of some functions is not needed for the execution of the function. The parameter may be there to provide backward compatibility with previous versions of the library or a version for another device. In any use, the user must specify an unsigned char vector brev of any size and any contains.

2.3 Twiddle Factors

The user must provide a vector of the twiddle factors that are used during the execution. The twiddle factors are the values of the cosine and sine functions with arguments on the unit circle. The way the FFT algorithm uses the twiddle factor effects how the twiddle factors are generated.

In the first phase of Radix 4 DIF FFT the data is divided into 4 blocks, and each block has N/4 complex numbers. Combining two blocks into one requires N/4 different complex twiddle factors.

In the second phase of Radix 4 DIF FFT the data is divided into 16 blocks, and each block has N/16 complex numbers. Combining two blocks into one requires N/16 different complex twiddle factors.

This pattern continues. For example, in phase P, the data is divided into 4^{**p} block, and each block has $N/(4^{**p})$ complex numbers. Combining two blocks into one requires $N/(4^{**p})$ complex different twiddle factors. See [Computing FFT Twiddle Factors](#) for an example and details about Radix 2 DIF and project the results into Radix 4 DIF FFT.

However, the first complex twiddle factor for any butterfly is $1 + 0j$ and thus does not require multiplication. As a result, TI's DIF FFT implementation does not use the first complex twiddle factor from the twiddle factor array for each butterfly, and this value must be omitted from the twiddle-factor vector. A simple program to generate twiddle factors is given in the unit test code of any FFT function in the release. The following code snippet is an example for single precision twiddle-factor generation taken from the unit test of the function DSPF_sp_fftSPxSP. A similar program (but not identical because of the different sign in the twiddle factor exponent) exists for IFFT.

```
void tw_gen (float *w, int n)//This is for FFT, IFF will be slightly different
{
    int i, j, k;
    const double PI = 3.141592654;
    for (j = 1, k = 0; J <= n >> 2; J = J << 2)
    {
        for (i = 0; i < n >> 2; i += j)
        {
            w[k] = (float) sin (2 * PI * i / n);
            w[k + 1] = (float) cos (2 * PI * i / n);
            w[k + 2] = (float) sin (4 * PI * i / n);
            w[k + 3] = (float) cos (4 * PI * i / n);
            w[k + 4] = (float) sin (6 * PI * i / n);
            w[k + 5] = (float) cos (6 * PI * i / n);

            k += 6;
        }
    }
}
```

Note that even if the number of elements is not a power of 4 and the last phase is Radix 2, the twiddle factors are still generated in the same way.

2.4 Twiddle Factors for Real FFT

The FFT transfer of a real-elements sequence has a conjugate symmetry attribute. This feature enables optimization of the FFT transfer. The same is true for inverse FFT of a conjugate symmetric. An optimized algorithm can use this fact to speed up execution. See [Implementing Fast Fourier Transform Algorithms of Real-Valued Sequences With the TMS320 DSP Platform](#) or [FFT of Pure Real Sequences](#) for examples.

TI implementation of real sequence FFT takes advantage of the symmetry feature to speed up the processing. See [Efficient FFT Computation of Real Input](#) for details about the algorithm that is used by the TI library.

However, the code requires a slightly different twiddle factor sequence. The unit tests of each of the real FFT implementations and the symmetry IFFT implementations can be found in the unit test of each function. The following is the twiddle factor generation code from the DSPF_sp_fftSPxSP_r2c_d.c unit test:

```
void tw_gen (float *w, int n)
{
    int i, j, k;
    const double PI = 3.141592654;

    for (i = 1, k = 0; i <= n >> 2; i++) {
        w[k    ] = sin (2 * PI * i / n);
        w[k + 1] = cos (2 * PI * i / n);

        k +=2;
    }
    for (i = 1; j <= n >> 3; j = j << 2) {
        for (i = 0; i < n >> 3; i += j) {

            w[k]          = (float) sin (4 * PI * i / n);
            w[k + 1] = (float) cos (4 * PI * i / n);
            w[k + 2] = (float) sin (8 * PI * i / n);
            w[k + 3] = (float) cos (8 * PI * i / n);
            w[k + 4] = (float) sin (12 * PI * i / n);
            w[k + 5] = (float) cos (12 * PI * i / n);
            k += 6;
        }
    }
}
```

3 Using DSPLIB FFT Functions

Each DSPLIB function has a unit test project and a natural C model of the algorithm to make it easier to use the function in an application. A description of typical parameters of FFT and IFFT functions is detailed in the following subsections.

3.1 Floating Point FFT Functions

```
void DSPF_sp_fftSPxSP_opt (int N, float *ptr_x, float *ptr_w, float *ptr_y, unsigned char *brev,
int n_min, int offset, int n_max);
```

DSPF_sp_fftSPxSP_opt is the optimized version of FFT. [Table 1](#) lists the parameters for this function.

Table 1. DSPF_sp_fftSPxSP_opt Parameters

Parameter	Description
int N	Number of elements. This must be a power of 2.
float *ptr_x	A pointer to the complex input vector. The vector size is 2 x N because the real and imaginary parts are interleaves as follows: Real[0] Image[0] Real[1] Image[1] And so on.
float *ptr_w	A point to the twiddle factor array. The contains of the vector and the order of elements are as described in Section 2.3 (and in the function unit test).
float *ptr_y	A pointer to the output results. While the FFT algorithm is done in place, the last stage is written to the output vector to arrange the output in normal order and not in bit reversal order. The vector size is 2*N because the real and imaginary parts are interleaves as follows: Real[0] Image[0] Real[1] Image[1] And so on.
unsigned char *brev	The brev vector is not used in this routine. The API has backward compatibility with older functions. However, for the tools to build the code the user must supply a float pointer to some array.
int n_min	The minimum Radix size. If the number of elements is a power of 4, n_min must be 4. If it is not a power of 4, n_min must be 2.
int offset	Offset is the location of the first output in the resulted vector. It is used when multiple cores process a large FFT by dividing the FFT into smaller blocks, and then use one core to complete the last few butterflies. For a single core implementation, the offset is usually set to 0
int n_max	n_max determines how many phases will be executed. The parameter is used when multiple cores process a large FFT by dividing the FFT into smaller blocks, and then use one core to complete the last few butterflies. For a single core implementation, n_max is usually set to N.

```
void DSPF_sp_fftSPxSP_r2c (int N, float *ptr_x, float *ptr_w, float *ptr_y,
unsigned char *brev, int n_min, int offset, int n_max)
```

DSPF_sp_fftSPxSP_r2c parameters are the same as DSPF_sp_fftSPxSP except for the input vector. This function expects real values as the input and not complex. While the input vector for the C model code (DSPF_sp_fftSPxSP_r2c_cn.c) is real value [0], 0.0, and so on, the optimized version eliminates the image portion from the input vector.

When the optimized version eliminates the image portion, the input vector is: real [0], real [1], real [2], and so on. All other parameters are the same.

```
void DSPF_sp_ifftSPxSP_opt (int N, float *ptr_x, float *ptr_w, float *ptr_y,
unsigned char *brev, int n_min, int offset, int n_max)
```

DSPF_sp_ifftSPxSP_opt parameters are the same as DSPF_sp_fftSPxSP except for the following two changes.

The twiddle factors are generated differently (see [Section 2.3](#)) and the output value is scaled by $1.0 / N$ as the definition of IFFT implies.

NOTE: There are multiple ways to scale FFT and IFFT to maintain Parseval's identity. See [Parseval's identity](#).

One way is to scale both results with the square root of N . The other way is used here, where FFT results are not scaled at all, and IFFT results are scaled by $1.0 / N$ (see [Scaling FFT output by number of points in FFT](#)).

3.2 Fixed-Point FFT Functions

Fixed point arithmetic has several challenges. The first challenge is the manipulation of the binary dot or the Q format of the data, the twiddle factors, and the results. The second challenge is the truncation error fixed-point FFT and IFFT algorithm that depends on the size of the FFT or IFFT. Large FFT implementations are meaningless because the truncation error is larger than the number of bits in the results.

DSPLIB provides 10 fixed-point functions to cover multiple used cases, each with its unit test. The unit test project is a good starting point for understanding the usage of these functions.

When the input format is 16-bit, special attention should be paid to the endianness of the data and the order of the real and imaginary part in the input, output, and the twiddle factor vectors. The following are some of the APIs:

- **void DSP_fft16x16 (const short * restrict ptr_w, int npoints, short * restrict ptr_x, short * restrict ptr_y)** — Data and twiddle factor are 16 bits (short)
- **void DSP_fft32x32 (const short * restrict ptr_w, int npoints, int * restrict ptr_x, int * restrict ptr_y)** — Data and twiddle factor are 32 bits (int)
- **void DSP_fft32x16 (const short * restrict ptr_w, int npoints, int * restrict ptr_x, int * restrict ptr_y)** — Data is 32 bit (int) and twiddle factor is 16 bits (short)
- **void DSP_fft16x16_imre (const short * restrict ptr_w, int npoints, short * restrict ptr_x, short * restrict ptr_y)** — Data and twiddle factor are 16 bits (short) with order of imaginary first and then real (thus the imre notation)

The usage of the fixed point function is explained in great details in the source code. [Table 2](#) lists the parameters and their descriptions.

Table 2. Fixed-Point Parameter Descriptions

Parameter	Description
const short * restrict ptr_w const int * restrict ptr_w	Pointer to the twiddle factor vector. Functions to build the twiddle factor are part of the release and are explained in the details in the code.
int npoints	Number of elements; must be a power of 2. The Radix is determined by the FFT and IFFT function.
int * restrict ptr_x short * restrict ptr_x	Pointer to the input complex vector where the real and image part are interleaved.
int * restrict ptr_y short * restrict ptr_y	Pointer to the input complex vector where the real and image part are interleaved.

NOTE: The restrict directive tells the compiler that no other vector shares a variable with the restrict vector.

NOTE: For usage of other fixed-point FFT, read the detailed description in the source code and in the C model code.

3.3 FFT Performances

Most of the unit test main functions use a timer to benchmark the performances of the natural C code as well as the optimized code. It is important to build the unit test project with full optimization and to suppress all symbolic debugging to measure the library performances.

Some of the FFT functions have assembly versions in addition to the C optimized code. The unit test of these functions benchmarks the assembly code as well. The following code snippet is an example of floating-point, single-precision FFT performances for different data sizes that are printed by the DSPF_sp_fftSPxSP_d unit test of the DSPF_sp_fftSPxSP function (some printings were omitted for clarity). natC is the natural C model of the code, optC is the optimized version of the code using C intrinsic, and SA is the optimized assembly code. The user should choose the correct functions for the application based on the FFT size.

DSPF_sp_fftSPxSP	N = 8	radix = 2	natC: 363	optC: 147	SA: 198
DSPF_sp_fftSPxSP	N = 16	radix = 4	natC: 558	optC: 169	SA: 198
DSPF_sp_fftSPxSP	N = 32	radix = 2	natC: 1207	optC: 279	SA: 300
DSPF_sp_fftSPxSP	N = 64	radix = 4	natC: 2290	optC: 486	SA: 442
DSPF_sp_fftSPxSP	N = 128	radix = 2	natC: 5350	optC: 1289	SA: 1056
DSPF_sp_fftSPxSP	N = 256	radix = 4	natC: 10602	optC: 2793	SA: 1953
DSPF_sp_fftSPxSP	N = 512	radix = 2	natC: 24613	optC: 6597	SA: 4548
DSPF_sp_fftSPxSP	N = 1024	radix = 4	natC: 48892	optC: 13230	SA: 8384

4 Biquad Function

One way of implementing an infinite-impulse response (IIR) filter is by using a cascade of biquad filters. Biquad filter output $Y[i]$ is a function of input signal $X[i]$ and the previous outputs $Y[i-1]$ and $Y[i-2]$ (see [Equation 1](#)).

$$a_0 \times Y[n] = b_0 \times X[n] + b_1 \times X[n - 1] + b_2 \times X[n - 2] - a_1 \times Y[n - 1] - a_2 \times Y[n - 2] \quad (1)$$

Assuming that a_0 is not zero, dividing by a_0 yields the canonical presentation, that is, the same formula where a_0 is equal to 1 (and is ignored). See [Equation 2](#).

$$Y[n] = b_0 \times X[n] + b_1 \times X[n - 1] + b_2 \times X[n - 2] - a_1 \times Y[n - 1] - a_2 \times Y[n - 2] \quad (2)$$

DSPLIB is optimized for multiple TI DSP cores. This page details DSPLIB for the C66x core that is part of multiple TI devices (AM57x, C667x, 66AK2Hxx, 66AK2Kxx), but information about the algorithm and usage is similar to the same function in DSPLIB that was optimized for another core. See [TMS320C67x DSP Library Programmer's Reference Guide](#) and [TMS320C55x DSP Library Programmer's Reference](#) for an example.

The algorithm of the optimized biquad function is given by the natural C model of the function DSPF_sp_biquad, and is shown in the following code snippet:

```
void DSPF_sp_biquad_cn (float *x, float *b, float *a,
    float *delay, float *y, const int nx)
{
    int i;

    for (i=0; i < nx;i++)
    {
        y[i] = b [0] * x[i] + delay [0];
        delay [0] = b [1] * x[i] - a [1] * y[i] + delay [1];
        delay [1] = b [2] * x[i] - a [2] * y[i];
    }
}
```

Table 3 shows the parameters and their descriptions.

Table 3. Parameters

Parameter	Description
float *x	A pointer to the input signal. The input signal has a 32-bit float format. The number of elements in the vector is at least two.
float *b	A pointer to the normalized input multiply (present the zeros) filter coefficients; see the filter coefficients definition from the C model (see Equation 2). The biquad function requires three 32-bit float format b coefficients. Note that all filter coefficients were normalized so that a[0] (see next parameter) is always 1. If a[0] is not 1, then all coefficients are divided by the value of a[0].
float *a	A pointer to the normalized output multiply (present the poles) filter coefficients; see the filter coefficients definition from the C model (see Equation 2). The biquad function requires two 32-bit float coefficients for a[1] and a[2]. Note that a[0] is normalized to 1, so a[0] is not needed and thus it is not part of the a vector. Note that all filter coefficients were normalized to guarantee that a[0] is 1. If a[0] is not 1, then all coefficients are divided by the original value of a[0].
float *delay	When the biquad function starts, it has to know the two-history output values to calculate the first output value. These two-history values are stored in the two 32-bit float format vector delay. Note that at the conclusion of the execution of the routine, the last two outputs are stored back into the history delay vector to be used the next time the function is called. It is common practice to set the history values to zero the first time the function is called in an application.
float *y	A pointer to the output signal. The output signal has a 32-bit float format. The number of elements in the vector is at least two.
const int nx	The number of elements in the input and output vectors. Must be two or more.

4.1 Performances

Because of the dependency of the output on the previous output, the bottleneck of the performances is the loop-carrier dependency. Every iteration of the internal loop generates an output value in seven cycles. The number of operations is five multiplications and four additions. Measured performances for N elements is $7 \times N + 68$ cycles where code and data are in L2 memory, and L1D and L1P caches are enabled.

5 About FIR

The primary advantage of an FIR filter is that the output depends only on the input and the fixed set of coefficients, thus processing of multiple outputs in parallel is feasible. This is not the case for adaptive filter coefficients (adaptive filters are outside of the scope of this chapter). A typical API for an FIR filter is the following (void DSP_fir_gen):

```
const short *restrict x,    /* Input array [nr+nh-1 elements] */
const short *restrict h,   /* Coeff array [nh elements] */
short *restrict r,        /* Output array [nr elements] */
int nh,                  /* Number of coefficients */
int nr                   /* Number of output samples */
```

When initializing the input vector for the function, the input vector must have at least the number of the result elements plus the number of filter coefficients minus one (see Equation 3).

$$N_x \geq N_R + N_H - 1$$

where

- N_x is the number of elements in the input vector
- N_R is the number of result elements
- N_H is the number of filter coefficients

(3)

6 References

1. [Introducing TI's Integrated Development Environment – Code Composer Studio™ \(CCS\) to Expert Engineers](#), (SWRA526)

DSPLIB Documentation in Release

TI uses a CHM system to present information about optimized library routines. This short document details how to access the information. All screenshots and directory structures are taken from the DSPLIB that is part of Processor SDK RTOS release 2.0.2 for C667x devices (DSPLIB version in the release is 3.4.0.0). Other devices, other libraries, and other revisions of the library have a similar structure.

The DSPLIB library location is relative to where the user installed the Processor SDK RTOS release. View the following directory: `\dsplib_c66x_3_4_0_0\packages\ti\dsplib`.

The src directory contains all of the library functions. Each function has three types of source code: an optimized version of the function, a natural C model of the function used for debug purposes and to understand the model of the function, and a unit test. For example, for the function DSP_XXX, directory DSP_XXX/c66 (or whatever code is used) has the functions DSP_XXX.c (optimized code), DSP_XXX_cn.c (natural C model code), and DSP_XXX_d.c (unit test). In addition, a set of projects enables the user to build the library binaries based on little or bit endian, and the format of the binary (COFF or ELF). [Figure 1](#) is an image of the C66 directory structure that shows the projects that can be used to build libraries.

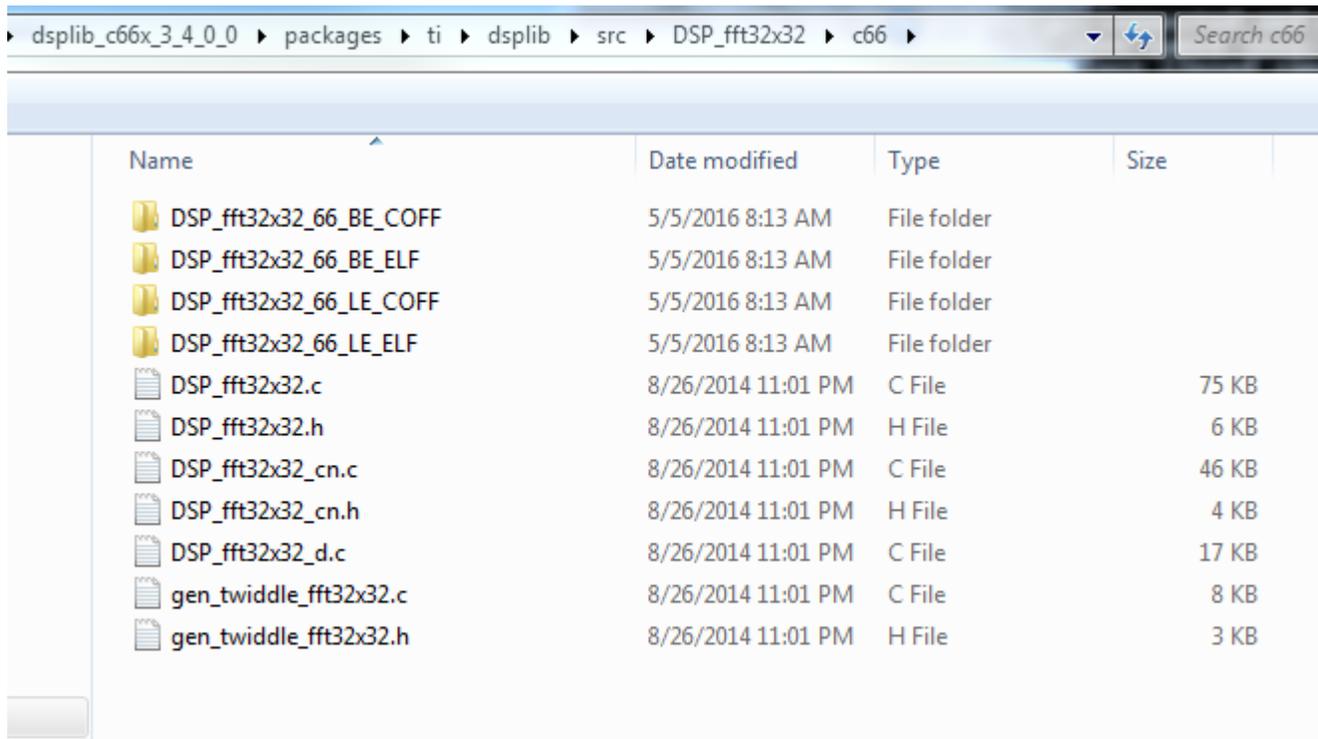


Figure 1. C66 Directory Structure

The documentation for all of the functions are part of the following directory: `\dsplib_c66x_3_4_0_0\packages\ti\dsplib\docs`.

Figure 2 shows the documents directory.

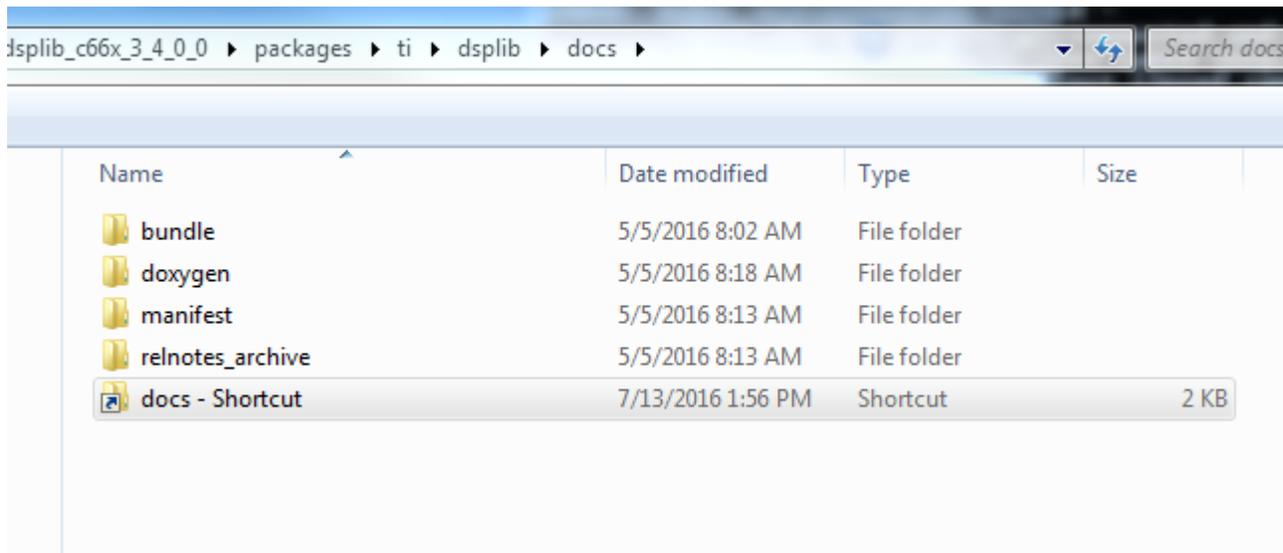


Figure 2. Documents Directory

Subdirectory doxygen has the DSPLIB.chm file. If the user does not have a CHM reader already installed, [List of CHM readers and viewers for Windows](#) provides a list of CHM readers on Windows® machines. The user should install one of the CHM readers. Figure 3 shows the doxygen directory.

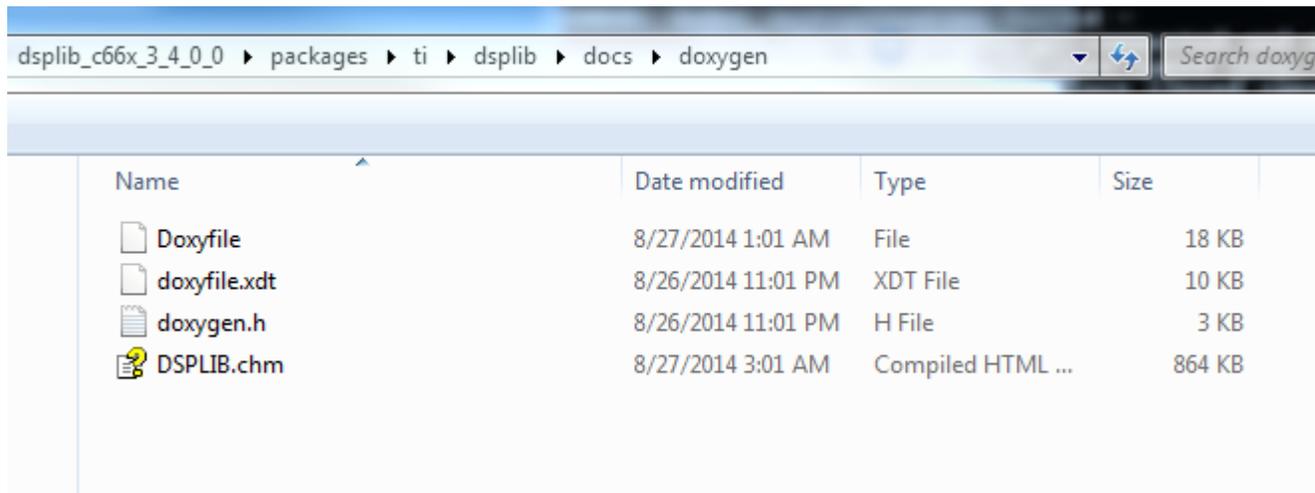


Figure 3. doxygen Directory

Double-clicking on DSPLIB.chm opens a small CHM introductory window (window size and placement may be different for different users).

Figure 4 shows the introductory window.

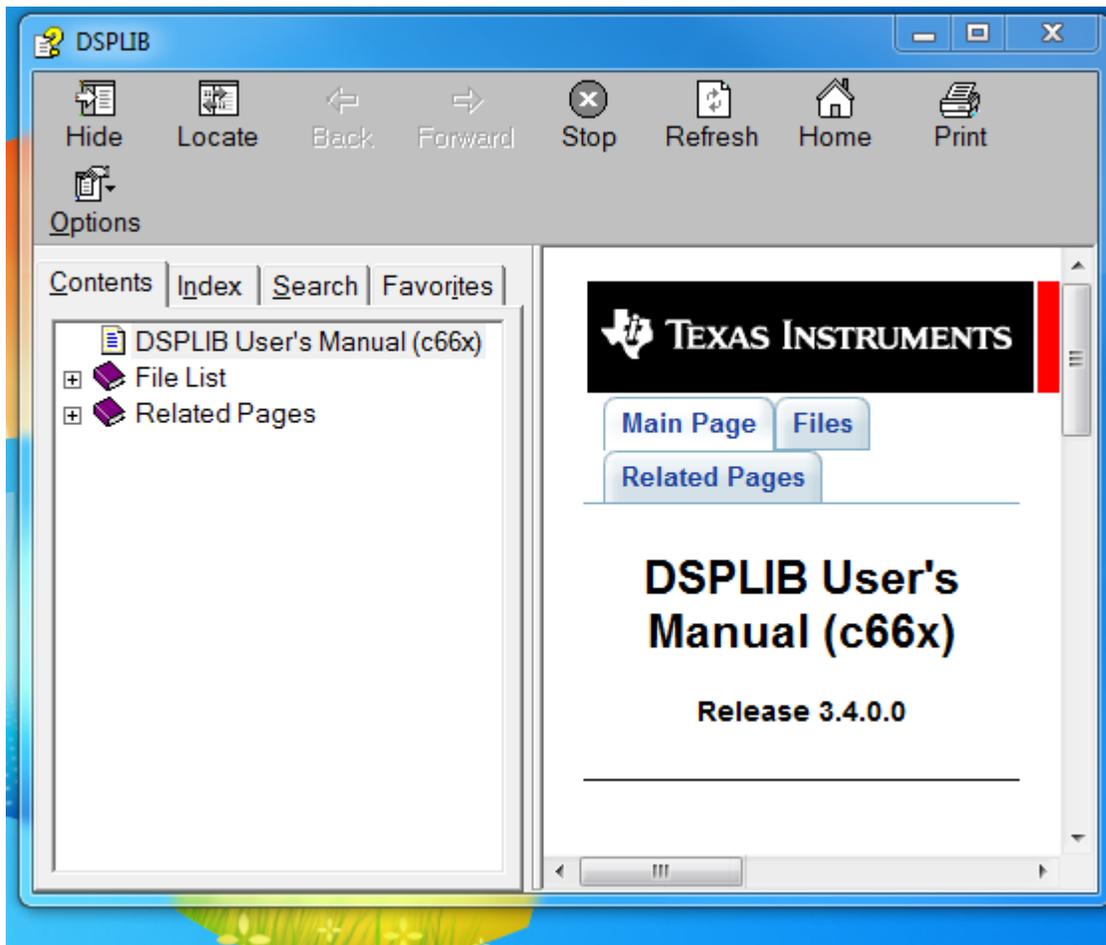
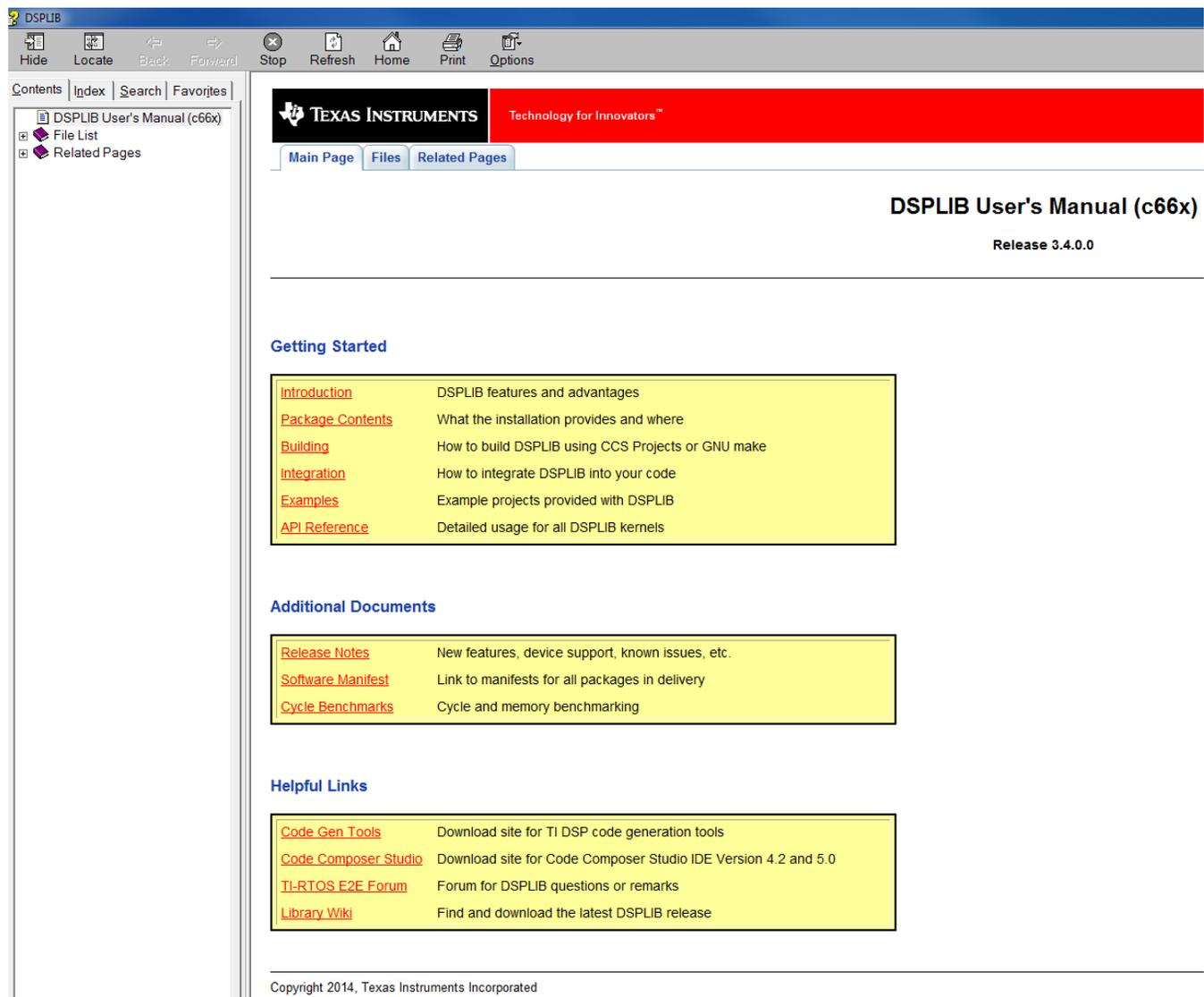


Figure 4. Introductory Window of DSPLIB.chm

If the image is not fullscreen, expand it so it looks like [Figure 5](#).



DSPLIB

Hide Locate Back Forward Stop Refresh Home Print Options

Contents Index Search Favorites

DSPLIB User's Manual (c66x)
File List
Related Pages

TEXAS INSTRUMENTS Technology for Innovators™

Main Page Files Related Pages

DSPLIB User's Manual (c66x)

Release 3.4.0.0

Getting Started

Introduction	DSPLIB features and advantages
Package Contents	What the installation provides and where
Building	How to build DSPLIB using CCS Projects or GNU make
Integration	How to integrate DSPLIB into your code
Examples	Example projects provided with DSPLIB
API Reference	Detailed usage for all DSPLIB kernels

Additional Documents

Release Notes	New features, device support, known issues, etc.
Software Manifest	Link to manifests for all packages in delivery
Cycle Benchmarks	Cycle and memory benchmarking

Helpful Links

Code Gen Tools	Download site for TI DSP code generation tools
Code Composer Studio	Download site for Code Composer Studio IDE Version 4.2 and 5.0
TI-RTOS E2E Forum	Forum for DSPLIB questions or remarks
Library Wiki	Find and download the latest DSPLIB release

Copyright 2014, Texas Instruments Incorporated

Figure 5. Fullscreen View of DSPLIB.chm

Double-click on the API Reference and a new window is opened with a list of DSPLIB MODULES (see Figure 6).

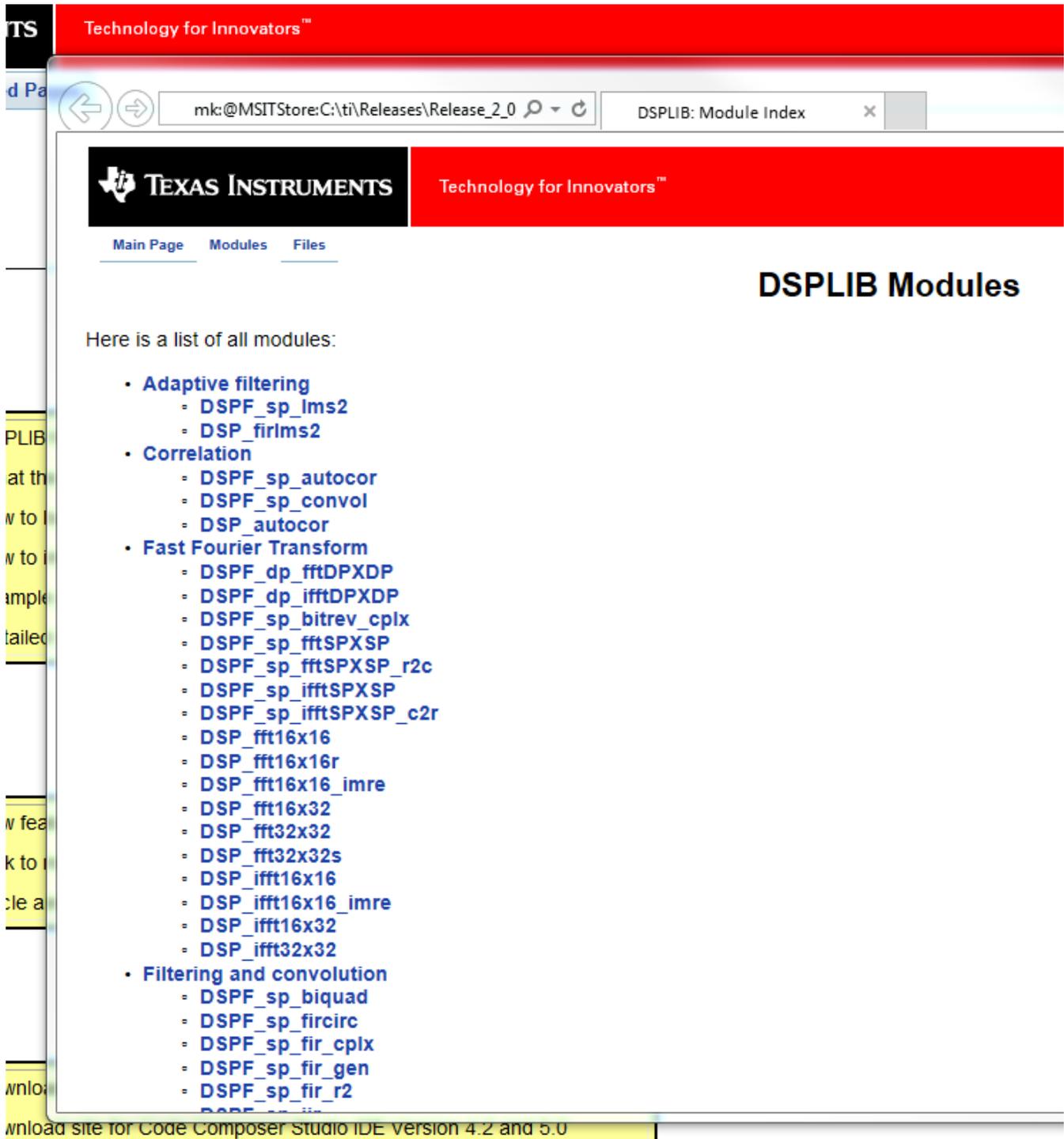
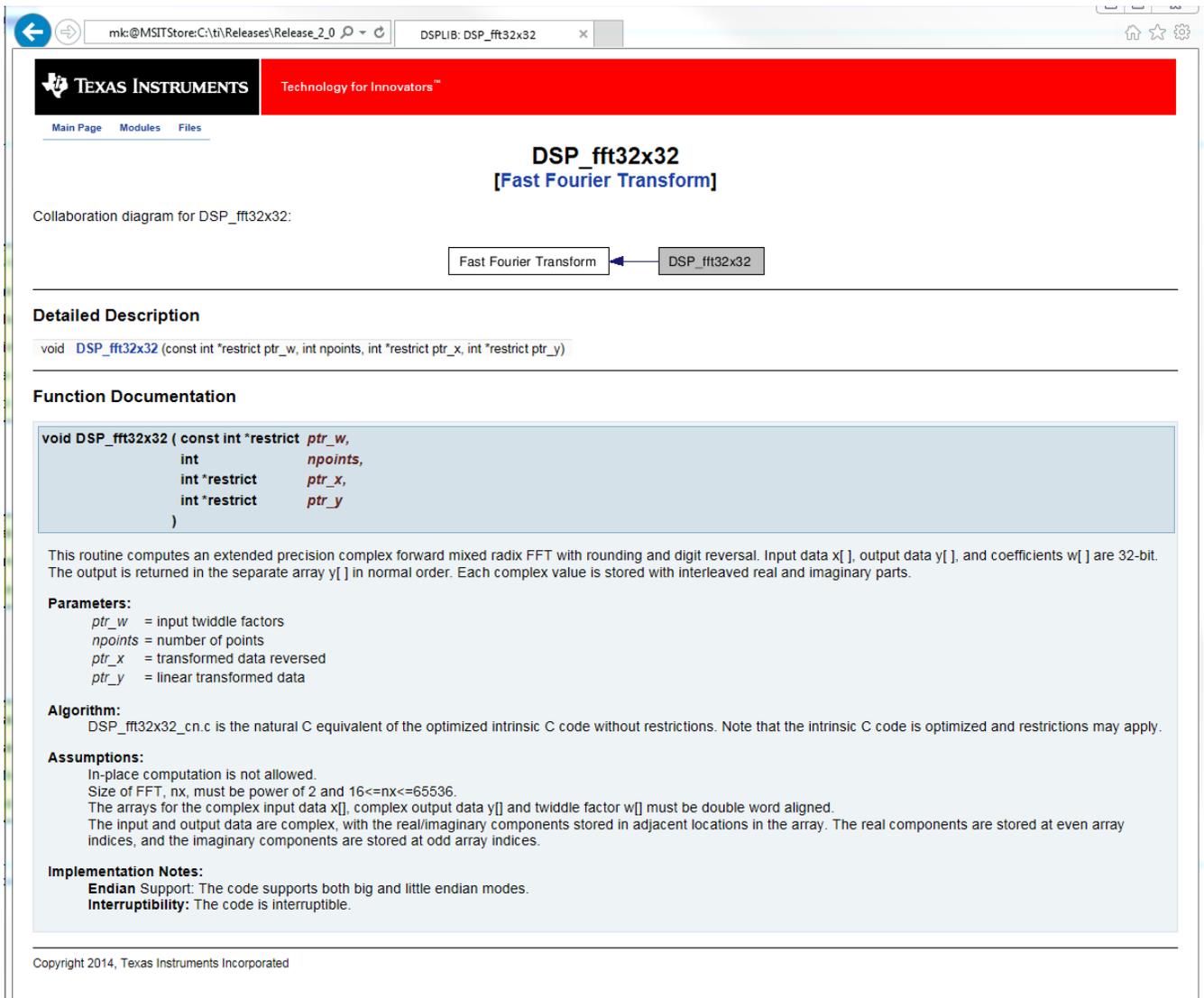


Figure 6. DSP Modules Window

Choose DSP_fft32x32 and double-click on it. A full page of detailed description is opened (see Figure 7).



DSP_fft32x32
[Fast Fourier Transform]

Collaboration diagram for DSP_fft32x32:

```

    graph LR
      FFT[Fast Fourier Transform] --> DSP[DSP_fft32x32]
  
```

Detailed Description

```
void DSP_fft32x32 (const int *restrict ptr_w, int npoints, int *restrict ptr_x, int *restrict ptr_y)
```

Function Documentation

```
void DSP_fft32x32 ( const int *restrict ptr_w,
                   int npoints,
                   int *restrict ptr_x,
                   int *restrict ptr_y
                   )
```

This routine computes an extended precision complex forward mixed radix FFT with rounding and digit reversal. Input data x[], output data y[], and coefficients w[] are 32-bit. The output is returned in the separate array y[] in normal order. Each complex value is stored with interleaved real and imaginary parts.

Parameters:

- ptr_w = input twiddle factors
- npoints = number of points
- ptr_x = transformed data reversed
- ptr_y = linear transformed data

Algorithm:

DSP_fft32x32_cn.c is the natural C equivalent of the optimized intrinsic C code without restrictions. Note that the intrinsic C code is optimized and restrictions may apply.

Assumptions:

- In-place computation is not allowed.
- Size of FFT, nx, must be power of 2 and 16<=nx<=65536.
- The arrays for the complex input data x[], complex output data y[] and twiddle factor w[] must be double word aligned.
- The input and output data are complex, with the real/imaginary components stored in adjacent locations in the array. The real components are stored at even array indices, and the imaginary components are stored at odd array indices.

Implementation Notes:

- Endian Support:** The code supports both big and little endian modes.
- Interruptibility:** The code is interruptible.

Copyright 2014, Texas Instruments Incorporated

Figure 7. DSP_fft32x32 Window

If the documentation is not clear enough, read the natural C model code to understand what the function is doing.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as "components") are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

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

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

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license 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 significant portions 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. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

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

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or "enhanced plastic" are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have **not** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products

Audio	www.ti.com/audio
Amplifiers	amplifier.ti.com
Data Converters	dataconverter.ti.com
DLP® Products	www.dlp.com
DSP	dsp.ti.com
Clocks and Timers	www.ti.com/clocks
Interface	interface.ti.com
Logic	logic.ti.com
Power Mgmt	power.ti.com
Microcontrollers	microcontroller.ti.com
RFID	www.ti-rfid.com
OMAP Applications Processors	www.ti.com/omap
Wireless Connectivity	www.ti.com/wirelessconnectivity

Applications

Automotive and Transportation	www.ti.com/automotive
Communications and Telecom	www.ti.com/communications
Computers and Peripherals	www.ti.com/computers
Consumer Electronics	www.ti.com/consumer-apps
Energy and Lighting	www.ti.com/energy
Industrial	www.ti.com/industrial
Medical	www.ti.com/medical
Security	www.ti.com/security
Space, Avionics and Defense	www.ti.com/space-avionics-defense
Video and Imaging	www.ti.com/video

TI E2E Community

e2e.ti.com