FPU DSP Software Library

USER'S GUIDE

Copyright

Texas Instruments
12203 Southwest Freeway
Houston, TX 77477
http://www.ti.com/c2000

Revision Information

This is version V2.11.00.00 of this document, last updated on Apr 7, 2024.

Table of Contents

Copyright . . . 2
Revision Information . . . 2
1 Introduction . . . 5
2 Other Resources . . . 6
3 Library Structure . . . 7
3.1 Header Files . . . 8
3.2 Source Files . . . 9
4 Using the FPU Library . . . 10
4.1 Library Build Configurations . . . 10
4.2 Integrating the Library into your Project . . . 12
4.3 Dependencies on Fast RTS library & ROM Tables . . . 19
5 Application Programming Interface (FPU32) . . . 21
5.1 Introduction to the Single Precision DSP Library API . . . 21
5.2 DSP Library Definitions and Types . . . 25
5.3 Single Precision DSP Library Data Types . . . 27
5.4 Complex Fast Fourier Transforms . . . 28
5.5 Real FFT Using a Complex FFT . . . 45
5.6 Real Fast Fourier Transforms . . . 46
5.7 Filters . . . 60
5.8 Vector Operations . . . 73
6 Application Programming Interface (FPU64) . . . 90
6.1 Introduction to the Double Precision DSP Library API . . . 90
6.2 DSP Library Definitions and Types . . . 93
6.3 Fast Fourier Transform (Double Precision) . . . 95
6.4 Real FFT Using a Complex FFT . . . 117
6.5 Filters (Double Precision) . . . 118
6.6 Vector Operations (Double Precision) . . . 129
7 Benchmarks . . . 144
8 Revision History . . . 155
IMPORTANT NOTICE . . . 159

64F2838x,F28P65x 32F28002x, F28003x, F280013x, F280015x, F28P55x, and F28P65x

1 Introduction

The Texas Instruments TMS320C28x Floating Point Unit Digital Signal Processing (FPU DSP) Library is a collection of optimized signal processing routines written for C2000 devices that support either a single precision Floating Point Unit (FPU32), an FPU32 with Triginometric Math Unit (TMU type 0), or a double precision FPU (FPU64).

These functions enable C/C++ programmers to take full advantage of the aforementioned hardware accelerators to speed up computation time. This document provides a description of each function included in the library.

chapter 2 provides a host of resources on the FPU 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 single precision routines, with their accompanying variables, data types and structures.

chapter 6 describes the double precision routines, with their accompanying variables, data types and structures.

chapter 7 lists the performance of each routine.

chapter 8 provides a revision history of the library.

Examples have been provided for each library routine. They can be found in the *examples* directory. For the current revision, the newest examples using FPU64 have been written and validated on the device based *controlCard*. All FPU32 based examples have been validated on based *controlCard*.

2 Other Resources

The user can refer to the , TRMs for specific details.

Also check out the TI C2000 portfolio at
http://www.ti.com/microcontrollers/c2000-real-time-control-mcus/overview.html

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

Building the FPU libraries and examples require **Codegen Tools v22.6.0.LTS** or later.

3 Library Structure

Header Files 8
Source Files 9

By default, the library and source code is installed into the c2000ware directory under the sub-folder

```
C:\ti\c2000\c2000ware_<version>\libraries\dsp\FPU
```

Figure. 3.1 shows the directory structure while the subsequent table 3 provides a description for each folder.

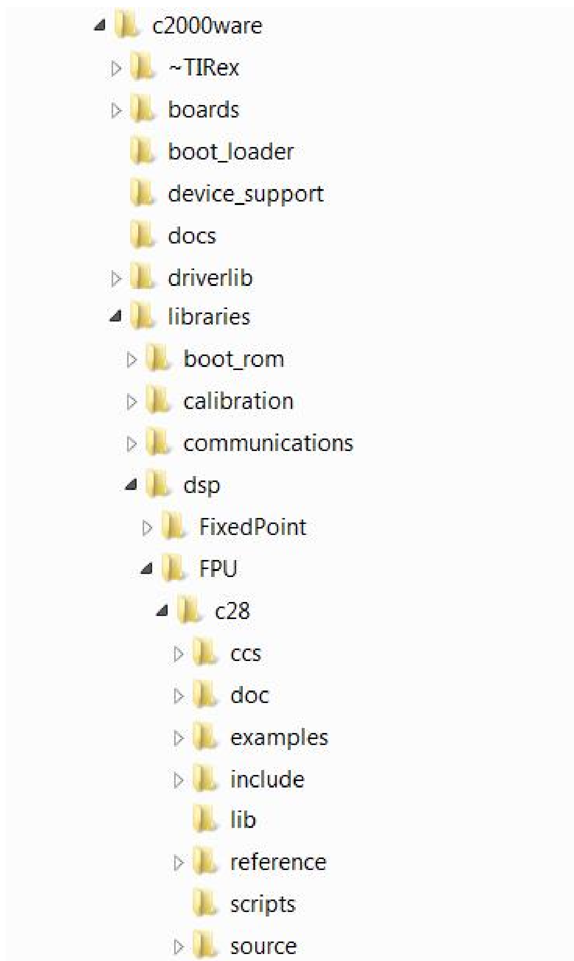

Figure 3.1: Directory Structure of the FPU Library

Folder	Description
<base>	Base install directory. By default this is C:/ti/c2000/c2000ware_5_01_00_00/libraries/dsp/FPU. 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>/doc	Documentation for the current revision of the library including revision history.
<base>/examplesFolder	FPU32 examples were tested and validated on *32* control cards. FPU64 examples were validated on *64*. All examples were validated using CCS version 12.
<base>/examplesFolder/common	Device specific setup, linker command files and ROM symbol libraries for the examples.
<base>/include/fpu32	Header files for the single precision floating point DSP library. These include function prototypes and structure definitions.
<base>/include/fpu64	Header files for the double precision floating point DSP library. These include function prototypes and structure definitions.
<base>/lib	Static libraries (single and double precision).
<base>/source/fpu32	Source files for the single precision floating point DSP library.
<base>/source/fpu64	Source files for the double precision floating point DSP library.

Table 3.1: FPU DSP Library Directory Structure Description

3.1 Header Files

The header files are sorted into two folders under *"include"*

- **fpu32**, single precision library header files
- **fpu64**, double precision library header files

The file *"dsp.h"* is common to both libraries and defines new data types and macros.

3.2 Source Files

The source files are sorted into two folders under *"source"*

- **fpu32** - single precision library header files
- **fpu64** - double precision library header files

Within each folder the source code is further split into modules

- **fft**, Fast Fourier Transform Module
- **filter**, Filter (FIR and IIR) Module
- **utility**, Utility Module
- **vector**, Vector Module

4 Using the FPU Library

Library Build Configurations . . . 10
Integrating the Library into your Project . . . 12
Dependencies on Fast RTS library ROM Tables . . . 19

The source code and project(s) for the FPU libraries are provided. The user may import the library project(s) into CCSv10 or later and be able to view and modify the source code for all routines. (see Figure 4.1)

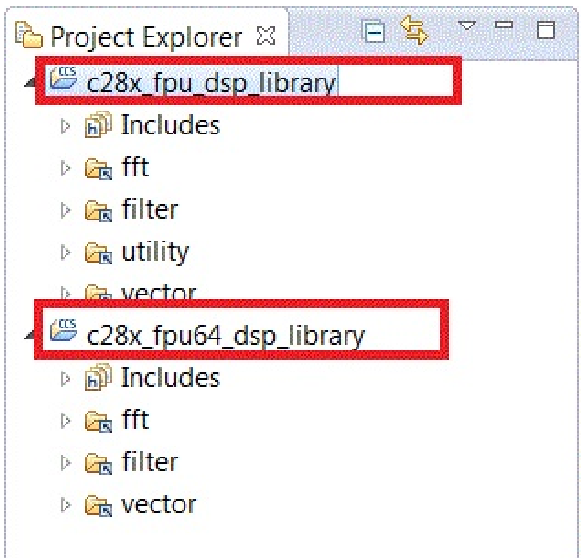

Figure 4.1: FPU Library Project View

4.1 Library Build Configurations

The single precision library has both COFF and EABI build configurations i.e. **ISA_C28FPU32**, **ISA_C28FPU32_EABI** (Figure 4.2) while the double precision library has only EABI configuration **ISA_C28FPU64** (Figure 4.3).

The **ISA_C28FPU32** and **ISA_C28FPU32_EABI** configurations are built with the **–float_support=fpu32** and **–tmu_support=tmu0** run-time support options enabled. Running a build on these configuration will generate **c28x_fpu_dsp_library_coff.lib** and **c28x_fpu_dsp_library_eabi.lib** in the lib folder. An **index library c28x_fpu_dsp_library.lib** is created using **libinfo2000** tool that can be linked against instead of directly linking to a coff or eabi-specific library. Thus any example irrespective of COFF/EABI can just link to this index library, the linker then uses the index library to automatically choose the appropriate version of the library to use based on the build attribute of the particular example. Some of the original routines have alternate versions that can make use of the TMU accelerator's (on devices that have it) ability to speed up certain trigonometric and math operations.

For devices that have a Floating Point Unit (FPU), but no Trigonometric Math Unit (TMU), the user will not be able to use the TMU0 variants of some functions.

NOTE: ATTEMPTING TO LINK IN THIS LIBRARY INTO A PROJECT THAT DOES NOT HAVE THE FLOAT_SUPPORT SET TO FPU32 WILL RESULT IN A COMPILER ERROR ABOUT MISMATCHING INSTRUCTION SET ARCHITECTURES

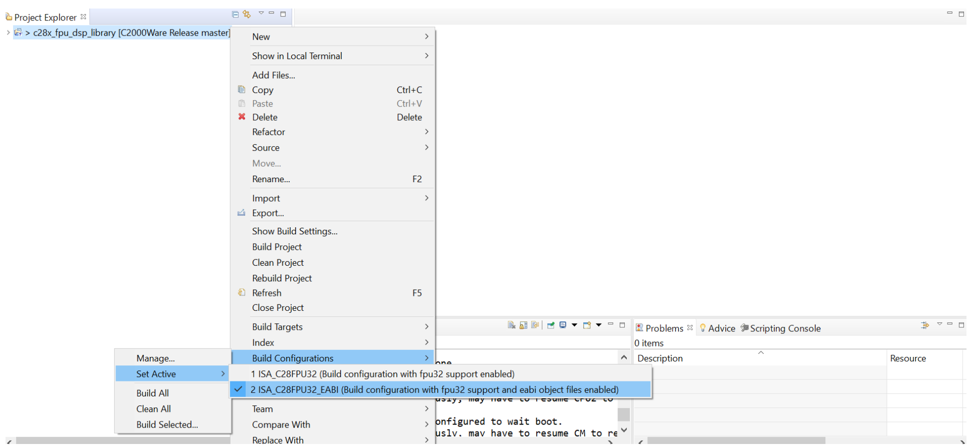

Figure 4.2: Single Precision Library Build Configuration

The **ISA_C28FPU64** configuration is built with the **–float_support=fpu64** run-time support option enabled. Running a build on this configuration will generate **c28x_fpu64_dsp_library.lib** in the lib folder.

NOTE: ATTEMPTING TO LINK IN THIS LIBRARY INTO A PROJECT THAT DOES NOT HAVE THE FLOAT_SUPPORT SET TO FPU64 WILL RESULT IN A COMPILER ERROR ABOUT MISMATCHING INSTRUCTION SET ARCHITECTURES

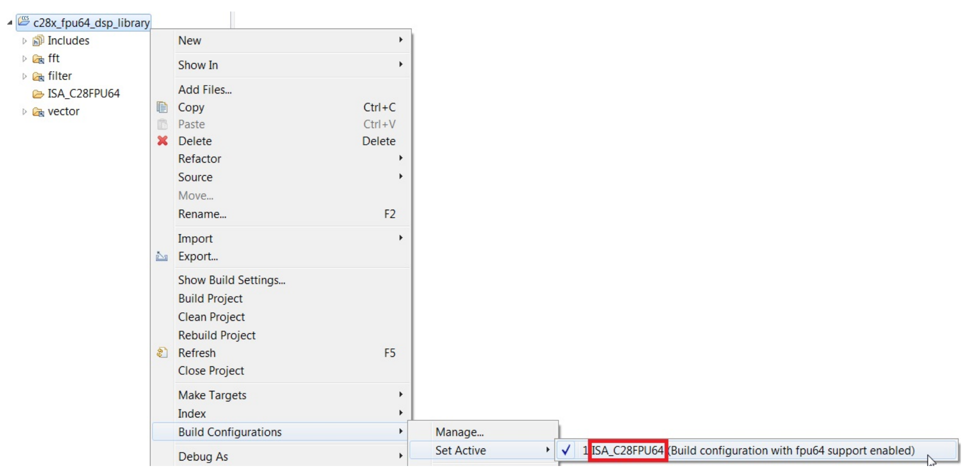

Figure 4.3: Double Precision Library Build Configuration

The table below summarizes all the build configurations and provides description of all single precision and double precision FPU libraries included in the folder.

Floating-Point precision	Library Name	Library Format	Build Configuration	Description
32-bit	c28x_fpu_dsp_library_coff.lib	COFF	ISA_C28FPU32	Single precision COFF FPUDSP library
	c28x_fpu_dsp_library_eabi.lib	EABI	ISA_C28FPU32_EABI	Single precision EABI FPUDSP library
	c28x_fpu_dsp_library.lib	-	-	Index library for above single precision COFF and EABI variants, this needs to be used in project's linker options
64-bit	c28x_fpu64_dsp_library.lib	EABI	ISA_C28FPU64	Double Precision FPUDSP library

Table 4.1: Build configurations and library description

NOTE: THE C2000 COMPILER WILL ONLY SUPPORT 64-BIT FLOATING-POINT INSTRUCTIONS IN EABI FORMAT, THAT IS WHY NO DOUBLE PRECISION COFF LIBRARY IS BEING PROVIDED. ANY PROJECT REQUIRING 64-BIT FLOATING-POINT SUPPORT MUST USE EABI. ALSO ÂĂIJFPUÂĂİ AND ÂĂIJFPU32ÂĂİ ARE EQUIVALENT IN THE LIBRARY NAMING TO BE CONSISTENT WITH THE PREVIOUSLY RELEASED LIBRARY NAME. FOR MORE INFORMATION, PLEASE REFER TO TMS320C28X OPTIMIZING C/C++ COMPILER USERÂĂŹS GUIDE WWW.TI.COM/LIT/SPRU514

NOTE: DO TAKE CARE WHILE ADDING/COPYING THE SINGLE PRECISION DSP LIBRARIES TO THE CCS PROJECTS I.E. ALL THE VERSIONS (COFF, EABI) OF THE LIBRARY INCLUDING THE INDEX LIBRARY NEED TO COPIED TO THE PROJECTS SO AS TO ALLOW THE LINKER TO PICK THE CORRECT VERSION OF THE LIBRARY. THIS STEP IS NOT REQUIRED IF THE LIBRARIES ARE BEING LINKED THROUGH PROJECT BUILD OPTIONS. THUS THE RECOMMENDED PRACTICE IS TO ALWAYS LINK THE DSP LIBRARIES INSTEAD OF COPYING.

4.2 Integrating the Library into your Project

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

1. Select the Run Time Support Library option to **automatic** while building your first single precision and double precision example project (see Figure 4.4). This will automatically generate the run time support libraries for FPU inside the compiler folder. Once the libraries are generated, then try to build any of the examples, the build should succeed.

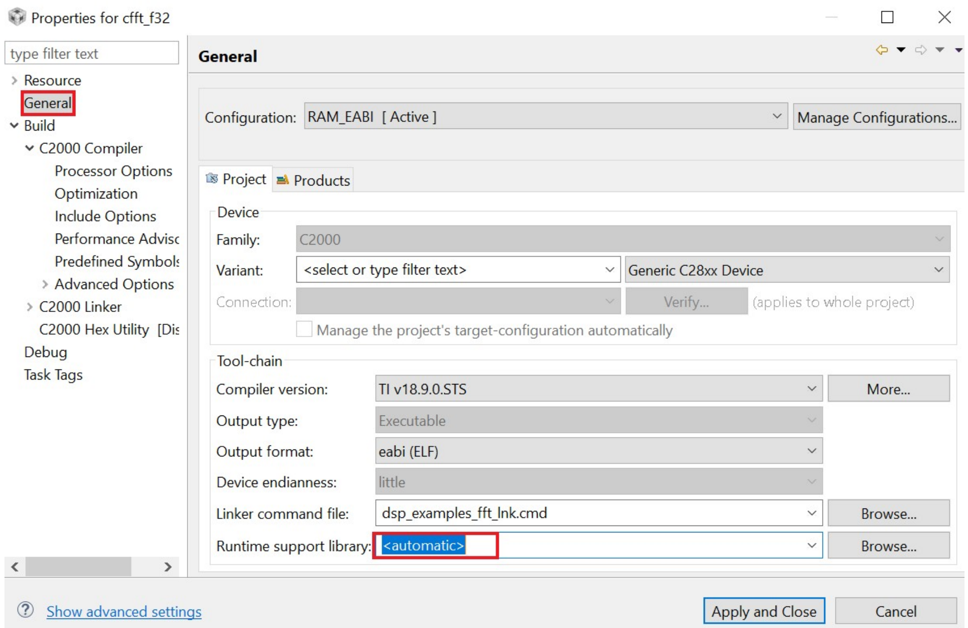

Figure 4.4: Generating run time support libraries inside compiler folder

2. Go to the **Project Properties->Resources->Linked Resources->Path Variables Tab** and add a new variable (see Figure 4.5), **FPU_DSP_LIB_ROOT**, and set its path to the root directory of the floating point DSP library in c2000ware; this is usually the **c28** folder.

   Additionally, you may want to set a variable that point to the Fast RTS Math Library (if using FastRTS enabled functions), **FASTRTS_MATH_LIB_ROOT**. This library is required in the computation of the FFT spectrum phase.
   Each example project requires two variables that are device specific,

   - **DRIVERLIB_ROOT**, points to the driver library of the target device on which the code will run
   - **DSP_EXAMPLES_COMMON**, points to the target specific device under the folder **c28/examples/common/**<**device**>; this folder contains the linker command files.
   - **DSP_EXAMPLES_COMMONSRC**, contains basic system setup code used in each of the examples; they can be customized to meet the user's needs.

   When switching to another compatible device, i.e. one having a hardware floating point unit, change the foregoing variables with the appropriate target device name; there must be a sub-folder in the *"common"* directory with the target device name.

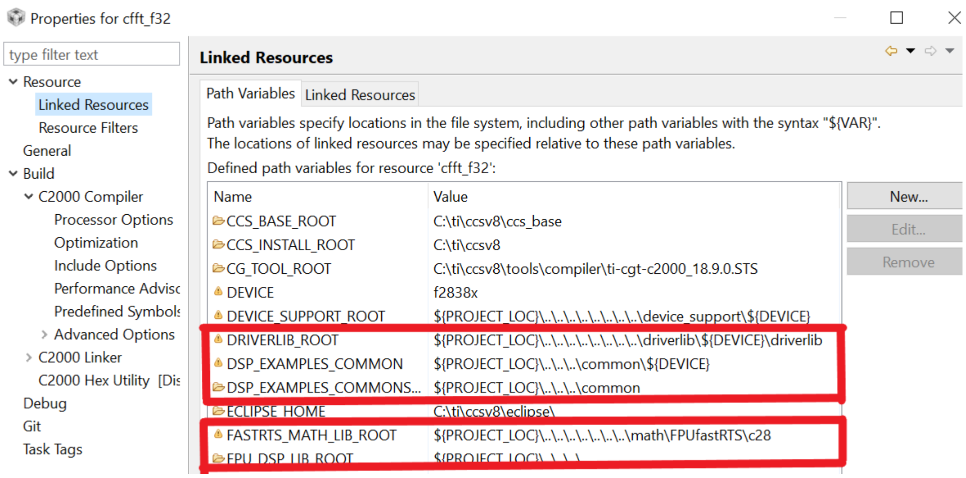

Figure 4.5: Creating a new build variable

Add the new path, **FPU_DSP_LIB_ROOT/include**, to the *Include Options* section of the project properties (Figure 4.6). This option tells the compiler where to find the library header files. In addition, you must add the driver library (**DRIVERLIB_ROOT**)path as well as the common folder (**DSP_EXAMPLES_COMMON**) and (**DSP_EXAMPLES_COMMONSRC**) path for the target device in use.

There are some functions like phase which require the arc-tangent function. The call can either be handled by the standard C Math library (more accurate but slower) or the FastRTS library (faster, less accurate). If the user decides to use the FastRTS library instead of the standard C math library, they must add the search path,

```
${FASTRTS_MATH_LIB_ROOT}/include/fpu32
```

for single precision examples and,

```
${FASTRTS_MATH_LIB_ROOT}/include/fpu64
```

for double precision examples, to the project properties.

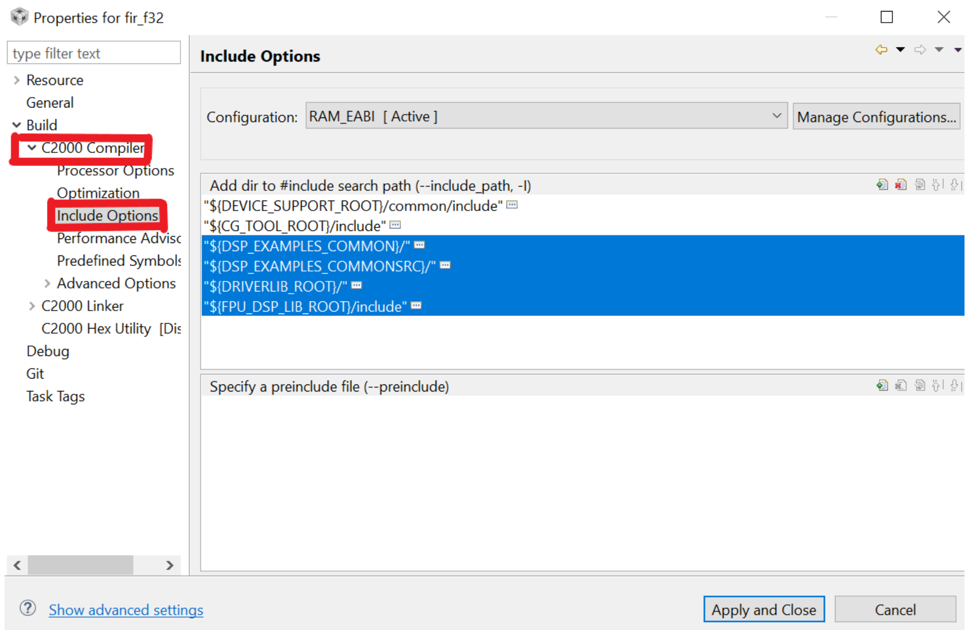

Figure 4.6: Adding the Library Header Path to the Include Options

3. For the single prcecision library, first set active **RAM_EABI** and set the **–float_support** option to fpu32 in the **Runtime Model Options** (Figure 4.7).

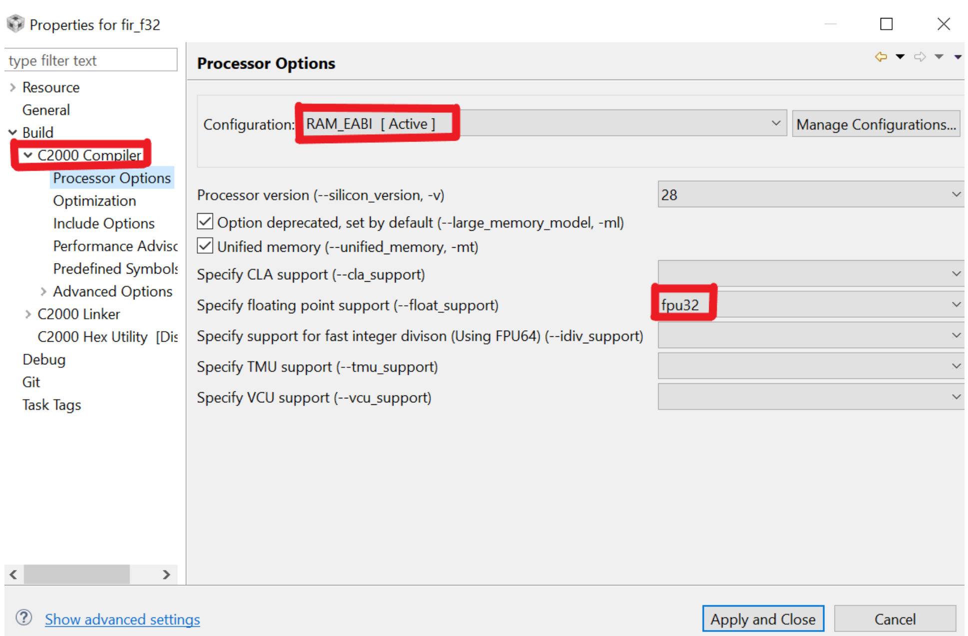

Figure 4.7: Turning on FPU32 support

Additionally, add the **tmu_support** option to the compiler command line if the user wishes to use the TMU0 function variants, and if the device supports it (Figure 4.8).

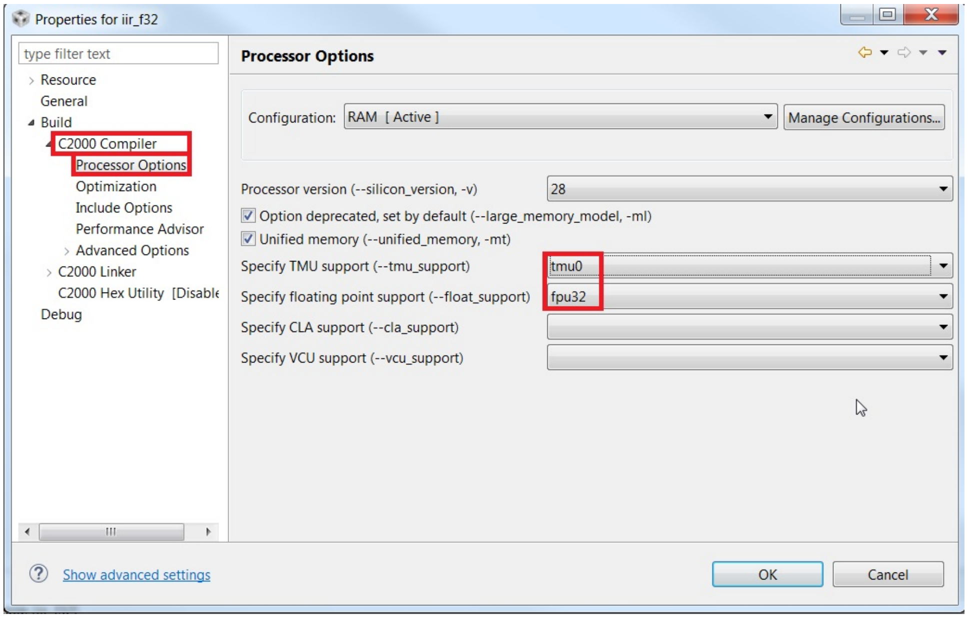

Figure 4.8: Turning on TMU support

4. For the double prcecision library enable the fpu64 support as shown in Figure 4.9.

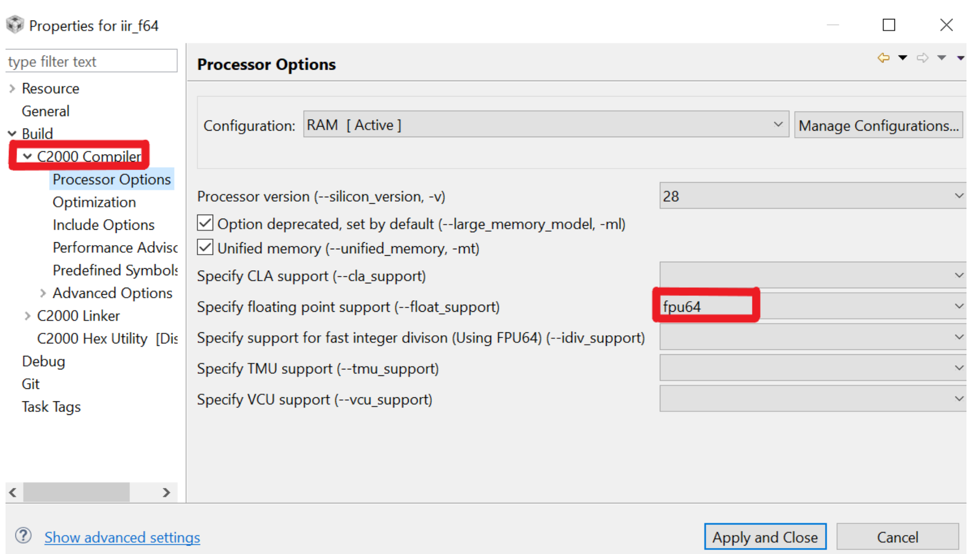

Figure 4.9: Turning on FPU64 support

5. Add the name of the library and its location to the **File Search Path** as shown in Figure 4.10 and Figure 4.11. For the single precision DSP library add **c28x_fpu_dsp_library.lib**, and **c28x_fpu64_dsp_library.lib** for the double precision library.

   NOTE: BE SURE TO ENABLE FLOAT_SUPPORT (AND, OPTIONALLY, TMU_SUPPORT IF THE DEVICE SUPPORTS IT) IN YOUR PROJECT PROPERTIES

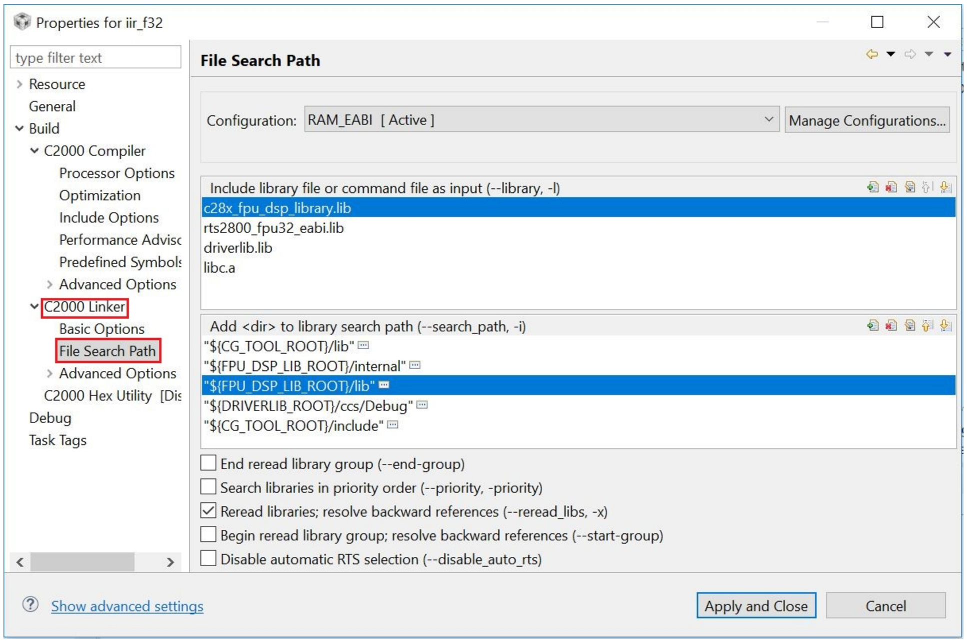

Figure 4.10: Adding the library and location to the file search path

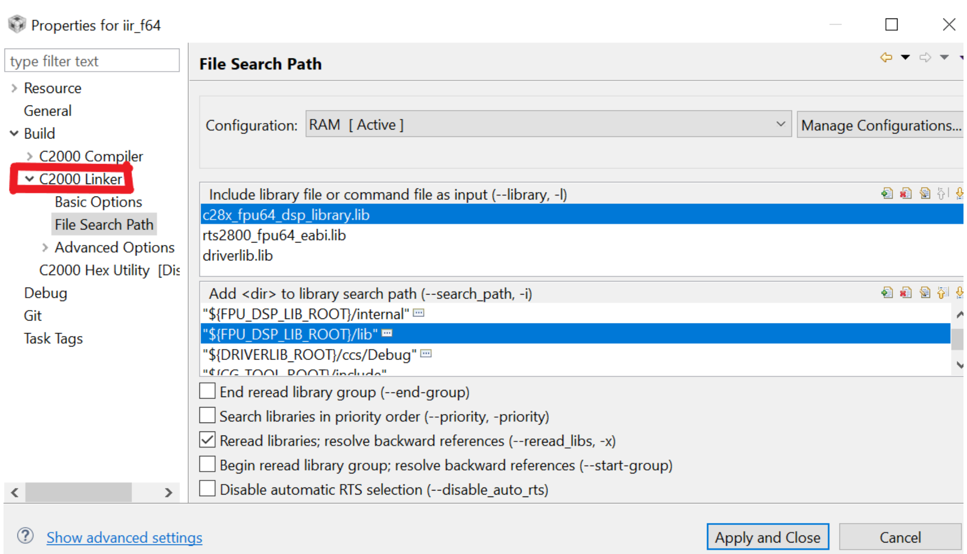

Figure 4.11: Adding the library and location to the file search path

6. When the user enables the TMU support option, the compiler automatically defines the macro __TMS320C28XX_TMU__ . It can be used to switch between TMU and non-TMU variants of the functions in the library. For example, the magnitude function has two variants, **CFFT_f32_mag** and **CFFT_f32_mag_TMU0**, the user can use the compiler defined macro to swtich between them, as follows

```
#ifdef __TMS320C28XX_TMU__ //defined when --tmu_support=tmu0 in the project
                           // properties
```

```
    // Calculate magnitude, result stored in CurrentOutPtr
    CFFT_f32_mag_TMU0(hnd_cfft);
#else
    // Calculate magnitude, result stored in CurrentOutPtr
    CFFT_f32_mag(hnd_cfft);
#endif
```

7. By default, all single precision examples are on F28002x.In order to run FPU32 examples on , change the "DEVICE" strings to "f2838x" in both properties -> build -> Variables and properties -> Resources -> Linked Resources as shown in fig:device$_1 andfig : device_2$.

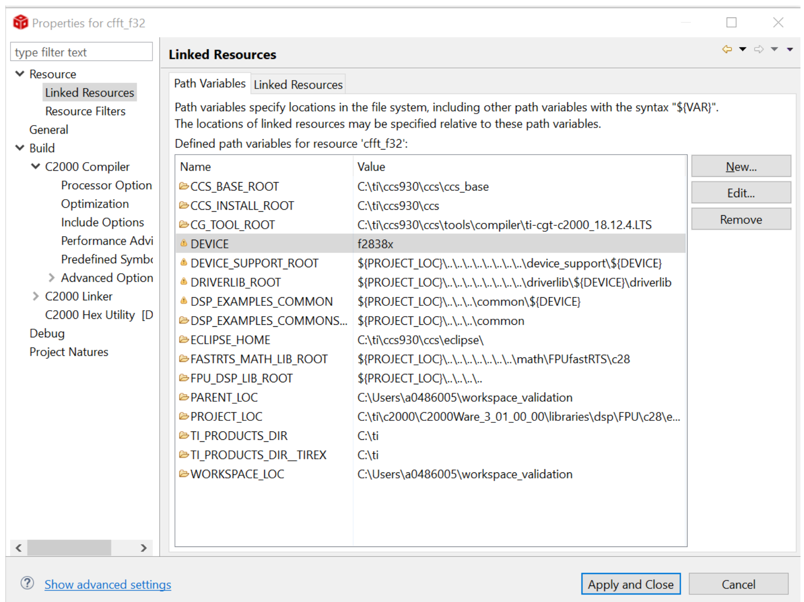

Figure 4.12: Changing Device Build Variable

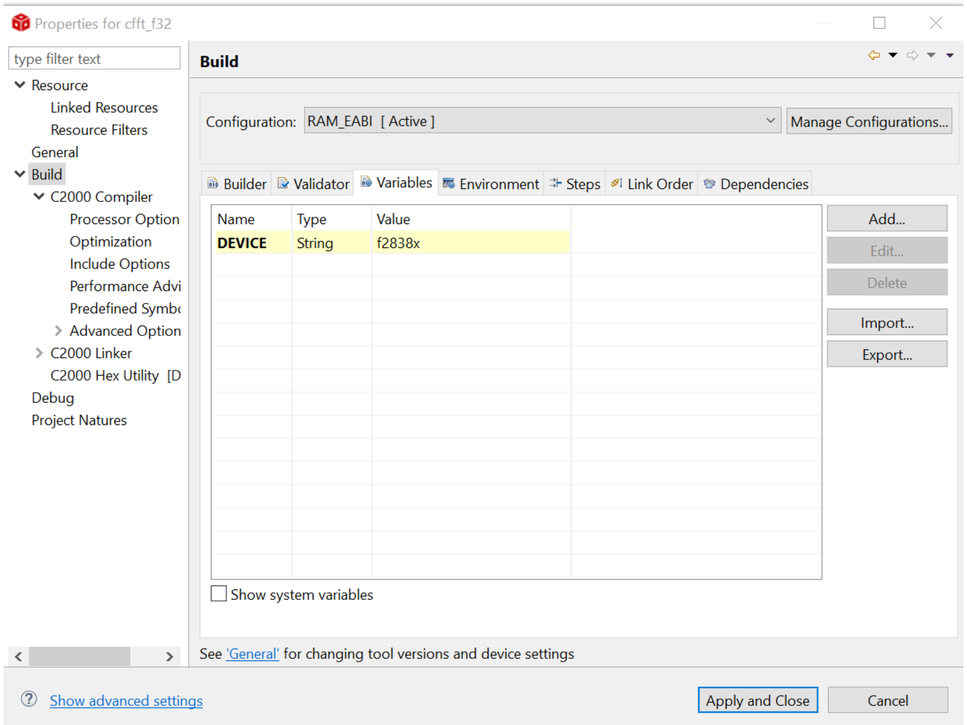

Figure 4.13: Changing Linked Resources Variable

4.3 Dependencies on Fast RTS library & ROM Tables

The FFT tables for both the single and double precision libraries are usually present in ROM of the target device (check target device TRM to determine which tables have been placed in ROM), and can be used by the FFT routines; this will save the user from having to either create the tables at runtime or load the table tables to FLASH. Thus all the FFT examples are provided with **RAM_ROMTABLES** configuration (along with RAM and FLASH) to make use of pre-stored FFT tables in ROM. First set active **RAM_ROMTABLES** configuration and include the ROMsymbols library as shown in Figure 4.14.

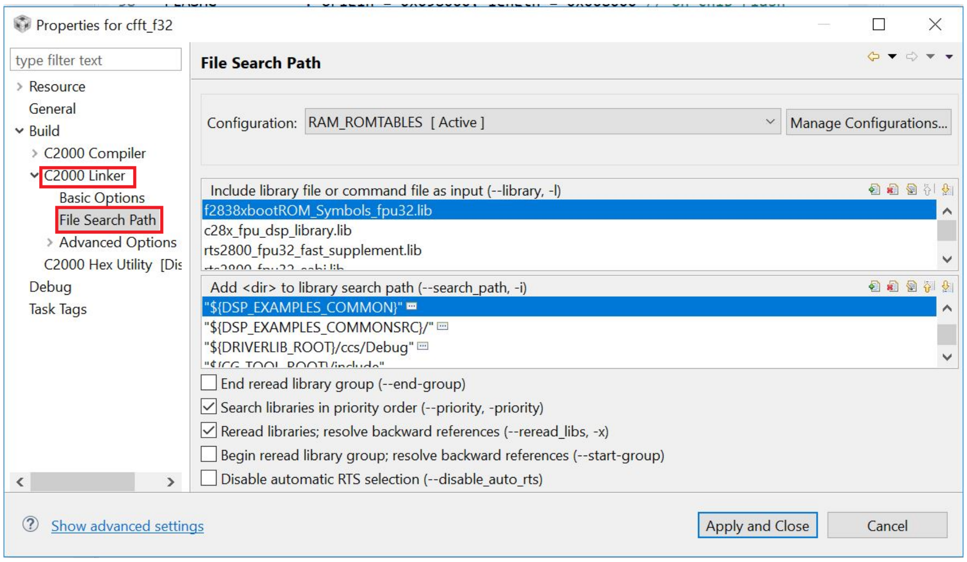

Figure 4.14: Including the bootROM Symbols library to use FFT tables stored in ROM

If using the FastRTS math functions, include the FastRTS library in the search path,

```
${FASTRTS_MATH_LIB_ROOT}/lib
```

and the name of the library in the *--library* box. For the single precision Fast RTS library add **rts2800_fpu32_fast_supplement.lib**, and **rts2800_fpu64_fast_supplement.lib** for the double precision library.

The linker searches libraries in priority order (when the **--priority** box is checked) to find the referenced function; in order to pull in the math routines from the Fast RTS library as opposed to the standard run-time support library the Fast RTS library must be placed above the standard RTS.

5 Application Programming Interface (FPU32)

5.1 Introduction to the Single Precision DSP Library API

The source code for the single precision library can be found under source/fpu32. They are organized into the following modules:

FFT:
The Fast Fourier Transform module contains routines to run both the complex and real FFT, the inverse FFT, generate the twiddle factor tables, “pack” and “unpack” the complex spectrum allowing for an alternative way to compute the Inverse and Forward Real FFT respectively. This module also contains pre-generated twiddle factor tables that can be loaded into memory at load time obviating the need to compute these factors at run time. Please note that the tolerance values of the "windowed FFT" examples are subject to change based on the input vector size.

FFT Module	
CFFT_f32	void CFFT_f32(CFFT_F32_STRUCT *);
CFFT_f32t	void CFFT_f32t(CFFT_F32_STRUCT *);
CFFT_f32i	void CFFT_f32i(CFFT_F32_STRUCT *);
CFFT_f32it	void CFFT_f32it(CFFT_F32_STRUCT *);
CFFT_f32u	void CFFT_f32u(CFFT_F32_STRUCT *);
CFFT_f32ut	void CFFT_f32ut(CFFT_F32_STRUCT *);
CFFT_f32_mag	void CFFT_f32_mag(CFFT_F32_STRUCT *);
CFFT_f32_mag_TMU0	void CFFT_f32_mag_TMU0(CFFT_F32_STRUCT *);
CFFT_f32s_mag	void CFFT_f32s_mag(CFFT_F32_STRUCT *);
CFFT_f32s_mag_TMU0	void CFFT_f32s_mag_TMU0(CFFT_F32_STRUCT *);
CFFT_f32_pack	void CFFT_f32_pack(CFFT_F32_STRUCT *);
CFFT_f32_phase	void CFFT_f32_phase(CFFT_F32_STRUCT *);
CFFT_f32_phase_TMU0	void CFFT_f32_phase_TMU0(CFFT_F32_STRUCT *);
CFFT_f32_sincostable	void CFFT_f32_sincostable(CFFT_F32_STRUCT *);
CFFT_f32_unpack	void CFFT_f32_unpack(CFFT_F32_STRUCT *);
CFFT32_f32_win	void CFFT32_f32_win(float *, float *, uint16_t);
CFFT32_f32_win_dual	void CFFT32_f32_win_dual(float *, float *, uint16_t);
ICFFT_f32	void ICFFT_f32(CFFT_F32_STRUCT *);
ICFFT_f32t	void ICFFT_f32t(CFFT_F32_STRUCT *);
RFFT_f32	void RFFT_f32(RFFT_F32_STRUCT *);
RFFT_f32u	void RFFT_f32u(RFFT_F32_STRUCT *);
RFFT_adc_f32	void RFFT_adc_f32(RFFT_ADC_F32_STRUCT *);
RFFT_adc_f32u	void RFFT_adc_f32u(RFFT_ADC_F32_STRUCT *);
RFFT_adc_f32_win	void RFFT_adc_f32_win(RFFT_ADC_F32_STRUCT *);
RFFT_f32_mag	void RFFT_f32_mag(RFFT_F32_STRUCT *);
RFFT_f32_mag_TMU0	void RFFT_f32_mag_TMU0(RFFT_F32_STRUCT *);
RFFT_f32s_mag	void RFFT_f32s_mag(RFFT_F32_STRUCT *);
RFFT_f32s_mag_TMU0	void RFFT_f32s_mag_TMU0(RFFT_F32_STRUCT *);
RFFT_f32_phase	void RFFT_f32_phase(RFFT_F32_STRUCT *);
RFFT_f32_phase_TMU0	void RFFT_f32_phase_TMU0(RFFT_F32_STRUCT *);
RFFT_f32_sincostable	void RFFT_f32_sincostable(RFFT_F32_STRUCT *);
RFFT_f32_win	void RFFT_f32_win(float *, float *, uint16_t);

Table 5.1

Table 5.1: List of FFT Functions

FILTER:

The filter module contains the Finite Impule Response (FIR) and Infinite Impulse Response (IIR) filters

Filter Module	
FIR_f32	void FIR_f32_calc(FIR_f32_handle);
IIR_f32	void IIR_f32_calc(IIR_f32_handle);

Table 5.2: List of Filter Functions

VECTOR:

The Vector module contains various routines to handle complex vector arithmetic

Vector Module	
abs_SP_CV	void abs_SP_CV(float32 *, const complex_float *, const Uint16);
abs_SP_CV_2	void abs_SP_CV_2(float32 *, const complex_float *, const Uint16);
abs_SP_CV_TMU0	void abs_SP_CV_TMU0(float32 *, const complex_float *, const Uint16);
add_SP_CSxCV	void add_SP_CSxCV(complex_float *, const complex_float *, const complex_float, const Uint16);
add_SP_CVxCV	void add_SP_CVxCV(complex_float *, const complex_float *, const complex_float *, const Uint16);
iabs_SP_CV	void iabs_SP_CV(float32 *, const complex_float *, const Uint16);
iabs_SP_CV_2	void iabs_SP_CV_2(float32 *, const complex_float *, const Uint16);
iabs_SP_CV_TMU0	void iabs_SP_CV_TMU0(float32 *, const complex_float *, const Uint16);
mac_SP_CVxCV	complex_float mac_SP_RVxCV(const complex_float *, const complex_float *, const uint16_t);
mac_SP_RVxCV	complex_float mac_SP_RVxCV(const complex_float *, const float *, const uint16_t);
mac_SP_i16RVxCV	complex_float mac_SP_i16RVxCV(const complex_float *, const int16_t *, const uint16_t);
maxidx_SP_RV_2	Uint16 maxidx_SP_RV_2(float32 *, Uint16);
mean_SP_CV_2	complex_float mean_SP_CV_2(const complex_float *, const Uint16);
median_noreorder_SP_RV	float32 median_noreorder_SP_RV(const float32 *, Uint16);
median_SP_RV	float32 median_SP_RV(float32 *, Uint16);
mpy_SP_CSxCS	complex_float mpy_SP_CSxCS(complex_float, complex_float);
mpy_SP_CVxCV	void mpy_SP_CVxCV(complex_float *, const complex_float *, const complex_float *, const Uint16);

Continued on next page

Table 5.3 – continued from previous page

mpy_SP_CVxCVC	void mpy_SP_CVxCVC(complex_float *, const complex_float *, const complex_float *, const Uint16);
mpy_SP_RSxRV_2	void mpy_SP_RSxRV_2(float32 *, const float32 *, const float32, const Uint16);
mpy_SP_RSxRVxRV_2	void mpy_SP_RSxRVxRV_2(float32 *, const float32 *, const float32 *, const float32, const Uint16);
mpy_SP_RVxCV	void mpy_SP_RVxCV(complex_float *, const complex_float *, const float32 *, const Uint16);
mpy_SP_RVxRV_2	void mpy_SP_RVxRV_2(float32 *, const float32 *, const float32 *, const Uint16);
qsort_SP_RV	void qsort_SP_RV(void *, Uint16);
rnd_SP_RS	float32 rnd_SP_RS(float32);
sub_SP_CSxCV	void sub_SP_CSxCV(complex_float *, const complex_float *, const complex_float, const Uint16);
sub_SP_CVxCV	void sub_SP_CVxCV(complex_float *, const complex_float *, const complex_float *, const Uint16);

Table 5.3: List of Vector Functions

UTILITY:

The Utility module has routines to copy and set data

Utility Module	
memcpy_fast	void memcpy_fast(void *, const void *, Uint16);
memcpy_fast_far	void memcpy_fast_far(volatile void* , volatile const void* , uint16_t);
memset_fast	void memset_fast(void*, int16, Uint16);

Table 5.4: List of Utility Functions

The examples for each of these routines was built using **CGT v22.6.0.LTS** with the following options:

```
-v28 -mt -ml -g --diag_warning=225 --float_support=fpu32 --tmu_support=tmu0
--define=CPU1
```

Each example has at least two build configurations, **RAM** and **FLASH**. Certain examples like the FFT have additional build configurations that either demonstrate the TMU variants of certain functions, or the use of the fast RTS support library to speed up phase calculations, or the use or ROM tables for the FFT computation. RAM, RAM_ROMTABLES and FLASH configurations have been validated on F28002x and F28003x and RAM and RAM_ROMTABLES configurations have been validated on and .

Certain functions can be redefined to their TMU alternatives in order to maintain legacy code functionality. For example,

```
#ifdef __TMS320C28XX_TMU__
#define CFFT_f32_mag    CFFT_f32_mag_TMU0
#warn "Legacy function has been redefined to its TMU variant"
#endif //__TMS320C28XX_TMU__
```

The macro **__TMS320C28XX_TMU__** is defined by the compiler when the tmu_support option is turned on.

In order to highlight the interleaving ability of the compiler for the fast square root function, its example was built with the options

```
-v28 -mt -ml -g -O2 --diag_warning=225 --optimize_with_debug
--float_support=fpu32
```

5.2 DSP Library Definitions and Types

Data Structures

- float64u_t
- float32u_t

Defines

- LIBRARY_VERSION

5.2.1 Data Structure Documentation

5.2.1.1 float64u_t

Definition:

```
typedef struct
{
    uint64_t ui64;
    int64_t i64;
    float64_t f64;
}
float64u_t
```

Members:

ui64 Unsigned long long representation.

i64 Signed long long represntaion.

f64 Double precision (64-bit) representation.

Description:

64-bit Double Precision Float The union of a double precision value, an unsigned long long and a signed long long allows for manipulation of the hex representation of the floating point value as well as signed and unsigned arithmetic to determine error metrics. This data type is only defined if the compiler option –float_support is set to fpu64

5.2.1.2 float32u_t

Definition:

```
typedef struct
{
    uint32_t ui32;
    int32_t i32;
    float f32;
}
float32u_t
```

Members:

ui32 Unsigned long representation.

i32 Signed long represntaion.

f32 Single precision (32-bit) representation.

Description:

32-bit Single Precision Float The union of a single precision value, an unsigned long and a signed long allows for manipulation of the hex representation of the floating point value as well as signed and unsigned arithmetic to determine error metrics. This data type is only defined if the compiler option –float_support is set to fpu32

5.2.2 Define Documentation

5.2.2.1 LIBRARY_VERSION

Definition:

```
#define LIBRARY_VERSION
```

Description:

DSP Library Version.

5.2.3 Typedef Documentation

5.2.3.1 v_pfn_v

Function pointer with a void pointer argument returning nothing.

5.3 Single Precision DSP Library Data Types

Data Structures

- complex_float

Defines

- USE_LEGACY_NAMES

5.3.1 Data Structure Documentation

5.3.1.1 complex_float

Definition:

```
typedef struct
{
    HASH(0x563bc84d21c0) mutable;
}
complex_float
```

Members:

mutable

Description:

Complex Float data type for the single precision DSP library.

5.3.2 Define Documentation

5.3.2.1 USE_LEGACY_NAMES

Definition:

```
#define USE_LEGACY_NAMES
```

Description:

Set to 1U if you would like to use legacy names from v1.50.00.00 of the DSP library; set to 0U (default) to use the new naming convention. It is important to note that not all module elements were updated from the old naming scheme. Rebuild the library and examples after changing this value

5.4 Complex Fast Fourier Transforms

Data Structures

- CFFT_F32_STRUCT

Functions

- static void CFFT_f32_setInputPtr (CFFT_F32_STRUCT_Handle fh, const float ∗pi)
- static float ∗ CFFT_f32_getInputPtr (CFFT_F32_STRUCT_Handle fh)
- static void CFFT_f32_setOutputPtr (CFFT_F32_STRUCT_Handle fh, const float ∗po)
- static float ∗ CFFT_f32_getOutputPtr (CFFT_F32_STRUCT_Handle fh)
- static void CFFT_f32_setTwiddlesPtr (CFFT_F32_STRUCT_Handle fh, const float ∗pc)
- static float ∗ CFFT_f32_getTwiddlesPtr (CFFT_F32_STRUCT_Handle fh)
- static void CFFT_f32_setCurrInputPtr (CFFT_F32_STRUCT_Handle fh, const float ∗pi)
- static float ∗ CFFT_f32_getCurrInputPtr (CFFT_F32_STRUCT_Handle fh)
- static void CFFT_f32_setCurrOutputPtr (CFFT_F32_STRUCT_Handle fh, const float ∗po)
- static float ∗ CFFT_f32_getCurrOutputPtr (CFFT_F32_STRUCT_Handle fh)
- static void CFFT_f32_setStages (CFFT_F32_STRUCT_Handle fh, const uint16_t st)
- static uint16_t CFFT_f32_getStages (CFFT_F32_STRUCT_Handle fh)
- static void CFFT_f32_setFFTSize (CFFT_F32_STRUCT_Handle fh, const uint16_t sz)
- static uint16_t CFFT_f32_getFFTSize (CFFT_F32_STRUCT_Handle fh)
- void CFFT_f32 (CFFT_F32_STRUCT_Handle hndCFFT_F32)
- void CFFT_f32_brev (CFFT_F32_STRUCT_Handle hndCFFT_F32)
- void CFFT_f32i (CFFT_F32_STRUCT_Handle hndCFFT_F32)
- void CFFT_f32t (CFFT_F32_STRUCT_Handle hndCFFT_F32)
- void CFFT_f32it (CFFT_F32_STRUCT_Handle hndCFFT_F32)
- void CFFT_f32u (CFFT_F32_STRUCT_Handle hndCFFT_F32)
- void CFFT_f32ut (CFFT_F32_STRUCT_Handle hndCFFT_F32)
- void CFFT_f32_sincostable (CFFT_F32_STRUCT_Handle hndCFFT_F32)
- void CFFT_f32_win (float ∗pBuffer, const float ∗pWindow, const uint16_t size)
- void CFFT_f32_win_dual (float ∗pBuffer, const float ∗pWindow, const uint16_t size)
- void CFFT_f32_mag (CFFT_F32_STRUCT_Handle hndCFFT_F32)
- void CFFT_f32s_mag (CFFT_F32_STRUCT_Handle hndCFFT_F32)
- void CFFT_f32_phase (CFFT_F32_STRUCT_Handle hndCFFT_F32)
- void CFFT_f32_unpack (CFFT_F32_STRUCT_Handle hndCFFT_F32)
- void CFFT_f32_pack (CFFT_F32_STRUCT_Handle hndCFFT_F32)
- void CFFT_f32_mag_TMU0 (CFFT_F32_STRUCT_Handle hndCFFT_F32)
- void CFFT_f32s_mag_TMU0 (CFFT_F32_STRUCT_Handle hndCFFT_F32)
- void CFFT_f32_phase_TMU0 (CFFT_F32_STRUCT_Handle hndCFFT_F32)
- void ICFFT_f32 (CFFT_F32_STRUCT_Handle hndCFFT_F32)
- void ICFFT_f32t (CFFT_F32_STRUCT_Handle hndCFFT_F32)

Variables

- float CFFT_f32_twiddleFactors[1536]

5.4.1 Data Structure Documentation

5.4.1.1 CFFT_F32_STRUCT

Definition:

```
typedef struct
{
    float *InPtr;
    float *OutPtr;
    float *CoefPtr;
    float *CurrentInPtr;
    float *CurrentOutPtr;
    uint16_t Stages;
    uint16_t FFTSize;
}
CFFT_F32_STRUCT
```

Members:

InPtr Pointer to the input buffer.
OutPtr Pointer to the output buffer.
CoefPtr Pointer to the twiddle factors.
CurrentInPtr Points to input buffer at each FFT stage.
CurrentOutPtr Points to output buffer at each FFT stage.
Stages Number of FFT stages.
FFTSize Size of the FFT (number of complex data points).

Description:

Complex FFT structure.

5.4.2 Function Documentation

5.4.2.1 CFFT_f32_setInputPtr

Set the input buffer pointer.

Prototype:

```
static void
CFFT_f32_setInputPtr(CFFT_F32_STRUCT_Handle fh,
                     const float *pi) [inline, static]
```

Parameters:

← ***fh*** handle to the 'CFFT_F32_STRUCT' object
← ***pi*** pointer to the input buffer

5.4.2.2 static float* CFFT_f32_getInputPtr (CFFT_F32_STRUCT_Handle *fh*) [inline, static]

Get the input buffer pointer.

Parameters:
← ***fh*** handle to the 'CFFT_F32_STRUCT' object

Returns:
pi pointer to the input buffer

5.4.2.3 static void CFFT_f32_setOutputPtr (CFFT_F32_STRUCT_Handle *fh*, const float * *po*) [inline, static]

Set the output buffer pointer.

Parameters:
← ***fh*** handle to the 'CFFT_F32_STRUCT' object
← ***po*** pointer to the output buffer

5.4.2.4 static float* CFFT_f32_getOutputPtr (CFFT_F32_STRUCT_Handle *fh*) [inline, static]

Get the output buffer pointer.

Parameters:
← ***fh*** handle to the 'CFFT_F32_STRUCT' object

Returns:
po pointer to the output buffer

5.4.2.5 static void CFFT_f32_setTwiddlesPtr (CFFT_F32_STRUCT_Handle *fh*, const float * *pc*) [inline, static]

Set the twiddles pointer.

Parameters:
← ***fh*** handle to the 'CFFT_F32_STRUCT' object
← ***pt*** pointer to the twiddles

5.4.2.6 static float* CFFT_f32_getTwiddlesPtr (CFFT_F32_STRUCT_Handle *fh*) [inline, static]

Get the twiddles pointer.

Parameters:
← ***fh*** handle to the 'CFFT_F32_STRUCT' object

Returns:
pc pointer to the twiddles

5.4.2.7 static void CFFT_f32_setCurrInputPtr (CFFT_F32_STRUCT_Handle *fh*, const float * *pi*) [inline, static]

Set the current input buffer pointer.

Parameters:
← ***fh*** handle to the 'CFFT_F32_STRUCT' object
← ***pi*** pointer to the current input buffer

5.4.2.8 static float* CFFT_f32_getCurrInputPtr (CFFT_F32_STRUCT_Handle *fh*) [inline, static]

Get the current input buffer pointer.

Parameters:
← ***fh*** handle to the 'CFFT_F32_STRUCT' object

Returns:
pi pointer to the current input buffer

5.4.2.9 static void CFFT_f32_setCurrOutputPtr (CFFT_F32_STRUCT_Handle *fh*, const float * *po*) [inline, static]

Set the current output buffer pointer.

Parameters:
← ***fh*** handle to the 'CFFT_F32_STRUCT' object
← ***po*** pointer to the current output buffer

5.4.2.10 static float* CFFT_f32_getCurrOutputPtr (CFFT_F32_STRUCT_Handle *fh*) [inline, static]

Get the current output buffer pointer.

Parameters:
← ***fh*** handle to the 'CFFT_F32_STRUCT' object

Returns:
po pointer to the current output buffer

5.4.2.11 static void CFFT_f32_setStages (CFFT_F32_STRUCT_Handle *fh*, const uint16_t *st*) [inline, static]

Set the number of FFT stages.

Parameters:
← ***fh*** handle to the 'CFFT_F32_STRUCT' object
← ***st*** number of FFT stages

5.4.2.12 static uint16_t CFFT_f32_getStages (CFFT_F32_STRUCT_Handle *fh*) [inline, static]

Get the size of the FFT.

Parameters:
← ***fh*** handle to the 'CFFT_F32_STRUCT' object

Returns:
st number of FFT stages

5.4.2.13 static void CFFT_f32_setFFTSize (CFFT_F32_STRUCT_Handle *fh*, const uint16_t *sz*) [inline, static]

Set the number of FFT stages.

Parameters:
← ***fh*** handle to the 'CFFT_F32_STRUCT' object
← ***sz*** size of the FFT

5.4.2.14 static uint16_t CFFT_f32_getFFTSize (CFFT_F32_STRUCT_Handle *fh*) [inline, static]

Get the size of the FFT.

Parameters:
← ***fh*** handle to the 'CFFT_F32_STRUCT' object

Returns:
sz size of the FFT

5.4.2.15 void CFFT_f32 (CFFT_F32_STRUCT_Handle *hndCFFT_F32*)

Complex Fast Fourier Transform.

This routine computes the 32-bit floating-point FFT for an N-pt ($N = 2^n, n = 5 : 10$) complex input. This function will reorder the input in bit-reversed format before proceeding with the FFT. The routine uses two buffers in ping-pong fashion i.e. after each FFT stage the output and input buffers become the input and output buffers respectively for the next stage. The CFFT_F32 object uses two pointers, CurrentInPtr and CurrentOutPtr to keep track of the switching. The user can determine the address of the final output by looking at the CurrentInPtr.

Parameters:
hndCFFT_F32 Pointer to the CFFT_F32 object

Attention:
1. The routine requires the use of two buffers, each of size 2N (32-bit float), for computation; the input buffer must be aligned to a memory address of 4N words (16-bit). Refer to the CFFT linker command file to see an example of this.
2. If alignment is not possible the user can use the alternative, albeit slower, function CFFT_f32u

Warning:
This function is not re-entrant as it uses global variables to store certain parameters

Samples	Cycles
64	2334
128	5032
256	11024
512	24250
1024	53220

Table 5.5: Performance Data

5.4.2.16 void CFFT_f32_brev (CFFT_F32_STRUCT_Handle *hndCFFT_F32*)

Complex Data Bit Reversal.

This routine will reorder an N point complex data set in bit reverse order. It can be run in-place, i.e. with the input and output buffer pointers pointing to the same array, or off-place with different input and output buffers

Parameters:
hndCFFT_F32 Pointer to the CFFT_F32 object

Attention:
1. The buffer must be aligned to a memory address of 4N words (16-bit). Refer to the CFFT linker command file to see an example of this.
2. You could use the bit-reversal function off-place to preserve the original input while calling the in-place FFT function, CFFT_f32i() or CFFT_f32it(), on the bit reversed buffer
3. This function is mainly to be used prior to calling the in-place variant of the complex FFT functions, CFFT_f32i() or CFFT_f32it().
4. If the user has the space to use two N-point complex float buffers then use the faster CFFT_f32() or CFFT_f32t() functions that does the bit reversal as part of its stage 1 computation

5.4.2.17 void CFFT_f32i (CFFT_F32_STRUCT_Handle *hndCFFT_F32*)

Complex Fast Fourier Transform (In-Place).

Samples	Cycles
64	1621
128	3217
256	6352
512	13968
1024	28500

Table 5.6: Performance Data

Parameters:
hndCFFT_F32 Pointer to the CFFT_F32 object

Attention:
1. This function does not reorder the input in bit-reversed format before proceeding with the FFT. It assumes the user has already bit reversed the input by calling CFFT_f32_brev() prior to calling this function
2. The routine requires a single buffer of size 2N (32-bit float), for computation; the buffer must be aligned to a memory address of 4N words (16-bit). Refer to the CFFT linker command file to see an example of this.
3. If alignment is not possible the user can use the alternative, albeit slower, function CFFT_f32u

Warning:
This function is not re-entrant as it uses global variables to store certain parameters

Samples	Cycles
64	2407
128	5248
256	11591
512	25648
1024	56535

Table 5.7: Performance Data

5.4.2.18 void CFFT_f32t (CFFT_F32_STRUCT_Handle *hndCFFT_F32*)

Complex Fast Fourier Transform using a Pre Generated Twiddle Factor Table.

This routine computes the 32-bit floating-point FFT for an N-pt ($N = 2^n, n = 5 : 10$) complex input. This function will reorder the input in bit-reversed format before proceeding with the FFT. The routine uses two buffers in ping-pong fashion i.e. after each FFT stage the output and input buffers become the input and output buffers respectively for the next stage. The CFFT_F32 object uses two pointers, CurrentInPtr and CurrentOutPtr to keep track of the switching. The user can determine the address of the final output by looking at the CurrentInPtr.

Parameters:
hndCFFT_F32 Pointer to the CFFT_F32 object

Attention:
1. The routine requires the use of two buffers, each of size 2N (32-bit float), for computation; the input buffer must be aligned to a memory address of 4N words (16-bit). Refer to the CFFT linker command file to see an example of this.

2. If alignment is not possible the user can use the alternative, albeit slower, function CFFT_f32ut

Warning:

This function is not re-entrant as it uses global variables to store certain parameters

Samples	Cycles
64	2334
128	5032
256	11026
512	24250
1024	53220

Table 5.8: Performance Data

5.4.2.19 void CFFT_f32it (CFFT_F32_STRUCT_Handle *hndCFFT_F32*)

Complex Fast Fourier Transform (In-Place) using a Pre Generated Twiddle Factor Table.

Parameters:

hndCFFT_F32 Pointer to the CFFT_F32 object

Attention:

1. This function does not reorder the input in bit-reversed format before proceeding with the FFT. It assumes the user has already bit reversed the input by calling CFFT_f32_brev() prior to calling this function
2. The routine requires a single buffer of size 2N (32-bit float), for computation; the buffer must be aligned to a memory address of 4N words (16-bit). Refer to the CFFT linker command file to see an example of this.
3. If alignment is not possible the user can use the alternative, albeit slower, function CFFT_f32ut

Warning:

This function is not re-entrant as it uses global variables to store certain parameters

Samples	Cycles
64	2407
128	5248
256	11593
512	25648
1024	56535

Table 5.9: Performance Data

5.4.2.20 void CFFT_f32u (CFFT_F32_STRUCT_Handle *hndCFFT_F32*)

Complex Fast Fourier Transform (Unaligned).

This routine computes the 32-bit floating-point FFT for an N-pt ($N = 2^n, n = 5 : 10$) complex input. This function will reorder the input in bit-reversed format before proceeding with the FFT.

The routine uses two buffers in ping-pong fashion i.e. after each FFT stage the output and input buffers become the input and output buffers respectively for the next stage. The CFFT_F32 object uses two pointers, CurrentInPtr and CurrentOutPtr to keep track of the switching. The user can determine the address of the final output by looking at the CurrentInPtr.

Parameters:

hndCFFT_F32 Pointer to the CFFT_F32 object

Attention:

1. The routine requires the use of two buffers, each of size 2N (32-bit float), for computation; the input buffer need not be aligned to any boundary.
2. If alignment is possible the user can use the faster routine, CFFT_f32

Warning:

This function is not re-entrant as it uses global variables to store certain parameters

Samples	Cycles
64	2787
128	5933
256	12821
512	27839
1024	60393

Table 5.10: Performance Data

5.4.2.21 void CFFT_f32ut (CFFT_F32_STRUCT_Handle *hndCFFT_F32*)

Complex Fast Fourier Transform (Unaligned) using a statically generated twiddle factor table.

This routine computes the 32-bit floating-point FFT for an N-pt ($N = 2^n, n = 5 : 10$) complex input. This function will reorder the input in bit-reversed format before proceeding with the FFT. The routine uses two buffers in ping-pong fashion i.e. after each FFT stage the output and input buffers become the input and output buffers respectively for the next stage. The CFFT_F32 object uses two pointers, CurrentInPtr and CurrentOutPtr to keep track of the switching. The user can determine the address of the final output by looking at the CurrentInPtr.

Parameters:

hndCFFT_F32 Pointer to the CFFT_F32 object

Attention:

1. The routine requires the use of two buffers, each of size 2N (32-bit float), for computation; the input buffer need not be aligned to any boundary.
2. If alignment is possible the user can use the faster routine, CFFT_f32

Warning:

This function is not re-entrant as it uses global variables to store certain parameters

5.4.2.22 void CFFT_f32_sincostable (CFFT_F32_STRUCT_Handle *hndCFFT_F32*)

Generate twiddle factors for the Complex FFT.

Samples	Cycles
64	2787
128	5933
256	12821
512	27839
1024	60393

Table 5.11: Performance Data

Parameters:
hndCFFT_F32 Pointer to the CFFT_F32 object

Attention:
This function is written in C and compiled without optimization turned on.

5.4.2.23 void CFFT_f32_win (float ∗ *pBuffer*, const float ∗ *pWindow*, const uint16_t *size*)

Windowing function for the 32-bit complex FFT.

Parameters:
pBuffer pointer to the buffer that needs to be windowed
pWindow pointer to the windowing table
size size of the buffer This function applies the window to only the real portion of the N point complex buffer that has already been reordered in the bit-reversed format

Samples	Cycles
64	316
128	604
256	1180
512	2332
1024	4636

Table 5.12: Performance Data

5.4.2.24 void CFFT_f32_win_dual (float ∗ *pBuffer*, const float ∗ *pWindow*, const uint16_t *size*)

Windowing function for the 32-bit complex FFT.

Parameters:
pBuffer pointer to the buffer that needs to be windowed
pWindow pointer to the windowing table
size size of the buffer This function applies the window to both the real and imaginary parts of the input complex buffer that has already been reordered in the bit-reversed format

5.4.2.25 void CFFT_f32_mag (CFFT_F32_STRUCT_Handle *hndCFFT_F32*)

Complex FFT Magnitude.

Samples	Cycles
64	595
128	1171
256	2323
512	4627
1024	9235

Table 5.13: Performance Data

This module computes the complex FFT magnitude. The output from CFFT_f32_mag matches the magnitude output from the FFT found in common mathematics software and Code Composer Studio FFT graphs. If instead a normalized magnitude like that performed by the fixed-point TMS320C28x IQmath FFT library is required, then the CFFT_f32s_mag function can be used. In fixed-point algorithms scaling is performed to avoid overflowing data. Floating-point calculations do not need this scaling to avoid overflow and therefore the CFFT_f32_mag function can be used instead.

Parameters:

hndCFFT_F32 Pointer to the CFFT_F32 object

Attention:

1. The Magnitude buffer does not require memory alignment to a boundary
2. For C28x devices that have the TMU0 (or higher) module, use CFFT_f32_mag_TMU0() instead for better performance.

Samples	Cycles
64	1181
128	2333
256	4635
512	9243
1024	18459

Table 5.14: Performance Data

5.4.2.26 void CFFT_f32s_mag (CFFT_F32_STRUCT_Handle *hndCFFT_F32*)

Complex FFT Magnitude (Scaled).

This module computes the scaled complex FFT magnitude. The scaling is $\frac{1}{[2^{FFT\_STAGES-1}]}$, and is done to match the normalization performed by the fixed-point TMS320C28x IQmath FFT library for overflow avoidance. Floating-point calculations do not need this scaling to avoid overflow and therefore the CFFT_f32_mag function can be used instead. The output from CFFT_f32s_mag matches the magnitude output from the FFT found in common mathematics software and Code Composer Studio FFT graphs.

Parameters:

hndCFFT_F32 Pointer to the CFFT_F32 object

Attention:

1. The Magnitude buffer does not require memory alignment to a boundary
2. For C28x devices that have the TMU0 (or higher) module, use CFFT_f32s_mag_TMU0() instead for better performance.

Samples	Cycles
64	1284
128	2506
256	4942
512	9812
1024	19546

Table 5.15: Performance Data

5.4.2.27 void CFFT_f32_phase (CFFT_F32_STRUCT_Handle *hndCFFT_F32*)

Complex FFT Phase.

Parameters:

hndCFFT_F32 Pointer to the CFFT_F32 object

Attention:

1. The Phase buffer does not require memory alignment to a boundary
2. The phase function calls the atan2 function in the runtime-support library. The phase function has not been optimized at this time.
3. The use of the atan2 function in the FPUfastRTS library will speed up this routine. The example for the CFFT has an alternate build configuration (Debug_FASTRTS) where the rts2800_fpu32_fast_supplement.lib is used in place of the standard runtime library rts2800_fpu32.lib.
4. For C28x devices that have the TMU0 (or higher) module, use CFFT_f32_phase_TMU0() instead for better performance.

Using atan2() from the fast RTS library	
Samples	Cycles
64	3797
128	7573
256	14866
512	29714
1024	60434

Table 5.16: Performance Data

5.4.2.28 void CFFT_f32_unpack (CFFT_F32_STRUCT_Handle *hndCFFT_F32*)

Unpack the N-point complex FFT output to get the FFT of a 2N point real sequence.

In order to get the FFT of a real N-point sequence, we treat the input as an N/2-point complex sequence, take its complex FFT, use the following properties to get the N-pt Fourier transform of the real sequence

$$FFT_n(k,f) = FFT_{N/2}(k,f_e) + e^{\frac{-j2\pi k}{N}} FFT_{N/2}(k,f_o)$$

where f_e is the even elements, f_o the odd elements, k = 0 to $\frac{N}{2} - 1$ and

$$F_e(k) = \frac{Z(k) + Z(\frac{N}{2} - k)^*}{2}$$

$$F_o(k) = -j\frac{Z(k) - Z(\frac{N}{2} - k)^*}{2}$$

We get the first N/2 points of the FFT by combining the above two equations

$$F(k) = F_e(k) + e^{\frac{-j2\pi k}{N}} F_o(k)$$

Parameters:

hndCFFT_F32 Pointer to the CFFT_F32 object

Note:

1. The unpack routine yields the spectrum of the real input data; the spectrum has a real part that is symmetric, and an imaginary part that is antisymmetric about the nyquist frequency. We only need calculate half the spectrum, up to the nyquist bin, while the latter half can be derived from the first half using the conjugate symmetry properties of the spectrum
   - the latter half can be derived using the symmetry properties
2. The output is written to the buffer pointer to by CurrentOutPtr
3. Only use the CFFT_f32t version with this function

See also:

http://www.engineeringproductivitytools.com/stuff/T0001/PT10.HTM for the entire derivation

Real FFT Using a Complex FFT

Samples	Cycles
64	562
128	1136
256	2098
512	4399
1024	8241

Table 5.17: Performance Data

5.4.2.29 void CFFT_f32_pack (CFFT_F32_STRUCT_Handle *hndCFFT_F32*)

Pack the N/2-point complex FFT output to get the spectrum of an N-point real sequence.

In order to reverse the process of the forward real FFT,

$$F_e(k) = \frac{F(k) + F(\frac{N}{2} - k)^*}{2}$$

$$F_o(k) = \frac{F(k) - F(\frac{N}{2} - k)^*}{2} e^{\frac{j2\pi k}{N}}$$

where f_e is the even elements, f_o the odd elements, and k = 0 to $\frac{N}{2} - 1$. The array for the IFFT then becomes:

$$Z(k) = F_e(k) + jF_o(k),\ k = 0...\frac{N}{2} - 1$$

Parameters:

hndCFFT_F32 Pointer to the CFFT_F32 object

Note:

1. The output is written to the buffer pointer to by CurrentOutPtr
2. Only use the CFFT_f32t version with this function

See also:

http://www.engineeringproductivitytools.com/stuff/T0001/PT10.HTM for the entire derivation

Real FFT Using a Complex FFT

Samples	Cycles
64	564
128	1076
256	2100
512	4148
1024	8243

Table 5.18: Performance Data

5.4.2.30 void CFFT_f32_mag_TMU0 (CFFT_F32_STRUCT_Handle *hndCFFT_F32*)

Complex FFT Magnitude using the TMU0 module.

This module computes the complex FFT magnitude. The output from CFFT_f32_mag matches the magnitude output from the FFT found in common mathematics software and Code Composer Studio FFT graphs. If instead a normalized magnitude like that performed by the fixed-point TMS320C28x IQmath FFT library is required, then the CFFT_f32s_mag function can be used. In fixed-point algorithms scaling is performed to avoid overflowing data. Floating-point calculations do not need this scaling to avoid overflow and therefore the CFFT_f32_mag function can be used instead.

Parameters:

hndCFFT_F32 Pointer to the CFFT_F32 object

Attention:

1. The Magnitude buffer does not require memory alignment to a boundary
2. This function requires a C28x device with TMU0 or higher module. For devices without the TMU, use CFFT_f32_mag() instead.

Samples	Cycles
64	342
128	662
256	1302
512	2582
1024	5142

Table 5.19: Performance Data

5.4.2.31 void CFFT_f32s_mag_TMU0 (CFFT_F32_STRUCT_Handle *hndCFFT_F32*)

Complex FFT Magnitude using the TMU0 module (Scaled).

This module computes the scaled complex FFT magnitude. The scaling is $\frac{1}{[2^{FFT\_STAGES-1}]}$, and is done to match the normalization performed by the fixed-point TMS320C28x IQmath FFT library for overflow avoidance. Floating-point calculations do not need this scaling to avoid overflow and therefore the CFFT_f32_mag function can be used instead. The output from CFFT_f32s_mag matches the magnitude output from the FFT found in common mathematics software and Code Composer Studio FFT graphs.

Parameters:

hndCFFT_F32 Pointer to the CFFT_F32 object

Attention:

1. The Magnitude buffer does not require memory alignment to a boundary
2. The magnitude calculation calls the sqrt function within the runtime-support library. The magnitude function has not been optimized at this time.
3. The use of the sqrt function in the FPUfastRTS library will speed up this routine. The example for the CFFT has an alternate build configuration (Debug_FASTRTS) where the rts2800_fpu32_fast_supplement.lib is used in place of the standard runtime library rts2800_fpu32.lib.
4. This function requires a C28x device with TMU0 or higher module. For devices without the TMU, use CFFT_f32s_mag() instead.

Samples	Cycles
64	412
128	769
256	1476
512	2889
1024	5710

Table 5.20: Performance Data

5.4.2.32 void CFFT_f32_phase_TMU0 (CFFT_F32_STRUCT_Handle *hndCFFT_F32*)

Complex FFT Phase using the TMU0 module.

Parameters:

hndCFFT_F32 Pointer to the CFFT_F32 object

Attention:

1. The Phase buffer does not require memory alignment to a boundary
2. The phase function calls the atan2 function in the runtime-support library. The phase function has not been optimized at this time.
3. The use of the atan2 function in the FPUfastRTS library will speed up this routine. The example for the CFFT has an alternate build configuration (Debug_FASTRTS) where the rts2800_fpu32_fast_supplement.lib is used in place of the standard runtime library rts2800_fpu32.lib.
4. This function requires a C28x device with TMU0 or higher module. For devices without the TMU, use CFFT_f32_phase() instead.

Samples	Cycles
64	480
128	928
256	1824
512	3615
1024	7199

Table 5.21: Performance Data

5.4.2.33 void ICFFT_f32 (CFFT_F32_STRUCT_Handle *hndCFFT_F32*)

Inverse Complex FFT.

This routine computes the 32-bit floating-point Inverse FFT for an N-pt ($N = 2^n, n = 5 : 10$) complex input. It uses the forward FFT to do this by first swapping the real and imaginary parts of the input, running the forward FFT and then swapping the real and imaginary parts of the output to get the final answer. The routine uses two buffers in ping-pong fashion i.e. after each FFT stage the output and input buffers become the input and output buffers respectively for the next stage. The CFFT_F32 object uses two pointers, CurrentInPtr and CurrentOutPtr to keep track of the switching. The user can determine the address of the final output by looking at the CurrentInPtr.

Parameters:

hndCFFT_F32 Pointer to the CFFT_F32 object

Attention:

1. The routine requires the use of two buffers, each of size 2N (32-bit float), for computation; the input buffer must be aligned to a memory address of 4N words (16-bit). Refer to the ICFFT linker command file to see an example of this.

Samples	Cycles
64	2808
128	5954
256	12843
512	27861
1024	60415

Table 5.22: Performance Data

5.4.2.34 void ICFFT_f32t (CFFT_F32_STRUCT_Handle *hndCFFT_F32*)

Inverse Complex FFT using a statically generated twiddle factor table.

This routine computes the 32-bit floating-point Inverse FFT for an N-pt ($N = 2^n, n = 5 : 10$) complex input. It uses the forward FFT to do this by first swapping the real and imaginary parts of the input, running the forward FFT and then swapping the real and imaginary parts of the output to get the final answer. The routine uses two buffers in ping-pong fashion i.e. after each FFT stage the output and input buffers become the input and output buffers respectively for the next stage. The CFFT_F32 object uses two pointers, CurrentInPtr and CurrentOutPtr to keep track of the switching. The user can determine the address of the final output by looking at the CurrentInPtr.

Parameters:

hndCFFT_F32 Pointer to the CFFT_F32 object

Attention:

1. The routine requires the use of two buffers, each of size 2N (32-bit float), for computation; the input buffer must be aligned to a memory address of 4N words (16-bit). Refer to the ICFFT linker command file to see an example of this.

Samples	Cycles
64	2921
128	6180
256	13549
512	29271
1024	64256

Table 5.23: Performance Data

5.4.3 Variable Documentation

5.4.3.1 float CFFT_f32_twiddleFactors[1536]

Twiddle Factor Table for a 1024-pt (max) Complex FFT.

Note: 1. CFFT_f32_twiddleFactors name is deprecated and only supported for legacy reasons. Users are encouraged to use FPU32CFFTtwiddleFactors as the table symbol. 2. The CFFT_f32_twiddleFactors is an alias for the new table. It is not a separate table.

5.5 Real FFT Using a Complex FFT

It is possible to run the Fast Fourier Transform on a sequence of real data using the complex FFT. For a 2N point real sequence, the user would treat the data as N-pt complex (no rearrangement required) and run it through an N point complex FFT. In order to derive the correct spectrum, you would have to “unpack” the output. The derivations can be found here:

http://www.engineeringproductivitytools.com/stuff/T0001/PT10.HTM

Similarly, to run an inverse Real FFT, the user would “pack” the data and run it through an N-point Forward Complex FFT and then conjugate its complex output to get the original 2N point signal.

Note 1 When running an inverse real FFT after the forward real FFT, the user must take care to first switch the **Input** and **Output** pointers in the FFT object before calling the FFT routine again.

Note 2 Refer to the project, **rfft_f32**, in the examples folder for a demonstration of the use of a complex FFT followed by the unpack function to compute the forward real FFT. The unpack routine yields the spectrum of the real input data; the spectrum has a real part that is symmetric, and an imaginary part that is antisymmetric about the nyquist frequency. We only need calculate half the spectrum, up to the nyquist bin, while the latter half can be derived from the first half using the conjugate symmetry properties of the spectrum.

Note 3 The project **rfft_alt_f32**, on the other hand, demonstrates the real FFT function, **R**FFT_f32, which runs a 2N point complex FFT on the real input data treating the imaginary portion as zeros, computing only half the spectrum - the spectrum of real data is complex conjugate about the nyquist point - and preserving only the real parts of certain points in the butterfly groups (in every stage) that are necessarily real.

Note 4 Refer to the project, **irfft_f32**, in the examples folder for a demonstration of the use of the pack function followed by a complex FFT to compute the inverse real FFT.

See also:
CFFT_f32_pack, CFFT_f32_unpack

5.6 Real Fast Fourier Transforms

Data Structures

- RFFT_F32_STRUCT
- RFFT_F32_STRUCT
- RFFT_ADC_F32_STRUCT

Functions

- static void RFFT_f32_setInputPtr (RFFT_F32_STRUCT_Handle fh, const float ∗pi)
- static float ∗ RFFT_f32_getInputPtr (RFFT_F32_STRUCT_Handle fh)
- static void RFFT_f32_setOutputPtr (RFFT_F32_STRUCT_Handle fh, const float ∗po)
- static float ∗ RFFT_f32_getOutputPtr (RFFT_F32_STRUCT_Handle fh)
- static void RFFT_f32_setTwiddlesPtr (RFFT_F32_STRUCT_Handle fh, const float ∗pc)
- static float ∗ RFFT_f32_getTwiddlesPtr (RFFT_F32_STRUCT_Handle fh)
- static void RFFT_f32_setMagnitudePtr (RFFT_F32_STRUCT_Handle fh, const float ∗pm)
- static float ∗ RFFT_f32_getMagnitudePtr (RFFT_F32_STRUCT_Handle fh)
- static void RFFT_f32_setPhasePtr (RFFT_F32_STRUCT_Handle fh, const float ∗pp)
- static float ∗ RFFT_f32_getPhasePtr (RFFT_F32_STRUCT_Handle fh)
- static void RFFT_f32_setStages (RFFT_F32_STRUCT_Handle fh, const uint16_t st)
- static uint16_t RFFT_f32_getStages (RFFT_F32_STRUCT_Handle fh)
- static void RFFT_f32_setFFTSize (RFFT_F32_STRUCT_Handle fh, const uint16_t sz)
- static uint16_t RFFT_f32_getFFTSize (RFFT_F32_STRUCT_Handle fh)
- static void RFFT_ADC_f32_setInBufPtr (RFFT_ADC_F32_STRUCT_Handle fh, const uint16_t ∗pi)
- static uint16_t ∗ RFFT_ADC_f32_getInBufPtr (RFFT_ADC_F32_STRUCT_Handle fh)
- static void RFFT_ADC_f32_setTailPtr (RFFT_ADC_F32_STRUCT_Handle fh, const void ∗pt)
- static void ∗ RFFT_ADC_f32_getTailPtr (RFFT_ADC_F32_STRUCT_Handle fh)
- void RFFT_f32 (RFFT_F32_STRUCT_Handle hndRFFT_F32)
- void RFFT_f32u (RFFT_F32_STRUCT_Handle hndRFFT_F32)
- void RFFT_adc_f32 (RFFT_ADC_F32_STRUCT_Handle hndRFFT_ADC_F32)
- void RFFT_adc_f32u (RFFT_ADC_F32_STRUCT_Handle hndRFFT_ADC_F32)
- void RFFT_f32_win (float ∗pBuffer, const float ∗pWindow, const uint16_t size)
- void RFFT_adc_f32_win (uint16_t ∗pBuffer, const uint16_t ∗pWindow, const uint16_t size)
- void RFFT_f32_mag (RFFT_F32_STRUCT_Handle hndRFFT_F32)
- void RFFT_f32s_mag (RFFT_F32_STRUCT_Handle hndRFFT_F32)
- void RFFT_f32_phase (RFFT_F32_STRUCT_Handle hndRFFT_F32)
- void RFFT_f32_mag_TMU0 (RFFT_F32_STRUCT_Handle hndRFFT_F32)
- void RFFT_f32s_mag_TMU0 (RFFT_F32_STRUCT_Handle hndRFFT_F32)
- void RFFT_f32_phase_TMU0 (RFFT_F32_STRUCT_Handle hndRFFT_F32)
- void RFFT_f32_sincostable (RFFT_F32_STRUCT_Handle hndRFFT_F32)

Variables

- float RFFT_f32_twiddleFactors[1020]

5.6.1 Data Structure Documentation

5.6.1.1 RFFT_F32_STRUCT

Definition:

```
typedef struct
{
    float *InBuf;
    float *OutBuf;
    float *CosSinBuf;
    float *MagBuf;
    float *PhaseBuf;
    uint16_t FFTSize;
    uint16_t FFTStages;
}
RFFT_F32_STRUCT
```

Members:

InBuf Pointer to the input buffer.
OutBuf Pointer to the output buffer.
CosSinBuf Pointer to the twiddle factors.
MagBuf Pointer to the magnitude buffer.
PhaseBuf Pointer to the phase buffer.
FFTSize Size of the FFT (number of real data points).
FFTStages Number of FFT stages.

Description:

Structure for the Real FFT.

5.6.1.2 RFFT_F32_STRUCT

Definition:

```
typedef struct
{
    float *InBuf;
    float *OutBuf;
    float *CosSinBuf;
    float *MagBuf;
    float *PhaseBuf;
    uint16_t FFTSize;
    uint16_t FFTStages;
}
RFFT_F32_STRUCT
```

Members:

InBuf Pointer to the input buffer.

OutBuf Pointer to the output buffer.
CosSinBuf Pointer to the twiddle factors.
MagBuf Pointer to the magnitude buffer.
PhaseBuf Pointer to the phase buffer.
FFTSize Size of the FFT (number of real data points).
FFTStages Number of FFT stages.

Description:
Structure for the Real FFT.

5.6.1.3 RFFT_ADC_F32_STRUCT

Definition:

```
typedef struct
{
    uint16_t *InBuf;
    void *Tail;
}
RFFT_ADC_F32_STRUCT
```

Members:
InBuf Pointer to the input buffer.
Tail HASH(0x563bc89f6610)

Description:
Structure for the Real FFT with ADC input.

5.6.2 Function Documentation

5.6.2.1 RFFT_f32_setInputPtr

Set the input buffer pointer.

Prototype:

```
static void
RFFT_f32_setInputPtr(RFFT_F32_STRUCT_Handle fh,
                     const float *pi) [inline, static]
```

Parameters:
← ***fh*** handle to the 'RFFT_F32_STRUCT' object
← ***pi*** pointer to the input buffer

5.6.2.2 static float* RFFT_f32_getInputPtr (RFFT_F32_STRUCT_Handle *fh*) [inline, static]

Get the input buffer pointer.

Parameters:
← ***fh*** handle to the 'RFFT_F32_STRUCT' object

Returns:
pi pointer to the input buffer

5.6.2.3 static void RFFT_f32_setOutputPtr (RFFT_F32_STRUCT_Handle *fh*, const float ∗ *po*) [inline, static]

Set the output buffer pointer.

Parameters:
← ***fh*** handle to the 'RFFT_F32_STRUCT' object
← ***po*** pointer to the output buffer

5.6.2.4 static float∗ RFFT_f32_getOutputPtr (RFFT_F32_STRUCT_Handle *fh*) [inline, static]

Get the output buffer pointer.

Parameters:
← ***fh*** handle to the 'RFFT_F32_STRUCT' object

Returns:
po pointer to the output buffer

5.6.2.5 static void RFFT_f32_setTwiddlesPtr (RFFT_F32_STRUCT_Handle *fh*, const float ∗ *pc*) [inline, static]

Set the twiddles pointer.

Parameters:
← ***fh*** handle to the 'RFFT_F32_STRUCT' object
← ***pt*** pointer to the twiddles

5.6.2.6 static float∗ RFFT_f32_getTwiddlesPtr (RFFT_F32_STRUCT_Handle *fh*) [inline, static]

Get the twiddles pointer.

Parameters:
← ***fh*** handle to the 'RFFT_F32_STRUCT' object

Returns:
pt pointer to the twiddles

5.6.2.7 static void RFFT_f32_setMagnitudePtr (RFFT_F32_STRUCT_Handle *fh*, const float * *pm*) [inline, static]

Set the magnitude buffer pointer.

Parameters:
- ← ***fh*** handle to the 'RFFT_F32_STRUCT' object
- ← ***pm*** pointer to the magnitude buffer

5.6.2.8 static float* RFFT_f32_getMagnitudePtr (RFFT_F32_STRUCT_Handle *fh*) [inline, static]

Get the magnitude buffer pointer.

Parameters:
- ← ***fh*** handle to the 'RFFT_F32_STRUCT' object

Returns:
pm pointer to the magnitude buffer

5.6.2.9 static void RFFT_f32_setPhasePtr (RFFT_F32_STRUCT_Handle *fh*, const float * *pp*) [inline, static]

Set the phase buffer pointer.

Parameters:
- ← ***fh*** handle to the 'RFFT_F32_STRUCT' object
- ← ***pp*** pointer to the phase buffer

5.6.2.10 static float* RFFT_f32_getPhasePtr (RFFT_F32_STRUCT_Handle *fh*) [inline, static]

Get the phase buffer pointer.

Parameters:
- ← ***fh*** handle to the 'RFFT_F32_STRUCT' object

Returns:
pp pointer to the phase buffer

5.6.2.11 static void RFFT_f32_setStages (RFFT_F32_STRUCT_Handle *fh*, const uint16_t *st*) [inline, static]

Set the number of FFT stages.

Parameters:
- ← ***fh*** handle to the 'RFFT_F32_STRUCT' object
- ← ***st*** number of FFT stages

5.6.2.12 static uint16_t RFFT_f32_getStages (RFFT_F32_STRUCT_Handle *fh*) `[inline, static]`

Get the number of FFT stages.

Parameters:

← ***fh*** handle to the 'RFFT_F32_STRUCT' object

Returns:

st number of FFT stages

5.6.2.13 static void RFFT_f32_setFFTSize (RFFT_F32_STRUCT_Handle *fh*, const uint16_t *sz*) `[inline, static]`

Set the size of the FFT.

Parameters:

← ***fh*** handle to the 'RFFT_F32_STRUCT' object

← ***sz*** size of the FFT

5.6.2.14 static uint16_t RFFT_f32_getFFTSize (RFFT_F32_STRUCT_Handle *fh*) `[inline, static]`

Get the size of the FFT.

Parameters:

← ***fh*** handle to the 'RFFT_F32_STRUCT' object

Returns:

sz size of the FFT

5.6.2.15 static void RFFT_ADC_f32_setInBufPtr (RFFT_ADC_F32_STRUCT_Handle *fh*, const uint16_t * *pi*) `[inline, static]`

Set the input buffer pointer.

Parameters:

← ***fh*** handle to the 'RFFT_ADC_F32_STRUCT' object

← ***pi*** pointer to the input buffer

5.6.2.16 static uint16_t* RFFT_ADC_f32_getInBufPtr (RFFT_ADC_F32_STRUCT_Handle *fh*) `[inline, static]`

get the input buffer pointer

Parameters:

← ***fh*** handle to the 'RFFT_ADC_F32_STRUCT' object

Returns:
pi pointer to the input buffer

5.6.2.17 static void RFFT_ADC_f32_setTailPtr (RFFT_ADC_F32_STRUCT_Handle *fh*, const void $*$ *pt*) `[inline, static]`

Set the tail pointer.

Parameters:
← ***fh*** handle to the 'RFFT_ADC_F32_STRUCT' object
← ***pt*** pointer to the tail

5.6.2.18 static void$*$ RFFT_ADC_f32_getTailPtr (RFFT_ADC_F32_STRUCT_Handle *fh*) `[inline, static]`

Get the tail pointer.

Parameters:
← ***fh*** handle to the 'RFFT_ADC_F32_STRUCT' object

Returns:
pt pointer to the tail

5.6.2.19 void RFFT_f32 (RFFT_F32_STRUCT_Handle *hndRFFT_F32*)

Real Fast Fourier Transform (RFFT).

This routine computes the 32-bit floating-point FFT for an N-pt ($N = 2^n, n = 5 : 10$) This routine computes the 32-bit single precision FFT for an N-pt ($N = 2^n, n = 5 : 10$) real input. This function reorders the input in bit-reversed format as part of the stage 1,2 and 3 computations. The routine uses two buffers in ping-pong fashion i.e. after each FFT stage the output and input buffers become the input and output buffers respectively for the next stage. This algorithm only allocates memory, and performs computation, for the non-zero elements of the input (the real part). The complex conjugate nature of the spectrum (for real only data) affords savings in space and computaion, therefore the algorithm only calculates the spectrum from the 0th bin to the nyquist bin (included).

Another approach to calculate the real FFT would be to treat the real N-point data as N/2 complex, run the forward complex N/2 point FFT followed by an "unpack" function

Parameters:
hndRFFT_F32 Pointer to the RFFT_F32 object

Attention:
1. The routine requires the use of two buffers, each of size N (32-bit float), for computation; the input buffer must be aligned to a memory address of 2N words (16-bit). Refer to the RFFT linker command file to see an example of this.
2. If alignment is not possible the user can use the alternative, albeit slower, function RFFT_f32u

See also:
Real FFT Using a Complex FFT

Samples	Cycles
64	1281
128	2779
256	6149
512	13674
1024	30356

Table 5.24: Performance Data

5.6.2.20 void RFFT_f32u (RFFT_F32_STRUCT_Handle *hndRFFT_F32*)

Real FFT (Unaligned).

This routine computes the 32-bit single precision FFT for an N-pt ($N = 2^n, n = 5 : 10$) real input. This function reorders the input in bit-reversed format as part of the stage 1,2 and 3 computations. The routine uses two buffers in ping-pong fashion i.e. after each FFT stage the output and input buffers become the input and output buffers respectively for the next stage. This algorithm only allocates memory, and performs computation, for the non-zero elements of the input (the real part). The complex conjugate nature of the spectrum (for real only data) affords savings in space and computaion, therefore the algorithm only calculates the spectrum from the 0th bin to the nyquist bin (included).

Another approach to calculate the real FFT would be to treat the real N-point data as N/2 complex, run the forward complex N/2 point FFT followed by an "unpack" function

Parameters:
hndRFFT_F32 Pointer to the RFFT_F32 object

Attention:
1. The routine requires the use of two buffers, each of size N (32-bit float), for computation; the input buffer need not be aligned to a boundary
2. If alignment is possible it is recommended to use the faster routine, RFFT_f32

See also:
Real FFT Using a Complex FFT

Samples	Cycles
64	1393
128	3003
256	6597
512	14570
1024	32148

Table 5.25: Performance Data

5.6.2.21 void RFFT_adc_f32 (RFFT_ADC_F32_STRUCT_Handle *hndRFFT_ADC_F32*)

Real FFT with ADC Input.

This routine computes the 32-bit single precision FFT for an N-pt ($N = 2^n, n = 5 : 10$) real 12-bit ADC input. This function reorders the input in bit-reversed format as part of the stage 1,2 and 3 computations. The routine uses two buffers in ping-pong fashion i.e. after each FFT stage the output and input buffers become the input and output buffers respectively for the next stage. This algorithm only allocates memory, and performs computation, for the non-zero elements of the input (the real part). The complex conjugate nature of the spectrum (for real only data) affords savings in space and computaion, therefore the algorithm only calculates the spectrum from the 0th bin to the nyquist bin (included).

Another approach to calculate the real FFT would be to treat the real N-point data as N/2 complex, run the forward complex N/2 point FFT followed by an "unpack" function

Parameters:

hndRFFT_ADC_F32 Pointer to the RFFT_ADC F32 object

Attention:

1. The routine requires the use of two buffers, the input of size 2N and type uint16_t, the output of size N and type float, for computation; the input buffer must be aligned to a memory address of N words (16-bit). Refer to the RFFT linker command file to see an example of this.
2. If alignment is not possible the user can use the alternative, albeit slower, function RFFT_adc_f32u

Samples	Cycles
64	1295
128	2769
256	6059
512	13360
1024	29466

Table 5.26: Performance Data

5.6.2.22 void RFFT_adc_f32u (RFFT_ADC_F32_STRUCT_Handle *hndRFFT_ADC_F32*)

Real FFT with ADC Input (Unaligned).

This routine computes the 64-bit double precision FFT for an N-pt ($N = 2^n, n = 5 : 10$) real 12-bit ADC input. This function reorders the input in bit-reversed format as part of the stage 1,2 and 3 computations. The routine uses two buffers in ping-pong fashion i.e. after each FFT stage the output and input buffers become the input and output buffers respectively for the next stage. This algorithm only allocates memory, and performs computation, for the non-zero elements of the input (the real part). The complex conjugate nature of the spectrum (for real only data) affords savings in space and computaion, therefore the algorithm only calculates the spectrum from the 0th bin to the nyquist bin (included).

Another approach to calculate the real FFT would be to treat the real N-point data as N/2 complex, run the forward complex N/2 point FFT followed by an "unpack" function

Parameters:

hndRFFT_ADC_F32 Pointer to the RFFT_ADC F32 object

Attention:

1. The routine requires the use of two buffers, the input of size 2N and type uint16_t, the output of size N and type float, for computation; the input buffer need not be aligned to any boundary.
2. If alignment is possible it is recommended to use the faster function RFFT_adc_f32

Samples	Cycles
64	1415
128	3009
256	6539
512	14320
1024	31386

Table 5.27: Performance Data

5.6.2.23 void RFFT_f32_win (float ∗ *pBuffer*, const float ∗ *pWindow*, const uint16_t *size*)

Windowing function for the 32-bit real FFT.

Parameters:

pBuffer pointer to the buffer that needs to be windowed

pWindow pointer to the windowing table

size size of the buffer This function applies the window to a 2N point real data buffer that has not been reordered in the bit-reversed format

Samples	Cycles
64	308
128	596
256	1172
512	2324
1024	4628

Table 5.28: Performance Data

5.6.2.24 void RFFT_adc_f32_win (uint16_t ∗ *pBuffer*, const uint16_t ∗ *pWindow*, const uint16_t *size*)

Windowing function for the 32-bit real FFT with ADC Input.

Parameters:

pBuffer pointer to the uint16_t buffer that needs to be windowed

pWindow pointer to the float windowing table

size size of the buffer This function applies the window to a 2N point real data buffer that has not been reordered in the bit-reversed format

Attention:

1. The routine requires the window to be unsigned int (16-bit). The user must take the desired floating point window from the header files (e.g. HANN1024 from fpu32/fpu_fft_hann.h)and convert it to Q16 by multiplying by 2^16 and then flooring the value before converting it to uint16_t

5.6.2.25 void RFFT_f32_mag (RFFT_F32_STRUCT_Handle *hndRFFT_F32*)

Real FFT Magnitude.

Samples	Cycles
64	346
128	666
256	1306
512	2586
1024	5146

Table 5.29: Performance Data

This module computes the real FFT magnitude. The output from RFFT_f32_mag matches the magnitude output from the FFT found in common mathematics software and Code Composer Studio FFT graphs. If instead a normalized magnitude like that performed by the fixed-point TMS320C28x IQmath FFT library is required, then the RFFT_f32s_mag function can be used. In fixed-point algorithms scaling is performed to avoid overflowing data. Floating-point calculations do not need this scaling to avoid overflow and therefore the RFFT_f32_mag function can be used instead.

Parameters:

hndRFFT_F32 Pointer to the RFFT_F32 object

Attention:

1. The Magnitude buffer does not require memory alignment to a boundary
2. For C28x devices that have the TMU0 (or higher) module, use RFFT_f32_mag_TMU0() instead for better performance.

Samples	Cycles
64	635
128	1243
256	2459
512	4888
1024	9752

Table 5.30: Performance Data

5.6.2.26 void RFFT_f32s_mag (RFFT_F32_STRUCT_Handle *hndRFFT_F32*)

Real FFT Magnitude (Scaled).

This module computes the scaled real FFT magnitude. The scaling is $\frac{1}{[2^{FFT\_STAGES-1}]}$, and is done to match the normalization performed by the fixed-point TMS320C28x IQmath FFT library for overflow avoidance. Floating-point calculations do not need this scaling to avoid overflow and therefore the RFFT_f32_mag function can be used instead. The output from RFFT_f32s_mag matches the magnitude output from the FFT found in common mathematics software and Code Composer Studio FFT graphs.

Parameters:

hndRFFT_F32 Pointer to the RFFT_F32 object

Attention:

1. The Magnitude buffer does not require memory alignment to a boundary
2. For C28x devices that have the TMU0 (or higher) module, use RFFT_f32s_mag_TMU0() instead for better performance.

Samples	Cycles
64	723
128	1336
256	2557
512	4990
1024	9859

Table 5.31: Performance Data

5.6.2.27 void RFFT_f32_phase (RFFT_F32_STRUCT_Handle *hndRFFT_F32*)

Real FFT Phase.

Parameters:

hndRFFT_F32 Pointer to the RFFT_F32 object

Attention:

1. The Phase buffer does not require memory alignment to a boundary
2. For C28x devices that have the TMU0 (or higher) module, use RFFT_f32_phase_TMU0() instead for better performance.

Samples	Cycles
64	1949
128	3933
256	7901
512	16835
1024	31707

Table 5.32: Performance Data

5.6.2.28 void RFFT_f32_mag_TMU0 (RFFT_F32_STRUCT_Handle *hndRFFT_F32*)

Real FFT Magnitude using the TMU0 module.

This module computes the real FFT magnitude. The output from RFFT_f32_mag matches the magnitude output from the FFT found in common mathematics software and Code Composer Studio FFT graphs. If instead a normalized magnitude like that performed by the fixed-point TMS320C28x IQmath FFT library is required, then the RFFT_f32s_mag function can be used. In fixed-point algorithms scaling is performed to avoid overflowing data. Floating-point calculations do not need this scaling to avoid overflow and therefore the RFFT_f32_mag function can be used instead.

Parameters:

hndRFFT_F32 Pointer to the RFFT_F32 object

Attention:

1. The Magnitude buffer does not require memory alignment to a boundary
2. This function requires a C28x device with TMU0 or higher module. For devices without the TMU, use RFFT_f32_mag() instead.

Samples	Cycles
64	161
128	289
256	545
512	1055
1024	2079

Table 5.33: Performance Data

5.6.2.29 void RFFT_f32s_mag_TMU0 (RFFT_F32_STRUCT_Handle *hndRFFT_F32*)

Real FFT Magnitude using the TMU0 module (Scaled).

This module computes the scaled real FFT magnitude. The scaling is $\frac{1}{[2^{FFT\_STAGES-1}]}$, and is done to match the normalization performed by the fixed-point TMS320C28x IQmath FFT library for overflow avoidance. Floating-point calculations do not need this scaling to avoid overflow and therefore the RFFT_f32_mag function can be used instead. The output from RFFT_f32s_mag matches the magnitude output from the FFT found in common mathematics software and Code Composer Studio FFT graphs.

Parameters:
hndRFFT_F32 Pointer to the RFFT_F32 object

Attention:
1. The Magnitude buffer does not require memory alignment to a boundary
2. This function requires a C28x device with TMU0 or higher module. For devices without the TMU, use RFFT_f32s_mag() instead.

Samples	Cycles
64	268
128	466
256	856
512	1375
1024	2661

Table 5.34: Performance Data

5.6.2.30 void RFFT_f32_phase_TMU0 (RFFT_F32_STRUCT_Handle *hndRFFT_F32*)

Real FFT Phase using TMU0 module.

Parameters:
hndRFFT_F32 Pointer to the RFFT_F32 object

Attention:
1. The Phase buffer does not require memory alignment to a boundary
2. This function requires a C28x device with TMU0 or higher module. For devices without the TMU, use RFFT_f32_phase() instead.

Samples	Cycles
64	279
128	535
256	1047
512	2068
1024	4116

Table 5.35: Performance Data

5.6.2.31 void RFFT_f32_sincostable (RFFT_F32_STRUCT_Handle *hndRFFT_F32*)

Generate twiddle factors for the Real FFT.

Parameters:

hndRFFT_F32 Pointer to the RFFT_F32 object

Attention:

This function is written in C and compiled without optimization turned on.

5.6.3 Variable Documentation

5.6.3.1 float RFFT_f32_twiddleFactors[1020]

Twiddle Factor Table for a 2048-pt (max) Real FFT.

Note: 1. RFFT_f32_twiddleFactors name is deprecated and only supported for legacy reasons. Users are encouraged to use FPU32RFFTtwiddleFactors as the table symbol. 2. The RFFT_f32_twiddleFactors is an alias for the new table. It is not a separate table.

5.7 Filters

Data Structures

- FIR_f32
- IIR_f32

Defines

- FIR_FP
- FIR_FP_Handle
- FIR_FP_init
- FIR_FP_calc
- FIR_f32_DEFAULTS

Functions

- static void FIR_f32_setCoefficientsPtr (FIR_f32_Handle fh, const float ∗pc)
- static float ∗ FIR_f32_getCoefficientsPtr (FIR_f32_Handle fh)
- static void FIR_f32_setDelayLinePtr (FIR_f32_Handle fh, const float ∗pdl)
- static float ∗ FIR_f32_getDelayLinePtr (FIR_f32_Handle fh)
- static void FIR_f32_setInput (FIR_f32_Handle fh, const float in)
- static float FIR_f32_getInput (FIR_f32_Handle fh)
- static void FIR_f32_setOutput (FIR_f32_Handle fh, const float out)
- static float FIR_f32_getOutput (FIR_f32_Handle fh)
- static void FIR_f32_setOrder (FIR_f32_Handle fh, const uint16_t order)
- static uint16_t FIR_f32_getOrder (FIR_f32_Handle fh)
- static void FIR_f32_setInitFunction (FIR_f32_Handle fh, const v_pfn_v pfn)
- static v_pfn_v FIR_f32_getInitFunction (FIR_f32_Handle fh)
- static void FIR_f32_setCalcFunction (FIR_f32_Handle fh, const v_pfn_v pfn)
- static v_pfn_v FIR_f32_getCalcFunction (FIR_f32_Handle fh)
- static void IIR_f32_setCoefficientsAPtr (IIR_f32_Handle fh, const float ∗pca)
- static float ∗ IIR_f32_getCoefficientsAPtr (IIR_f32_Handle fh)
- static void IIR_f32_setCoefficientsBPtr (IIR_f32_Handle fh, const float ∗pcb)
- static float ∗ IIR_f32_getCoefficientsBPtr (IIR_f32_Handle fh)
- static void IIR_f32_setDelayLinePtr (IIR_f32_Handle fh, const float ∗pdl)
- static float ∗ IIR_f32_getDelayLinePtr (IIR_f32_Handle fh)
- static void IIR_f32_setInputPtr (IIR_f32_Handle fh, const float ∗pi)
- static float ∗ IIR_f32_getInputPtr (IIR_f32_Handle fh)
- static void IIR_f32_setOutputPtr (IIR_f32_Handle fh, const float ∗po)
- static float ∗ IIR_f32_getOutputPtr (IIR_f32_Handle fh)
- static void IIR_f32_setScalePtr (IIR_f32_Handle fh, const float ∗psv)
- static float ∗ IIR_f32_getScalePtr (IIR_f32_Handle fh)
- static void IIR_f32_setOrder (IIR_f32_Handle fh, const uint16_t order)

- static uint16_t IIR_f32_getOrder (IIR_f32_Handle fh)
- static void IIR_f32_setInitFunction (IIR_f32_Handle fh, const v_pfn_v pfn)
- static v_pfn_v IIR_f32_getInitFunction (IIR_f32_Handle fh)
- static void IIR_f32_setCalcFunction (IIR_f32_Handle fh, const v_pfn_v pfn)
- static v_pfn_v IIR_f32_getCalcFunction (IIR_f32_Handle fh)
- void FIR_f32_calc (FIR_f32_Handle hndFIR_f32)
- void FIR_f32_init (FIR_f32_Handle hndFIR_f32)
- void IIR_f32_calc (IIR_f32_Handle hndIIR_f32)
- void IIR_f32_init (IIR_f32_Handle hndIIR_f32)

5.7.1 Data Structure Documentation

5.7.1.1 FIR_f32

Definition:

```
typedef struct
{
    float *coeff_ptr;
    float *dbuffer_ptr;
    int16_t cbindex;
    int16_t order;
    float input;
    float output;
    void (*init)(void *);
    void (*calc)(void *);
}
FIR_f32
```

Members:

coeff_ptr Pointer to Filter coefficient.

dbuffer_ptr Delay buffer pointer.

cbindex Circular Buffer Index.

order Order of the Filter.

input Latest Input sample.

output Filter Output.

init Pointer to Initialization function.

calc Pointer to the calculation function.

Description:

Structure for the Finite Impulse Response Filter

5.7.1.2 IIR_f32

Definition:

```
typedef struct
{
    float *p_coeff_A;
    float *p_coeff_B;
```

```
        float *p_dbuffer;
        float *p_input;
        float *p_output;
        float *p_scale;
        uint16_t order;
        void (*init)(void *);
        void (*calc)(void *);
    }
    IIR_f32
```

Members:

p_coeff_A Pointer to the denominator coefficients.
p_coeff_B Pointer to the numerator coefficients.
p_dbuffer Delay buffer pointer.
p_input Pointer to the latest input sample.
p_output Pointer to the filter output.
p_scale Pointer to the biquad(s) scale values.
order Order of the filter.
init Pointer to the initialization function.
calc Pointer to the calculation function.

Description:

Structure defintion for the Infinite Impulse Response Filter

5.7.2 Define Documentation

5.7.2.1 FIR_FP

Definition:

```
#define
```

Description:

FIR_FP is the legacy name of the FIR structure.

5.7.2.2 FIR_FP_Handle

Definition:

```
#define
```

Description:

FIR_FP_Handle is the legacy name of the FIR structure handle.

5.7.2.3 FIR_FP_init

Definition:

```
#define FIR_FP_init
```

Description:

FIR_FP_init is the legacy name of the FIR initialization function.

5.7.2.4 FIR_FP_calc

Definition:
```
#define FIR_FP_calc
```

Description:
FIR_FP_calc is the legacy name of the FIR calculation function.

5.7.2.5 FIR_f32_DEFAULTS

Definition:
```
#define FIR_f32_DEFAULTS
```

Description:
The default FIR object initializer.

5.7.3 Typedef Documentation

5.7.3.1 FIR_f32_Handle

Definition:
```
typedef FIR_f32 *FIR_f32_Handle
```

Description:
Handle to the Filter Structure Object.

5.7.3.2 IIR_f32_Handle

Definition:
```
typedef IIR_f32 *IIR_f32_Handle
```

Description:
Handle to the Filter Structure Object.

5.7.4 Function Documentation

5.7.4.1 FIR_f32_setCoefficientsPtr

Set the coefficients pointer.

Prototype:
```
static void
FIR_f32_setCoefficientsPtr(FIR_f32_Handle fh,
                           const float *pc) [inline, static]
```

Parameters:
← ***fh*** handle to the 'FIR_f32' object
← ***pc*** pointer to the coefficients

5.7.4.2 static float* FIR_f32_getCoefficientsPtr (FIR_f32_Handle *fh*) [inline, static]

Get the coefficients pointer.

Parameters:
← ***fh*** handle to the 'FIR_f32' object

Returns:
pc pointer to the coefficients

5.7.4.3 static void FIR_f32_setDelayLinePtr (FIR_f32_Handle *fh*, const float * *pdl*) [inline, static]

Set the delay line pointer.

Parameters:
← ***fh*** handle to the 'FIR_f32' object
← ***pdl*** pointer to the delay line

5.7.4.4 static float* FIR_f32_getDelayLinePtr (FIR_f32_Handle *fh*) [inline, static]

Get the delay line pointer.

Parameters:
← ***fh*** handle to the 'FIR_f32' object

Returns:
pdl pointer to the delay line

5.7.4.5 static void FIR_f32_setInput (FIR_f32_Handle *fh*, const float *in*) [inline, static]

Set the input.

Parameters:
← ***fh*** handle to the 'FIR_f32' object
← ***in*** current input

5.7.4.6 static float FIR_f32_getInput (FIR_f32_Handle *fh*) [inline, static]

Get the input.

Parameters:
← ***fh*** handle to the 'FIR_f32' object

Returns:
pin current input pointer

5.7.4.7 static void FIR_f32_setOutput (FIR_f32_Handle *fh*, const float *out*) [inline, static]

Set the output.

Parameters:
← ***fh*** handle to the 'FIR_f32' object
← ***out*** current output

5.7.4.8 static float FIR_f32_getOutput (FIR_f32_Handle *fh*) [inline, static]

Get the output.

Parameters:
← ***fh*** handle to the 'FIR_f32' object

Returns:
out current output

5.7.4.9 static void FIR_f32_setOrder (FIR_f32_Handle *fh*, const uint16_t *order*) [inline, static]

Set the order of the filter.

Parameters:
← ***fh*** handle to the 'FIR_f32' object
← ***order*** Order of the filter

5.7.4.10 static uint16_t FIR_f32_getOrder (FIR_f32_Handle *fh*) [inline, static]

Get the order of the filter.

Parameters:
← ***fh*** handle to the 'FIR_f32' object

Returns:
order Order of the filter

5.7.4.11 static void FIR_f32_setInitFunction (FIR_f32_Handle *fh*, const v_pfn_v *pfn*) [inline, static]

Set the init function.

Parameters:
← ***fh*** handle to the 'FIR_f32' object
← ***pfn*** pointer to the init function

5.7.4.12 static v_pfn_v FIR_f32_getInitFunction (FIR_f32_Handle *fh*) [inline, static]

Get the init function.

Parameters:

← ***fh*** handle to the 'FIR_f32' object

Returns:

pfn pointer to the init function

5.7.4.13 static void FIR_f32_setCalcFunction (FIR_f32_Handle *fh*, const v_pfn_v *pfn*) [inline, static]

Set the calc function.

Parameters:

← ***fh*** handle to the 'FIR_f32' object

← ***pfn*** pointer to the calc function

5.7.4.14 static v_pfn_v FIR_f32_getCalcFunction (FIR_f32_Handle *fh*) [inline, static]

Get the calc function.

Parameters:

← ***fh*** handle to the 'FIR_f32' object

Returns:

pfn pointer to the calc function

5.7.4.15 static void IIR_f32_setCoefficientsAPtr (IIR_f32_Handle *fh*, const float ∗ *pca*) [inline, static]

Set the denominator coefficients pointer.

Parameters:

← ***fh*** handle to the 'IIR_f32' object

← ***pca*** pointer to the denominator coefficients

5.7.4.16 static float∗ IIR_f32_getCoefficientsAPtr (IIR_f32_Handle *fh*) [inline, static]

Get the denominator coefficients pointer.

Parameters:

← ***fh*** handle to the 'IIR_f32' object

Returns:

pca pointer to the denominator coefficients

5.7.4.17 static void IIR_f32_setCoefficientsBPtr (IIR_f32_Handle *fh*, const float ∗ *pcb*) `[inline, static]`

Set the numerator coefficients pointer.

Parameters:

← ***fh*** handle to the 'IIR_f32' object

← ***pcb*** pointer to the numerator coefficients

5.7.4.18 static float∗ IIR_f32_getCoefficientsBPtr (IIR_f32_Handle *fh*) `[inline, static]`

Get the numerator coefficients pointer.

Parameters:

← ***fh*** handle to the 'IIR_f32' object

Returns:

pca pointer to the numerator coefficients

5.7.4.19 static void IIR_f32_setDelayLinePtr (IIR_f32_Handle *fh*, const float ∗ *pdl*) `[inline, static]`

Set the delay line pointer.

Parameters:

← ***fh*** handle to the 'IIR_f32' object

← ***pdl*** pointer to the delay line

5.7.4.20 static float∗ IIR_f32_getDelayLinePtr (IIR_f32_Handle *fh*) `[inline, static]`

Get the delay line pointer.

Parameters:

← ***fh*** handle to the 'IIR_f32' object

Returns:

pdl pointer to the delay line

5.7.4.21 static void IIR_f32_setInputPtr (IIR_f32_Handle *fh*, const float ∗ *pi*) [inline, static]

Set the input pointer.

Parameters:
← ***fh*** handle to the 'IIR_f32' object
← ***pi*** pointer to the current input

5.7.4.22 static float∗ IIR_f32_getInputPtr (IIR_f32_Handle *fh*) [inline, static]

Get the input pointer.

Parameters:
← ***fh*** handle to the 'IIR_f32' object

Returns:
pi pointer to the current input

5.7.4.23 static void IIR_f32_setOutputPtr (IIR_f32_Handle *fh*, const float ∗ *po*) [inline, static]

Set the output pointer.

Parameters:
← ***fh*** handle to the 'IIR_f32' object
← ***po*** pointer to the current output

5.7.4.24 static float∗ IIR_f32_getOutputPtr (IIR_f32_Handle *fh*) [inline, static]

Get the output pointer.

Parameters:
← ***fh*** handle to the 'IIR_f32' object

Returns:
po pointer to the current output

5.7.4.25 static void IIR_f32_setScalePtr (IIR_f32_Handle *fh*, const float ∗ *psv*) [inline, static]

Set the scale value pointer.

Parameters:
← ***fh*** handle to the 'IIR_f32' object
← ***psv*** pointer to the scale values for the biquads

5.7.4.26 static float* IIR_f32_getScalePtr (IIR_f32_Handle *fh*) [inline, static]

Get the scale value pointer.

Parameters:

← ***fh*** handle to the 'IIR_f32' object

Returns:

psv pointer to the scale values for the biquads

5.7.4.27 static void IIR_f32_setOrder (IIR_f32_Handle *fh*, const uint16_t *order*) [inline, static]

Set the order of the filter.

Parameters:

← ***fh*** handle to the 'IIR_f32' object
← ***order*** Order of the filter

5.7.4.28 static uint16_t IIR_f32_getOrder (IIR_f32_Handle *fh*) [inline, static]

Get the order of the filter.

Parameters:

← ***fh*** handle to the 'IIR_f32' object

Returns:

order Order of the filter

5.7.4.29 static void IIR_f32_setInitFunction (IIR_f32_Handle *fh*, const v_pfn_v *pfn*) [inline, static]

Set the init function.

Parameters:

← ***fh*** handle to the 'IIR_f32' object
← ***pfn*** pointer to the init function

5.7.4.30 static v_pfn_v IIR_f32_getInitFunction (IIR_f32_Handle *fh*) [inline, static]

Get the init function.

Parameters:

← ***fh*** handle to the 'IIR_f32' object

Returns:

pfn pointer to the init function

5.7.4.31 static void IIR_f32_setCalcFunction (IIR_f32_Handle *fh*, const v_pfn_v *pfn*) [inline, static]

Set the calc function.

Parameters:

← ***fh*** handle to the 'IIR_f32' object
← ***pfn*** pointer to the calc function

5.7.4.32 static v_pfn_v IIR_f32_getCalcFunction (IIR_f32_Handle *fh*) [inline, static]

Get the calc function.

Parameters:

← ***fh*** handle to the 'IIR_f32' object

Returns:

pfn pointer to the calc function

5.7.4.33 void FIR_f32_calc (FIR_f32_Handle *hndFIR_f32*)

Finite Impulse Response Filter.

This routine implements the non-recursive difference equation of an all-zero filter (FIR), of order N. All the coefficients of all-zero filter are assumed to be less than 1 in magnitude.

Parameters:

hndFIR_f32 Handle to the FIR_f32 object

Attention:

1. The delay and coefficients buffer must be aligned to a minimum of 2 x (order + 1) words. For example, if the filter order is 31, it will have 32 taps or coefficients each a 32-bit floating point value. A minimum of $(2 * 32) = 64$ words will need to be allocated for the delay and coefficients buffer.
2. To align the buffer, use the DATA_SECTION pragma to assign the buffer to a code section and then align the buffer to the proper offset in the linker command file. In the code example the buffer is assigned to the **firldb** section while the coefficients are assigned to the **coefffilt** section.
3. This routine requires the –c2xlp_src_compatible option to be enabled in the file specific properties when accepting 24x assembly instructions. This option is ONLY used if you are migrating C24x code to the C28x, and is not available for compilers after v6.

Number of Taps (Order + 1)	Cycles
28	103
59	165
117	281

Table 5.36: Performance Data

5.7.4.34 void FIR_f32_init (FIR_f32_Handle *hndFIR_f32*)

Finite Impulse Response Filter Initialization.

Zeros out the delay line

Parameters:

hndFIR_f32 Handle to the FIR_f32 object

Attention:

1. The delay and coefficients buffer must be aligned to a minimum of 2 x (order + 1) words. For example, if the filter order is 31, it will have 32 taps or coefficients each a 32-bit floating point value. A minimum of (2 $*$ 32) = 64 words will need to be allocated for the delay and coefficients buffer.
2. The delay buffer needs to be aligned to word boundary of 2 $*$ number of taps
3. To align the buffer, use the DATA_SECTION pragma to assign the buffer to a code section and then align the buffer to the proper offset in the linker command file. In the code example the buffer is assigned to the **firldb** section while the coefficients are assigned to the **coefffilt** section.

Number of Taps (Order + 1)	Cycles
28	79
59	141
117	257

Table 5.37: Performance Data

5.7.4.35 void IIR_f32_calc (IIR_f32_Handle *hndIIR_f32*)

Infinite Impulse Response Filter.

This routine implements the Transposed Direct form II recursive difference equation of an N pole-zero filter(IIR).

Parameters:

hndIIR_f32 Handle to the IIR_f32 object

Attention:

1. The delay line buffer must be 2$*$(n_biquads$*$n_delay_elements_per_biquad), since there are 4 delay elements per biquad that are single precision (32-bits) we require a total of 8$*$n_biquads words For example, if the filter is an 8th order filter it would require 4 biquads (each biqaud is a 2nd order construct) hence 8$*$4 = 32 words If the filter were a 9th order filter, it would require 5 biquads; the first four would be quadratic while the last is linear. The last biquad will be implemented with the B[2] and A[2] coefficients zero. We would require a total of 8$*$5 = 40 words
2. In the code example the buffer is assigned to the .ebss section while the coefficients are assigned to the .econst section.

5.7.4.36 void IIR_f32_init (IIR_f32_Handle *hndIIR_f32*)

Infinite Impulse Response Filter Initialization.

Zeros out the delay line

Filter Order	Number of Biquads	Cycles
2	1	68
6	3	116
12	6	188

Table 5.38: Performance Data

Parameters:

hndIIR_f32 Handle to the IIR_f32 object

Attention:

Please see the description of IIR_f32_calc for more details on the space requirements for the delay line and coefficients

Filter Order	Number of Biquads	Cycles
2	1	30
6	3	46
12	6	70

Table 5.39: Performance Data

5.8 Vector Operations

Functions

- void abs_SP_CV (float ∗y, const complex_float ∗x, const uint16_t N)
- void abs_SP_CV_2 (float ∗y, const complex_float ∗x, const uint16_t N)
- void abs_SP_CV_TMU0 (float ∗y, const complex_float ∗x, const uint16_t N)
- void add_SP_CSxCV (complex_float ∗y, const complex_float ∗x, const complex_float c, const uint16_t N)
- void add_SP_CVxCV (complex_float ∗y, const complex_float ∗w, const complex_float ∗x, const uint16_t N)
- void iabs_SP_CV (float ∗y, const complex_float ∗x, const uint16_t N)
- void iabs_SP_CV_2 (float ∗y, const complex_float ∗x, const uint16_t N)
- void iabs_SP_CV_TMU0 (float ∗y, const complex_float ∗x, const uint16_t N)
- complex_float mac_SP_CVxCV (const complex_float ∗w, const complex_float ∗x, const uint16_t N)
- complex_float mac_SP_RVxCV (const complex_float ∗w, const float ∗x, const uint16_t N)
- complex_float mac_SP_i16RVxCV (const complex_float ∗w, const int16_t ∗x, const uint16_t N)
- uint16_t maxidx_SP_RV_2 (const float ∗x, const uint16_t N)
- complex_float mean_SP_CV_2 (const complex_float ∗x, const uint16_t N)
- float median_noreorder_SP_RV (const float ∗x, const uint16_t N)
- float median_SP_RV (float ∗x, const uint16_t N)
- void memcpy_fast (void ∗dst, const void ∗src, const uint16_t N)
- void memset_fast (void ∗dst, const int16_t value, const uint16_t N)
- complex_float mpy_SP_CSxCS (const complex_float w, const complex_float x)
- void mpy_SP_CVxCV (complex_float ∗y, const complex_float ∗w, const complex_float ∗x, const uint16_t N)
- void mpy_SP_CVxCVC (complex_float ∗y, const complex_float ∗w, const complex_float ∗x, const uint16_t N)
- void mpy_SP_RMxRM (float ∗y, const float ∗w, const float ∗x, const uint16_t m, const uint16_t n, const uint16_t p)
- void mpy_SP_RMxRM_2 (float ∗y, const float ∗w, const float ∗x, const uint16_t m, const uint16_t n, const uint16_t p)
- void mpy_SP_RSxRV_2 (float ∗y, const float ∗x, const float c, const uint16_t N)
- void mpy_SP_RSxRVxRV_2 (float ∗y, const float ∗w, const float ∗x, const float c, const uint16_t N)
- void mpy_SP_RVxCV (complex_float ∗y, const complex_float ∗w, const float ∗x, const uint16_t N)
- void mpy_SP_RVxRV_2 (float ∗y, const float ∗w, const float ∗x, const uint16_t N)
- void qsort_SP_RV (void ∗x, const uint16_t N)
- float rnd_SP_RS (const float x)
- void sub_SP_CSxCV (complex_float ∗y, const complex_float ∗x, const complex_float c, const uint16_t N)
- void sub_SP_CVxCV (complex_float ∗y, const complex_float ∗w, const complex_float ∗x, const uint16_t N)

5.8.1 Function Documentation

5.8.1.1 abs_SP_CV

Absolute Value of a Complex Vector.

Prototype:

```
void
abs_SP_CV(float *y,
          const complex_float *x,
          const uint16_t N)
```

Description:

This module computes the absolute value of a complex vector. If N is even, use abs_SP_CV_2() for better performance.

$$y[i] = \sqrt{(x_{re}[i]^2 + x_{im}[i]^2)}$$

Parameters:

y pointer to the output vector

x pointer to the input vector

N length of the x and y vectors

Cycles	Comment
28∗N + 9	Cycle count includes the call and return

Table 5.40: Performance Data

5.8.1.2 void abs_SP_CV_2 (float ∗ *y*, const complex_float ∗ *x*, const uint16_t *N*)

Absolute Value of an Even Length Complex Vector.

This module computes the absolute value of an even length complex vector.

$$y[i] = \sqrt{(x_{re}[i]^2 + x_{im}[i]^2)}$$

Parameters:

y pointer to the output vector

x pointer to the input vector

N length of the x and y vectors

Attention:

N must be even

Cycles	Comment
18∗N + 22	Cycle count includes the call and return

Table 5.41: Performance Data

5.8.1.3 abs_SP_CV_TMU0

Absolute Value of a Complex Vector (TMU0).

Prototype:

```
void
abs_SP_CV_TMU0(float *y,
               const complex_float *x,
               const uint16_t N)
```

Description:

This module computes the absolute value of a complex vector. It uses the TMU Type 0 accelerator to speed up the calculation of square roots.

$$y[i] = \sqrt{(}x_{re}[i]^2 + x_{im}[i]^2)$$

Parameters:

y pointer to the output vector

x pointer to the input vector

N length of the x and y vectors

Attention:

1. This function is optimized for N>=8. It is less cycle efficient when N<8. For very small N (e.g., N=1, 2, maybe 3) the user might consider using the TMU intrinsics in the compiler instead of this function.

Cycles	Comment
	Cycle count includes the call and return
30	N = 1 (N - vector size)
7.5∗(N)+21	1 < N < 8 and N even
7.5∗(N-1)+38	1 < N < 8 and N odd
4∗(N-6)+56	N >= 8 and N even
4∗(N-7)+73	N >= 8 and N odd

Table 5.42: Performance Data

5.8.1.4 add_SP_CSxCV

Addition (Element-Wise) of a Complex Scalar to a Complex Vector.

Prototype:

```
void
add_SP_CSxCV(complex_float *y,
             const complex_float *x,
             const complex_float c,
             const uint16_t N)
```

Description:

This module adds a complex scalar element-wise to a complex vector.

$$y_{re}[i] = x_{re}[i] + c_{re}$$

$$y_{im}[i] = x_{im}[i] + c_{im}$$

Parameters:

y pointer to the complex output vector
x pointer to the complex input vector
c complex input scalar
N length of the x and y vectors

Cycles	Comment
4∗N + 19	Cycle count includes the call and return (COFF)
4∗N + 16	Cycle count includes the call and return (EABI)

Table 5.43: Performance Data

5.8.1.5 void add_SP_CVxCV (complex_float ∗ *y*, const complex_float ∗ *w*, const complex_float ∗ *x*, const uint16_t *N*)

Addition of Two Complex Vectors.

This module adds two complex vectors.

$$y_{re}[i] = w_{re}[i] + x_{re}[i]$$

$$y_{im}[i] = w_{im}[i] + x_{im}[i]$$

Parameters:

y pointer to the complex output vector
w pointer to the first complex input vector
x pointer to the second complex input vector
N length of the w x and y vectors

Cycles	Comment
6∗N + 15	Cycle count includes the call and return

Table 5.44: Performance Data

5.8.1.6 void iabs_SP_CV (float ∗ *y*, const complex_float ∗ *x*, const uint16_t *N*)

Inverse Absolute Value of a Complex Vector.

This module computes the inverse absolute value of a complex vector.

$$y[i] = \frac{1}{\sqrt{(x_{re}[i]^2 + x_{im}[i]^2)}}$$

Parameters:

y pointer to the output vector
x pointer to the input vector

N length of the x and y vectors

Attention:
N must be at least 2

Cycles	Comment
25∗N + 13	Cycle count includes the call and return

Table 5.45: Performance Data

5.8.1.7 iabs_SP_CV_2

Inverse Absolute Value of an Even Length Complex Vector.

Prototype:

```
void
iabs_SP_CV_2(float *y,
             const complex_float *x,
             const uint16_t N)
```

Description:
This module calculates the inverse absolute value of an even length complex vector.

$$y[i] = \frac{1}{\sqrt{(x_{re}[i]^2 + x_{im}[i]^2)}}$$

Parameters:

y pointer to the output vector
x pointer to the input vector
N length of the x and y vectors

Attention:
N must be even

Cycles	Comment
15∗N + 22	Cycle count includes the call and return

Table 5.46: Performance Data

5.8.1.8 iabs_SP_CV_TMU0

Inverse Absolute Value of a Complex Vector (TMU0).

Prototype:

```
void
iabs_SP_CV_TMU0(float *y,
                const complex_float *x,
                const uint16_t N)
```

Description:

This module computes the inverse absolute value of a complex vector. It uses the TMU Type 0 accelerator to speed up the calculation of square roots.

$$y[i] = \frac{1}{\sqrt{(x_{re}[i]^2 + x_{im}[i]^2)}}$$

Parameters:

y pointer to the output vector

x pointer to the input vector

N length of the x and y vectors

Attention:

1. This function is optimized for N>=8. It is less cycle efficient when N<8. For very small N (e.g., N=1, 2, maybe 3) the user might consider using the TMU intrinsics in the compiler instead of this function.

Cycles	Comment
	Cycle count includes the call and return
35	N = 1 (N - vector size)
10∗(N)+24	1 < N < 8 and N even
10∗(N-1)+46	1 < N < 8 and N odd
5∗(N-6)+67	N >= 8 and N even
5∗(N-7)+89	N >= 8 and N odd

Table 5.47: Performance Data

5.8.1.9 mac_SP_CVxCV

Multiply-and-Accumulate of a Complex Vector and a Complex Vector.

Prototype:

```
complex_float
mac_SP_CVxCV(const complex_float *w,
             const complex_float *x,
             const uint16_t N)
```

Description:

This module multiplies and accumulates a complex vector and another complex vector.

$$y_{re} = \sum(w_{re}[i] * x_{re}[i] - w_{im}[i] * x_{im}[i])$$

$$y_{im} = \sum(w_{re}[i] * x_{im}[i] + w_{im}[i] * x_{re}[i])$$

Parameters:

y complex result

w pointer to the first complex input vector

x pointer to the second complex input vector

N length of the w and x vectors

Attention:

N must be a minimum of 3

Cycles	Comment
5∗N + 24	Cycle count includes the call and return

Table 5.48: Performance Data

5.8.1.10 mac_SP_RVxCV

Multiply-and-Accumulate of a Real Vector and a Complex Vector.

Prototype:

```
complex_float
mac_SP_RVxCV(const complex_float *w,
             const float *x,
             const uint16_t N)
```

Description:

This module multiplies and accumulates a real vector and a complex vector.

$$y_{re} = \sum(x[i] * w_{re}[i])$$

$$y_{im} = \sum(x[i] * w_{im}[i])$$

Parameters:

y complex result

w pointer to the complex input vector

x pointer to the real input vector

N length of the w and x vectors

Attention:

N must be a minimum of 5

Cycles	Comment
3∗N + 27	Cycle count includes the call and return

Table 5.49: Performance Data

5.8.1.11 mac_SP_i16RVxCV

Multiply-and-Accumulate of a Real Vector and a Complex Vector.

Prototype:

```
complex_float
mac_SP_i16RVxCV(const complex_float *w,
                const int16_t *x,
                const uint16_t N)
```

Description:

This module multiplies and accumulates a 16-bit integer real vector and a floating pt. complex vector.

$$y_{re} = sum(x[i] * w_{re}[i])$$

$$y_{im} = sum(x[i] * w_{im}[i])$$

Parameters:

w pointer to the complex input vector

x pointer to the real input vector

N length of the w and x vectors

Returns:

complex floating pt. accumulation result

Attention:

N must be a minimum of 5

Cycles	Comment
	Cycle count includes the call and return
3∗N + 28	N is even
3∗N + 29	N is odd

Table 5.50: Performance Data

5.8.1.12 maxidx_SP_RV_2

Index of Maximum Value of an Even Length Real Array.

Prototype:

```
uint16_t
maxidx_SP_RV_2(const float *x,
               const uint16_t N)
```

Parameters:

x pointer to the input vector

N length of the x vector

Attention:

1. N must be even
2. If more than one instance of the max value exists in x[], the function will return the index of the first occurrence (lowest index value)

Cycles	Comment
3∗N + 21	Cycle count includes the call and return

Table 5.51: Performance Data

5.8.1.13 complex_float mean_SP_CV_2 (const complex_float ∗ *x*, const uint16_t *N*)

Mean of Real and Imaginary Parts of a Complex Vector.

This module calculates the mean of real and imaginary parts of a complex vector.

$$y_{re} = \frac{\Sigma x_{re}}{N}$$

$$y_{im} = \frac{\Sigma x_{im}}{N}$$

Parameters:

x pointer to the input vector

N length of the x vector

Attention:

N must be even and a minimum of 4

Cycles	Comment
2∗N + 34	Cycle count includes the call and return

Table 5.52: Performance Data

5.8.1.14 float median_noreorder_SP_RV (const float ∗ *x*, const uint16_t *N*)

Median of a Real Valued Array of Floats (Preserved Inputs).

This module computes the median of a real valued array of floats. The input array is preserved. If input array preservation is not required, use median_SP_RV() for better performance. This function calls median_SP_RV() and memcpy_fast().

Parameters:

x pointer to the real input vector

N length of the x vector

Attention:

This function simply makes a local copy of the input array, and then calls median_SP_CV() using the copy

The length of the copy of the input array is allocated at compile time by the constant "K" defined in the code. If the passed parameter N is greater than K memory corruption will result. Be sure to recompile the library with an appropriate value $K >= N$ before executing this code. The library uses K = 256 as the default value.

The first stage of this function (memory copy) is not interruptible.

Returns:

median of the vector x

Cycles	Comment
N/A	This is a C function

Table 5.53: Performance Data

5.8.1.15 float median_SP_RV (float ∗ *x*, const uint16_t *N*)

Median of a real array of floats.

This module computes the median of a real array of floats. The Input array is NOT preserved. If input array preservation is required, use median_noreorder_SP_RV().

Parameters:

x pointer to the real input vector

N length of the x vector

Attention:

1. This function is destructive to the input array x in that it will be sorted during function execution. If this is not allowable, use median_noreorder_SP_CV().
2. This function should be compiled with -o4, -mf5, and no -g compiler options for best performance.

Returns:

median of the vector x

Cycles	Comment
N/A	This is a C function

Table 5.54: Performance Data

5.8.1.16 void memcpy_fast (void ∗ *dst*, const void ∗ *src*, const uint16_t *N*)

Optimized Memory Copy.

Parameters:

src pointer to the source buffer

dst pointer to the destination buffer

N size (16-bits) of the buffer to be copied

Attention:

The function checks for the case of N=0 and just returns if true

This function is not interruptible. Use memcpy_fast_far instead for interruptibility.

This function does not support memory above 22 bits address. For input data above 22 bits address, use memcpy_fast_far instead.

Cycles	Comment
N + 20	Cycle count includes the call and return

Table 5.55: Performance Data

5.8.1.17 void memset_fast (void ∗ *dst*, const int16_t *value*, const uint16_t *N*)

Optimized Memory Set.

Parameters:

dst pointer to the destination buffer

value value to write to all the buffer locations

N size (16-bits) of the buffer to be written

Attention:

The function checks for the case of N=0 and just returns if true

This function is not interruptible

Cycles	Comment
N + 20	Cycle count includes the call and return

Table 5.56: Performance Data

5.8.1.18 complex_float mpy_SP_CSxCS (const complex_float *w*, const complex_float *x*)

Complex Multiply of Two Floating Point Numbers.

This module multiplies two floating point complex values.

$$y_{re} = w_{re} * x_{re} - w_{im} * x_{im}$$

$$y_{im} = w_{re} * x_{im} + w_{im} * x_{re}$$

Parameters:

w First complex input

x Second complex input

Returns:

complex product of the first and second complex input

Cycles	Comment
19	Cycle count includes the call and return (COFF)
17	Cycle count includes the call and return (EABI)

Table 5.57: Performance Data

5.8.1.19 void mpy_SP_CVxCV (complex_float ∗ *y*, const complex_float ∗ *w*, const complex_float ∗ *x*, const uint16_t *N*)

Complex Multiply of Two Complex Vectors.

This module performs complex multiplication on two input complex vectors.

$$y_{re}[i] = w_{re}[i] * x_{re}[i] - w_{im}[i] * x_{im}[i]$$

$$y_{im}[i] = w_{re}[i] * x_{im}[i] + w_{im}[i] * x_{re}[i]$$

Parameters:

y pointer to the complex product of the first and second complex vectors

w pointer to the first complex input vector

x pointer to the second complex input vector

N length of the w x and y vectors

Cycles	Comment
10∗N + 16	Cycle count includes the call and return

Table 5.58: Performance Data

5.8.1.20 void mpy_SP_CVxCVC (complex_float ∗ *y*, const complex_float ∗ *w*, const complex_float ∗ *x*, const uint16_t *N*)

Multiplication of a Complex Vector and the Complex Conjugate of another Vector.

This module multiplies a complex vector (w) and the complex conjugate of another complex vector (x).

$$x^*_{re}[i] = x_{re}[i]$$
$$x^*_{im}[i] = -x_{im}[i]$$
$$y_{re}[i] = w_{re}[i] * x_{re}[i] - w_{im}[i] * x^*_{im}[i]$$
$$y_{im}[i] = w_{re}[i] * x^*_{im}[i] + w_{im}[i] * x_{re}[i]$$

Parameters:
- ***y*** pointer to the complex conjugate product of the first and second complex vectors
- ***w*** pointer to the first complex input vector
- ***x*** pointer to the second complex input vector
- ***N*** length of the w x and y vectors

Cycles	Comment
11∗N + 16	Cycle count includes the call and return

Table 5.59: Performance Data

5.8.1.21 void mpy_SP_RMxRM (float ∗ *y*, const float ∗ *w*, const float ∗ *x*, const uint16_t *m*, const uint16_t *n*, const uint16_t *p*)

Multiplication of Two Real Matrices.

This module multiplies two real matrices.

$$y[] = w[] * x[]$$

Parameters:
- ***y*** pointer to result matrix
- ***w*** pointer to 1st source matrix
- ***x*** pointer to 2nd source matrix
- ***m*** number of rows in the first and output matrices
- ***n*** number of columns in the first and rows in the second matrix
- ***p*** number of columns in the second and output matrices

Attention:
1. There are no restrictions on the values for n, m, and pwith this function.
2. If n is even and at least 4, you can use mpy_SP_RMxRM_2() for better performance if desired.

Cycles	Comment
5∗m∗n∗p + overhead	Cycle count includes the call and return
m=2 n=8, p=2 takes ∼274 cycles	versus 5∗m∗n∗p = 160
m=8, n=8, p=8 takes ∼3694 cycles	versus 5∗m∗n∗p = 2560
m=64, n=64, p=64 takes ∼718030 cycles	versus 5∗m∗n∗p = 1310720

Table 5.60: Performance Data

5.8.1.22 void mpy_SP_RMxRM_2 (float ∗ *y*, const float ∗ *w*, const float ∗ *x*, const uint16_t *m*, const uint16_t *n*, const uint16_t *p*)

Multiplication of Two Real Matrices (n even).

This module multiplies two real matrices.

$$y[] = w[] * x[]$$

Parameters:

y pointer to result matrix

w pointer to 1st source matrix

x pointer to 2nd source matrix

m number of rows in the first and output matrices

n number of columns in the first and rows in the second matrix

p number of columns in the second and output matrices

Attention:

1. n must be even and at least 4. If not, use mpy_SP_RMxRM().
2. There are no restrictions on the values of m and p with this function.

Cycles	Comment
2.5∗m∗n∗p + overhead	Cycle count includes the call and return
m=2 n=8, p=2 takes ∼199 cycles	versus 2.5∗m∗n∗p = 80
m=8, n=8, p=8 takes ∼2479 cycles	versus 2.5∗m∗n∗p = 1280
m=64, n=64, p=64 takes ∼725663 cycles	versus 2.5∗m∗n∗p = 655360

Table 5.61: Performance Data

5.8.1.23 void mpy_SP_RSxRV_2 (float ∗ *y*, const float ∗ *x*, const float *c*, const uint16_t *N*)

Multiplication of a Real scalar and a Real Vector.

This module multiplies a real scalar and a real vector.

$$y[i] = c * x[i]$$

Parameters:

y pointer to the product of the scalar and a real vector

x pointer to the real input vector

c scalar multiplier
N length of the x and y vectors

Attention:
N must be even and a minimum of 4.

Cycles	Comment
2∗N + 15	Cycle count includes the call and return

Table 5.62: Performance Data

5.8.1.24 void mpy_SP_RSxRVxRV_2 (float ∗ *y*, const float ∗ *w*, const float ∗ *x*, const float *c*, const uint16_t *N*)

Multiplication of a Real Scalar, a Real Vector, and another Real Vector.

This module multiplies a real scalar with a real vector and another real vector.

$$y[i] = c * w[i] * x[i]$$

Parameters:
y pointer to the product of the scalar and two real vectors
w pointer to the first real input vector
x pointer to the second real input vector
c scalar multiplier
N length of the w x and y vectors

Attention:
N must be even and a minimum of 4.

Cycles	Comment
3∗N + 22	Cycle count includes the call and return

Table 5.63: Performance Data

5.8.1.25 void mpy_SP_RVxCV (complex_float ∗ *y*, const complex_float ∗ *w*, const float ∗ *x*, const uint16_t *N*)

Multiplication of a Real Vector and a Complex Vector.

This module multiplies a real vector and a complex vector.

$$y_{re}[i] = x[i] * w_{re}[i]$$

$$y_{im}[i] = x[i] * w_{im}[i]$$

Parameters:
y pointer to the product of the real and complex vectors

w pointer to the complex input vector
x pointer to the real input vector
N length of the w x and y vectors

Attention:
N must be at least 2

Cycles	Comment
5∗N + 15	Cycle count includes the call and return

Table 5.64: Performance Data

5.8.1.26 void mpy_SP_RVxRV_2 (float ∗ *y*, const float ∗ *w*, const float ∗ *x*, const uint16_t *N*)

Multiplication of a Real Vector and a Real Vector.

This module multiplies two real vectors.

$$y[i] = w[i] * x[i]$$

Parameters:
y pointer to the product of two real vectors
w pointer to the first real input vector
x pointer to the second real input vector
N length of the w x and y vectors

Attention:
N must be even and a minimum of 4.

Cycles	Comment
3∗N + 17	Cycle count includes the call and return

Table 5.65: Performance Data

5.8.1.27 void qsort_SP_RV (void ∗ *x*, const uint16_t *N*)

Sort an Array of Floats.

This module sorts an array of floats. This function is a partially optimized version of qsort.c from the C28x cgtools lib qsort() v6.0.1.

Parameters:
x pointer to the input array
N size of the input array

Attention:
Performance is best with -o1, -mf3 compiler options (cgtools v6.0.1)

Cycles	Comment
N/A	This is a C function

Table 5.66: Performance Data

5.8.1.28 float rnd_SP_RS (const float *x*)

Rounding (Unbiased) of a Floating Point Scalar.

numerical examples: rnd_SP_RS(+4.4) = +4.0 \ rnd_SP_RS(-4.4) = -4.0 \ rnd_SP_RS(+4.5) = +5.0 \ rnd_SP_RS(-4.5) = -5.0 \ rnd_SP_RS(+4.6) = +5.0 \ rnd_SP_RS(-4.6) = -5.0 \

Parameters:

x input value

Returns:

rounded

Cycles	Comment
18	Cycle count includes the call and return

Table 5.67: Performance Data

5.8.1.29 void sub_SP_CSxCV (complex_float $*$ *y*, const complex_float $*$ *x*, const complex_float *c*, const uint16_t *N*)

Subtraction of a Complex Scalar from a Complex Vector.

This module subtracts a complex scalar from a complex vector.

$$y_{re}[i] = x_{re}[i] - c_{re}$$

$$y_{im}[i] = x_{im}[i] - c_{im}$$

Parameters:

y pointer to the difference of a complex scalar from a complex vector

x pointer to the complex input vector

c complex input scalar

N length of the x and y vectors

Attention:

N must be at least 2

Cycles	Comment
4∗N + 19	Cycle count includes the call and return (COFF)
4∗N + 16	Cycle count includes the call and return (EABI)

Table 5.68: Performance Data

5.8.1.30 void sub_SP_CVxCV (complex_float ∗ *y*, const complex_float ∗ *w*, const complex_float ∗ *x*, const uint16_t *N*)

Subtraction of a Complex Vector and another Complex Vector.

This module subtracts a complex vector from another complex vector.

$$y_{re}[i] = w_{re}[i] - x_{re}[i]$$

$$y_{im}[i] = w_{im}[i] - x_{im}[i]$$

Parameters:

- ***y*** pointer to the difference of two complex vectors
- ***w*** pointer to the first complex input vector
- ***x*** pointer to the second complex input vector
- ***N*** length of the w x and y vectors

Attention:

N must be at least 2

Cycles	Comment
6∗N + 15	Cycle count includes the call and return

Table 5.69: Performance Data

6 Application Programming Interface (FPU64)

6.1 Introduction to the Double Precision DSP Library API

The following functions are included in this release of the FPU Library. The source code for these functions can be found in the *source/C28x_FPU_LIB* folder.

DSP	
CFFT_f32	void CFFT_f32(CFFT_F32_STRUCT *);
CFFT_f32t	void CFFT_f32t(CFFT_F32_STRUCT *);
CFFT_f32u	void CFFT_f32u(CFFT_F32_STRUCT *);
CFFT_f32ut	void CFFT_f32ut(CFFT_F32_STRUCT *);
CFFT_f32_mag	void CFFT_f32_mag(CFFT_F32_STRUCT *);
CFFT_f32_mag_TMU0	void CFFT_f32_mag_TMU0(CFFT_F32_STRUCT *);
CFFT_f32s_mag	void CFFT_f32s_mag(CFFT_F32_STRUCT *);
CFFT_f32s_mag_TMU0	void CFFT_f32s_mag_TMU0(CFFT_F32_STRUCT *);
CFFT_f32_phase	void CFFT_f32_phase(CFFT_F32_STRUCT *);
CFFT_f32_phase_TMU0	void CFFT_f32_phase_TMU0(CFFT_F32_STRUCT *);
CFFT_f32_sincostable	void CFFT_f32_sincostable(CFFT_F32_STRUCT *);
CFFT32_f32_win	void CFFT32_f32_win(float *, float *, uint16_t);
CFFT32_f32_win_dual	void CFFT32_f32_win_dual(float *, float *, uint16_t);
ICFFT_f32	void ICFFT_f32(CFFT_F32_STRUCT *);
ICFFT_f32t	void ICFFT_f32t(CFFT_F32_STRUCT *);
RFFT_f32	void RFFT_f32(RFFT_F32_STRUCT *);
RFFT_f32u	void RFFT_f32u(RFFT_F32_STRUCT *);
RFFT_adc_f32	void RFFT_adc_f32(RFFT_ADC_F32_STRUCT *);
RFFT_adc_f32u	void RFFT_adc_f32u(RFFT_ADC_F32_STRUCT *);
RFFT_f32_mag	void RFFT_f32_mag(RFFT_F32_STRUCT *);
RFFT_f32_mag_TMU0	void RFFT_f32_mag_TMU0(RFFT_F32_STRUCT *);
RFFT_f32s_mag	void RFFT_f32s_mag(RFFT_F32_STRUCT *);
RFFT_f32s_mag_TMU0	void RFFT_f32s_mag_TMU0(RFFT_F32_STRUCT *);
RFFT_f32_phase	void RFFT_f32_phase(RFFT_F32_STRUCT *);
RFFT_f32_phase_TMU0	void RFFT_f32_phase_TMU0(RFFT_F32_STRUCT *);
RFFT_f32_sincostable	void RFFT_f32_sincostable(RFFT_F32_STRUCT *);
RFFT_f32_win	void RFFT_f32_win(float *, float *, uint16_t);
Filter	
FIR_f32	void FIR_FP_calc(FIR_FP_handle);
Matrix and Vector	
abs_SP_CV	void abs_SP_CV(float32 *, const complex_float *, const Uint16);
abs_SP_CV_2	void abs_SP_CV_2(float32 *, const complex_float *, const Uint16);
abs_SP_CV_TMU0	void abs_SP_CV_TMU0(float32 *, const complex_float *, const Uint16);
add_SP_CSxCV	void add_SP_CSxCV(complex_float *, const complex_float *, const complex_float, const Uint16);

Continued on next page

Table 6.1 – continued from previous page

add_SP_CVxCV	void add_SP_CVxCV(complex_float *, const complex_float *, const complex_float *, const Uint16);
iabs_SP_CV	void iabs_SP_CV(float32 *, const complex_float *, const Uint16);
iabs_SP_CV_2	void iabs_SP_CV_2(float32 *, const complex_float *, const Uint16);
iabs_SP_CV_TMU0	void iabs_SP_CV_TMU0(float32 *, const complex_float *, const Uint16);
mac_SP_RVxCV	complex_float mac_SP_RVxCV(const complex_float *, const float *, const uint16_t);
mac_SP_i16RVxCV	complex_float mac_SP_i16RVxCV(const complex_float *, const int16_t *, const uint16_t);
maxidx_SP_RV_2	Uint16 maxidx_SP_RV_2(float32 *, Uint16);
mean_SP_CV_2	complex_float mean_SP_CV_2(const complex_float *, const Uint16);
median_noreorder_SP_RV	float32 median_noreorder_SP_RV(const float32 *, Uint16);
median_SP_RV	float32 median_SP_RV(float32 *, Uint16);
mpy_SP_CSxCS	complex_float mpy_SP_CSxCS(complex_float, complex_float);
mpy_SP_CVxCV	void mpy_SP_CVxCV(complex_float *, const complex_float *, const complex_float *, const Uint16);
mpy_SP_CVxCVC	void mpy_SP_CVxCVC(complex_float *, const complex_float *, const complex_float *, const Uint16);
mpy_SP_RSxRV_2	void mpy_SP_RSxRV_2(float32 *, const float32 *, const float32, const Uint16);
mpy_SP_RSxRVxRV_2	void mpy_SP_RSxRVxRV_2(float32 *, const float32 *, const float32 *, const float32, const Uint16);
mpy_SP_RVxCV	void mpy_SP_RVxCV(complex_float *, const complex_float *, const float32 *, const Uint16);
mpy_SP_RVxRV_2	void mpy_SP_RVxRV_2(float32 *, const float32 *, const float32 *, const Uint16);
qsort_SP_RV	void qsort_SP_RV(void *, Uint16);
rnd_SP_RS	float32 rnd_SP_RS(float32);
sub_SP_CSxCV	void sub_SP_CSxCV(complex_float *, const complex_float *, const complex_float, const Uint16);
sub_SP_CVxCV	void sub_SP_CVxCV(complex_float *, const complex_float *, const complex_float *, const Uint16);
Utility	
memcpy_fast	void memcpy_fast(void *, const void *, Uint16);
memcpy_fast_far	void memcpy_fast_far(volatile void* , volatile const void* , uint16_t);
memset_fast	void memset_fast(void*, int16, Uint16);

Table 6.1: List of Functions

The examples for each was built using **CGT v22.6.0.LTS** with the following options:

```
-v28 -mt -ml -g --diag_warning=225 --float_support=fpu32 --tmu_support=tmu0
```

```
--define=CPU1
```

Each example has at least two build configurations, **RAM** and **FLASH**. Certain examples like the FFT have additional build configurations that demonstrate the TMU variants of certain functions, or the use of the fast RTS support library to speed up phase calculations.

Certain functions can be redefined to their TMU alternatives in order to maintain legacy code functionality. For example,

```
#ifdef __TMS320C28XX_TMU__
#define CFFT_f32_mag    CFFT_f32_mag_TMU0
#warn "Legacy function has been redefined to its TMU variant"
#endif //__TMS320C28XX_TMU__
```

The macro **__TMS320C28XX_TMU__** is defined by the compiler when the tmu_support option is set to tmu0.

In order to highlight the interleaving ability of the compiler for the fast square root function, its example was built with the options

```
-v28 -mt -ml -g -O2 --diag_warning=225 --optimize_with_debug
--float_support=fpu32
```

Each example has a script **SetupDebugEnv.js** that can be used with the scripting console in CCS to setup the watch windows and graphs automatically in the debug session. Please see CCS4:Scripting Console for more information

6.2 DSP Library Definitions and Types

Data Structures

- float64u_t
- float32u_t

Defines

- LIBRARY_VERSION

6.2.1 Data Structure Documentation

6.2.1.1 float64u_t

Definition:

```
typedef struct
{
    uint64_t ui64;
    int64_t i64;
    float64_t f64;
}
float64u_t
```

Members:

ui64 Unsigned long long representation.

i64 Signed long long represntaion.

f64 Double precision (64-bit) representation.

Description:

64-bit Double Precision Float The union of a double precision value, an unsigned long long and a signed long long allows for manipulation of the hex representation of the floating point value as well as signed and unsigned arithmetic to determine error metrics. This data type is only defined if the compiler option –float_support is set to fpu64

6.2.1.2 float32u_t

Definition:

```
typedef struct
{
    uint32_t ui32;
    int32_t i32;
    float f32;
}
float32u_t
```

Members:

ui32 Unsigned long representation.

i32 Signed long represntaion.

f32 Single precision (32-bit) representation.

Description:
32-bit Single Precision Float The union of a single precision value, an unsigned long and a signed long allows for manipulation of the hex representation of the floating point value as well as signed and unsigned arithmetic to determine error metrics. This data type is only defined if the compiler option –float_support is set to fpu32

6.2.2 Define Documentation

6.2.2.1 LIBRARY_VERSION

Definition:
```
#define LIBRARY_VERSION
```

Description:
DSP Library Version.

6.2.3 Typedef Documentation

6.2.3.1 v_pfn_v

Function pointer with a void pointer argument returning nothing.

6.3 Fast Fourier Transform (Double Precision)

Data Structures

- CFFT_f64_Struct
- CFFT_ADC_f64_Struct

Defines

- RFFT_f64_Struct

Functions

- static void CFFT_f64_setInputPtr (CFFT_f64_Handle fh, const float64_t *pi)
- static float64_t * CFFT_f64_getInputPtr (CFFT_f64_Handle fh)
- static void CFFT_f64_setOutputPtr (CFFT_f64_Handle fh, const float64_t *po)
- static float64_t * CFFT_f64_getOutputPtr (CFFT_f64_Handle fh)
- static void CFFT_f64_setTwiddlesPtr (CFFT_f64_Handle fh, const float64_t *pc)
- static float64_t * CFFT_f64_getTwiddlesPtr (CFFT_f64_Handle fh)
- static void CFFT_f64_setCurrInputPtr (CFFT_f64_Handle fh, const float64_t *pi)
- static float64_t * CFFT_f64_getCurrInputPtr (CFFT_f64_Handle fh)
- static void CFFT_f64_setCurrOutputPtr (CFFT_f64_Handle fh, const float64_t *po)
- static float64_t * CFFT_f64_getCurrOutputPtr (CFFT_f64_Handle fh)
- static void CFFT_f64_setStages (CFFT_f64_Handle fh, const uint16_t st)
- static uint16_t CFFT_f64_getStages (CFFT_f64_Handle fh)
- static void CFFT_f64_setFFTSize (CFFT_f64_Handle fh, const uint16_t sz)
- static uint16_t CFFT_f64_getFFTSize (CFFT_f64_Handle fh)
- static void CFFT_f64_setInitFunction (CFFT_f64_Handle fh, const v_pfn_v pfn)
- static v_pfn_v CFFT_f64_getInitFunction (CFFT_f64_Handle fh)
- static void CFFT_f64_setCalcFunction (CFFT_f64_Handle fh, const v_pfn_v pfn)
- static v_pfn_v CFFT_f64_getCalcFunction (CFFT_f64_Handle fh)
- static void CFFT_f64_setMagFunction (CFFT_f64_Handle fh, const v_pfn_v pfn)
- static v_pfn_v CFFT_f64_getMagFunction (CFFT_f64_Handle fh)
- static void CFFT_f64_setPhaseFunction (CFFT_f64_Handle fh, const v_pfn_v pfn)
- static v_pfn_v CFFT_f64_getPhaseFunction (CFFT_f64_Handle fh)
- static void CFFT_f64_setWinFunction (CFFT_f64_Handle fh, const v_pfn_v pfn)
- static v_pfn_v CFFT_f64_getWinFunction (CFFT_f64_Handle fh)
- static void CFFT_ADC_f64_setInBufPtr (CFFT_ADC_f64_Handle fh, const uint16_t *pi)
- static uint16_t * CFFT_ADC_f64_getInBufPtr (CFFT_ADC_f64_Handle fh)
- static void CFFT_ADC_f64_setTailPtr (CFFT_ADC_f64_Handle fh, const void *pt)
- static void * CFFT_ADC_f64_getTailPtr (CFFT_ADC_f64_Handle fh)
- void CFFT_f64 (CFFT_f64_Handle hndCFFT_f64)
- void CFFT_f64u (CFFT_f64_Handle hndCFFT_f64)
- void CFFT_f64_mag (CFFT_f64_Handle hndCFFT_f64)

- void CFFT_f64s_mag (CFFT_f64_Handle hndCFFT_f64)
- void CFFT_f64_phase (CFFT_f64_Handle hndCFFT_f64)
- void CFFT_f64_unpack (CFFT_f64_Handle hndCFFT_f64)
- void CFFT_f64_pack (CFFT_f64_Handle hndCFFT_f64)
- void ICFFT_f64 (CFFT_f64_Handle hndCFFT_f64)
- void ICFFT_f64u (CFFT_f64_Handle hndCFFT_f64)
- void RFFT_f64 (CFFT_f64_Handle hndCFFT_f64)
- void RFFT_f64u (CFFT_f64_Handle hndCFFT_f64)
- void RFFT_adc_f64 (CFFT_ADC_f64_Handle hndCFFT_ADC_f64)
- void RFFT_adc_f64u (CFFT_ADC_f64_Handle hndCFFT_ADC_f64)
- void RFFT_adc_f64_win (uint16_t *pBuffer, const uint16_t *pWindow, const uint16_t size)
- void RFFT_f64_mag (CFFT_f64_Handle hndCFFT_f64)
- void RFFT_f64s_mag (CFFT_f64_Handle hndCFFT_f64)
- void RFFT_f64_phase (CFFT_f64_Handle hndCFFT_f64)

Variables

- float64_t FPU64CFFTtwiddleFactors[768]

6.3.1 Data Structure Documentation

6.3.1.1 CFFT_f64_Struct

Definition:

```
typedef struct
{
    float64_t *p_input;
    float64_t *p_output;
    float64_t *p_twiddles;
    float64_t *p_currInput;
    float64_t *p_currOutput;
    uint16_t stages;
    uint16_t FFTSize;
    void (*init)(void *);
    void (*calc)(void *);
    void (*mag)(void *);
    void (*phase)(void *);
    void (*win)(void *);
}
CFFT_f64_Struct
```

Members:

p_input Pointer to the input buffer.
p_output Pointer to the output buffer.
p_twiddles Pointer to the twiddle factors.
p_currInput Points to input buffer at each FFT stage.
p_currOutput Points to output buffer at each FFT stage.

stages Number of FFT stages.
FFTSize FFT size (number of complex data points).
init Pointer to the initialization function.
calc Pointer to the calculation function.
mag Pointer to the magnitude function.
phase Pointer to the phase function.
win Pointer to the windowing function.

Description:
Complex FFT structure

6.3.1.2 CFFT_ADC_f64_Struct

Definition:
```
typedef struct
{
    uint16_t *p_input;
    void *p_tail;
}
CFFT_ADC_f64_Struct
```

Members:
p_input Pointer to the input buffer.
p_tail HASH(0x563bc8a18710)

Description:
Structure for the Complex FFT with ADC input

6.3.2 Define Documentation

6.3.2.1 RFFT_f64_Struct

Definition:
```
#define RFFT_f64_Struct
```

Description:
Re-define CFFT64 structures and functions such that RFFT64 terminology can be used for better clarification

6.3.3 Typedef Documentation

6.3.3.1 CFFT_f64_Handle

Definition:
```
typedef CFFT_f64_Struct *CFFT_f64_Handle
```

Description:
Handle to the CFFT structure object

6.3.3.2 CFFT_ADC_f64_Handle

Definition:

```
typedef CFFT_ADC_f64_Struct *CFFT_ADC_f64_Handle
```

Description:
Handle to the Complex FFT (with ADC input) structure object

6.3.4 Function Documentation

6.3.4.1 CFFT_f64_setInputPtr

Set the input buffer pointer.

Prototype:

```
static void
CFFT_f64_setInputPtr(CFFT_f64_Handle fh,
                     const float64_t *pi) [inline, static]
```

Parameters:
← ***fh,handle*** to the 'CFFT_f64_Struct' object
← ***pi,pointer*** to the input buffer

6.3.4.2 static float64_t* CFFT_f64_getInputPtr (CFFT_f64_Handle *fh*) `[inline, static]`

Get the input buffer pointer.

Parameters:
← ***fh,handle*** to the 'CFFT_f64_Struct' object

Returns:
pi , pointer to the input buffer

6.3.4.3 static void CFFT_f64_setOutputPtr (CFFT_f64_Handle *fh*, const float64_t * *po*) `[inline, static]`

Set the output buffer pointer.

Parameters:
← ***fh,handle*** to the 'CFFT_f64_Struct' object
← ***po,pointer*** to the output buffer

6.3.4.4 static float64_t* CFFT_f64_getOutputPtr (CFFT_f64_Handle *fh*) `[inline, static]`

Get the output buffer pointer.

Parameters:
← ***fh,handle*** to the 'CFFT_f64_Struct' object

Returns:
po , pointer to the output buffer

6.3.4.5 static void CFFT_f64_setTwiddlesPtr (CFFT_f64_Handle *fh*, const float64_t ∗ *pc*) `[inline, static]`

Set the twiddles pointer.

Parameters:
← ***fh,handle*** to the 'CFFT_f64_Struct' object
← ***pt,pointer*** to the twiddles

6.3.4.6 static float64_t∗ CFFT_f64_getTwiddlesPtr (CFFT_f64_Handle *fh*) `[inline, static]`

Get the twiddles pointer.

Parameters:
← ***fh,handle*** to the 'CFFT_f64_Struct' object

Returns:
pt , pointer to the twiddles

6.3.4.7 static void CFFT_f64_setCurrInputPtr (CFFT_f64_Handle *fh*, const float64_t ∗ *pi*) `[inline, static]`

Set the current input buffer pointer.

Parameters:
← ***fh,handle*** to the 'CFFT_f64_Struct' object
← ***pi,pointer*** to the current input buffer

6.3.4.8 static float64_t∗ CFFT_f64_getCurrInputPtr (CFFT_f64_Handle *fh*) `[inline, static]`

Get the current input buffer pointer.

Parameters:
← ***fh,handle*** to the 'CFFT_f64_Struct' object

Returns:
pi , pointer to the current input buffer

6.3.4.9 static void CFFT_f64_setCurrOutputPtr (CFFT_f64_Handle *fh*, const float64_t ∗ *po*) [inline, static]

Set the current output buffer pointer.

Parameters:

← ***fh,handle*** to the 'CFFT_f64_Struct' object
← ***po,pointer*** to the current output buffer

6.3.4.10 static float64_t∗ CFFT_f64_getCurrOutputPtr (CFFT_f64_Handle *fh*) [inline, static]

Get the current output buffer pointer.

Parameters:

← ***fh,handle*** to the 'CFFT_f64_Struct' object

Returns:

po , pointer to the current output buffer

6.3.4.11 static void CFFT_f64_setStages (CFFT_f64_Handle *fh*, const uint16_t *st*) [inline, static]

Set the number of FFT stages.

Parameters:

← ***fh,handle*** to the 'CFFT_f64_Struct' object
← ***st,number*** of FFT stages

6.3.4.12 static uint16_t CFFT_f64_getStages (CFFT_f64_Handle *fh*) [inline, static]

Get the size of the FFT.

Parameters:

← ***fh,handle*** to the 'CFFT_f64_Struct' object

Returns:

st , number of FFT stages

6.3.4.13 static void CFFT_f64_setFFTSize (CFFT_f64_Handle *fh*, const uint16_t *sz*) [inline, static]

Set the number of FFT stages.

Parameters:

← ***fh,handle*** to the 'CFFT_f64_Struct' object
← ***sz,size*** of the FFT

6.3.4.14 static uint16_t CFFT_f64_getFFTSize (CFFT_f64_Handle *fh*) [inline, static]

Get the size of the FFT.

Parameters:
← ***fh,handle*** to the 'CFFT_f64_Struct' object

Returns:
sz , size of the FFT

6.3.4.15 static void CFFT_f64_setInitFunction (CFFT_f64_Handle *fh*, const v_pfn_v *pfn*) [inline, static]

Set the initialization function.

Parameters:
← ***fh,handle*** to the 'CFFT_f64_Struct' object
← ***pfn,pointer*** to the initialization function

6.3.4.16 static v_pfn_v CFFT_f64_getInitFunction (CFFT_f64_Handle *fh*) [inline, static]

Get the initialization function.

Parameters:
← ***fh,handle*** to the 'CFFT_f64_Struct' object

Returns:
pfn , pointer to the initialization function

6.3.4.17 static void CFFT_f64_setCalcFunction (CFFT_f64_Handle *fh*, const v_pfn_v *pfn*) [inline, static]

Set the calculation function.

Parameters:
← ***fh,handle*** to the 'CFFT_f64_Struct' object
← ***pfn,pointer*** to the calculation function

6.3.4.18 static v_pfn_v CFFT_f64_getCalcFunction (CFFT_f64_Handle *fh*) [inline, static]

Get the calculation function.

Parameters:
← ***fh,handle*** to the 'CFFT_f64_Struct' object

Returns:
pfn , pointer to the calculation function

6.3.4.19 static void CFFT_f64_setMagFunction (CFFT_f64_Handle *fh*, const v_pfn_v *pfn*) `[inline, static]`

Set the magnitude function.

Parameters:
← ***fh,handle*** to the 'CFFT_f64_Struct' object
← ***pfn,pointer*** to the magnitude function

6.3.4.20 static v_pfn_v CFFT_f64_getMagFunction (CFFT_f64_Handle *fh*) `[inline, static]`

Get the magnitude function.

Parameters:
← ***fh,handle*** to the 'CFFT_f64_Struct' object

Returns:
pfn , pointer to the magnitude function

6.3.4.21 static void CFFT_f64_setPhaseFunction (CFFT_f64_Handle *fh*, const v_pfn_v *pfn*) `[inline, static]`

Set the phase function.

Parameters:
← ***fh,handle*** to the 'CFFT_f64_Struct' object
← ***pfn,pointer*** to the phase function

6.3.4.22 static v_pfn_v CFFT_f64_getPhaseFunction (CFFT_f64_Handle *fh*) `[inline, static]`

Get the phase function.

Parameters:
← ***fh,handle*** to the 'CFFT_f64_Struct' object

Returns:
pfn , pointer to the phase function

6.3.4.23 static void CFFT_f64_setWinFunction (CFFT_f64_Handle *fh*, const v_pfn_v *pfn*) `[inline, static]`

Set the windowing function.

Parameters:
← ***fh,handle*** to the 'CFFT_f64_Struct' object
← ***pfn,pointer*** to the windowing function

6.3.4.24 static v_pfn_v CFFT_f64_getWinFunction (CFFT_f64_Handle *fh*) [inline, static]

Get the windowing function.

Parameters:
← ***fh,handle*** to the 'CFFT_f64_Struct' object

Returns:
pfn , pointer to the windowing function

6.3.4.25 static void CFFT_ADC_f64_setInBufPtr (CFFT_ADC_f64_Handle *fh*, const uint16_t * *pi*) [inline, static]

Set the input buffer pointer.

Parameters:
← ***fh*** handle to the 'CFFT_ADC_f64_Struct' object
← ***pi*** pointer to the input buffer

6.3.4.26 static uint16_t* CFFT_ADC_f64_getInBufPtr (CFFT_ADC_f64_Handle *fh*) [inline, static]

get the input buffer pointer

Parameters:
← ***fh*** handle to the 'CFFT_ADC_f64_Struct' object

Returns:
pi pointer to the input buffer

6.3.4.27 static void CFFT_ADC_f64_setTailPtr (CFFT_ADC_f64_Handle *fh*, const void * *pt*) [inline, static]

Set the tail pointer.

Parameters:
← ***fh*** handle to the 'CFFT_ADC_f64_Struct' object
← ***pt*** pointer to the tail

6.3.4.28 static void* CFFT_ADC_f64_getTailPtr (CFFT_ADC_f64_Handle *fh*) [inline, static]

Get the tail pointer.

Parameters:
← ***fh*** handle to the 'CFFT_ADC_f64_Struct' object

Returns:
pt pointer to the tail

6.3.4.29 void CFFT_f64 (CFFT_f64_Handle *hndCFFT_f64*)

Complex Fast Fourier Transform.

This routine computes the 64-bit double precision FFT for an N-pt ($N = 2^n, n = 5 : 10$) complex input. This function reorders the input in bit-reversed format during the stage 1, 2 calculations. The routine uses two buffers in ping-pong fashion i.e. after each FFT stage the output and input buffers become the input and output buffers respectively for the next stage. The CFFT_f64_Struct object uses two pointers, CurrentInPtr and CurrentOutPtr to keep track of the switching. The user can determine the address of the final output by looking at the CurrentInPtr.

Parameters:
hndCFFT_F64 Pointer to the CFFT_f64_Struct object

Attention:
1. The routine requires the use of two buffers, each of size 2N and type float64_t (64-bit), for computation; the input buffer must be aligned to a memory address of 8N words (16-bit). Refer to the CFFT linker command file to see an example of this.
2. If alignment is not possible the user can use the alternative, albeit slower, function CFFT_f64u

The maximum size of this complex FFT is 4096

Warning:
This function is not re-entrant as it uses global variables to store certain parameters

Samples	Cycles
64	3805
128	8649
256	19571
512	43935
1024	98683

Table 6.2: Performance Data

6.3.4.30 void CFFT_f64u (CFFT_f64_Handle *hndCFFT_f64*)

Complex Fast Fourier Transform (Unaligned).

This routine computes the 64-bit double precision FFT for an N-pt ($N = 2^n, n = 5 : 10$) complex input. This function reorders the input in bit-reversed format during the stage 1, 2 calculations. The routine uses two buffers in ping-pong fashion i.e. after each FFT stage the output and input buffers become the input and output buffers respectively for the next stage. The CFFT_f64_Struct object uses two pointers, CurrentInPtr and CurrentOutPtr to keep track of the switching. The user can determine the address of the final output by looking at the CurrentInPtr.

Parameters:
hndCFFT_F64 Pointer to the CFFT_f64_Struct object

Attention:

The routine requires the use of two buffers, each of size 2N and type float64_t (64-bit), for computation; the input buffer need not be aligned.

1. If alignment is possible the user can use the alternative faster function CFFT_f64

The maximum size of this complex FFT is 4096

Warning:

This function is not re-entrant as it uses global variables to store certain parameters

Samples	Cycles
64	4107
128	9254
256	20783
512	46362
1024	102648

Table 6.3: Performance Data

6.3.4.31 void CFFT_f64_mag (CFFT_f64_Handle *hndCFFT_f64*)

Complex FFT Magnitude.

This module computes the complex FFT magnitude. The output from CFFT_f64_mag matches the magnitude output from the FFT found in common mathematics software and Code Composer Studio FFT graphs. If instead a normalized magnitude like that performed by the fixed-point TMS320C28x IQmath FFT library is required, then the CFFT_f64s_mag function can be used. In fixed-point algorithms scaling is performed to avoid overflowing data. Floating-point calculations do not need this scaling to avoid overflow and therefore the CFFT_f64_mag function can be used instead.

Parameters:

hndCFFT_f64 Pointer to the CFFT_f64_Struct object

Attention:

1. The Magnitude buffer does not require memory alignment to a boundary

Samples	Cycles
64	2696
128	5288
256	10470
512	20838
1024	41574

Table 6.4: Performance Data

6.3.4.32 void CFFT_f64s_mag (CFFT_f64_Handle *hndCFFT_f64*)

Complex FFT Magnitude (Scaled).

This module computes the scaled complex FFT magnitude. The scaling is $\frac{1}{[2^{FFT\_STAGES-1}]}$, and is done to match the normalization performed by the fixed-point TMS320C28x IQmath FFT library for overflow avoidance. Floating-point calculations do not need this scaling to avoid overflow and therefore the CFFT_f64_mag function can be used instead. The output from CFFT_f64_s_mag matches the magnitude output from the FFT found in common mathematics software and Code Composer Studio FFT graphs.

Parameters:

hndCFFT_f64 Pointer to the CFFT_f64_Struct object

Attention:

1. The Magnitude buffer does not require memory alignment to a boundary

Samples	Cycles
64	2771
128	5427
256	10738
512	21362
1024	42610

Table 6.5: Performance Data

6.3.4.33 void CFFT_f64_phase (CFFT_f64_Handle *hndCFFT_f64*)

Complex FFT Phase.

Parameters:

hndCFFT_f64 Pointer to the CFFT_f64_Struct object

Attention:

1. The Phase buffer does not require memory alignment to a boundary
2. The phase function calls the atan2 function in the runtime-support library. The phase function has not been optimized at this time.
3. The use of the atan2 function in the FPU fast RTS library will speed up this routine.

Samples	Cycles (fastRTS)	Cycles (RTS)
64	6359	N/A
128	12695	N/A
256	25367	N/A
512	50709	N/A
1024	101397	N/A

Table 6.6: Performance Data

6.3.4.34 void CFFT_f64_unpack (CFFT_f64_Handle *hndCFFT_f64*)

Unpack the N-point complex FFT output to get the FFT of a 2N point real sequence.

In order to get the FFT of a real N-point sequence, we treat the input as an N/2-point complex sequence, take its complex FFT, use the following properties to get the N-pt Fourier transform of the real sequence

$$FFT_n(k,f) = FFT_{N/2}(k,f_e) + e^{\frac{-j2\pi k}{N}} FFT_{N/2}(k,f_o)$$

where f_e is the even elements, f_o the odd elements, k = 0 to $\frac{N}{2} - 1$ and

$$F_e(k) = \frac{Z(k) + Z(\frac{N}{2} - k)^*}{2}$$

$$F_o(k) = -j\frac{Z(k) - Z(\frac{N}{2} - k)^*}{2}$$

We get the first N/2 points of the FFT by combining the above two equations

$$F(k) = F_e(k) + e^{\frac{-j2\pi k}{N}} F_o(k)$$

Parameters:

hndCFFT_f64 Pointer to the CFFT_f64_Struct object

Note:

1. The unpack routine yields the spectrum of the real input data; the spectrum has a real part that is symmetric, and an imaginary part that is antisymmetric about the nyquist frequency. We only need calculate half the spectrum, up to the nyquist bin, while the latter half can be derived from the first half using the conjugate symmetry properties of the spectrum
2. The output is written to the buffer pointer to by p_currOutput

See also:

http://www.engineeringproductivitytools.com/stuff/T0001/PT10.HTM for the entire derivation

Real FFT Using a Complex FFT

Samples	Cycles
64	756
128	1460
256	2868
512	5683
1024	11315

Table 6.7: Performance Data

6.3.4.35 void CFFT_f64_pack (CFFT_f64_Handle *hndCFFT_f64*)

Pack the N/2-point complex FFT output to get the spectrum of an N-point real sequence.

In order to reverse the process of the forward real FFT,

$$F_e(k) = \frac{F(k) + F(\frac{N}{2} - k)^*}{2}$$

$$F_o(k) = \frac{F(k) - F(\frac{N}{2} - k)^*}{2} e^{\frac{j2\pi k}{N}}$$

where f_e is the even elements, f_o the odd elements, and k = 0 to $\frac{N}{2} - 1$. The array for the IFFT then becomes:

$$Z(k) = F_e(k) + jF_o(k),\ k = 0...\frac{N}{2} - 1$$

Parameters:

hndCFFT_f64 Pointer to the CFFT_f64_Struct object

Note:

The output is written to the buffer pointer to by p_currOutput

See also:

http://www.engineeringproductivitytools.com/stuff/T0001/PT10.HTM for the entire derivation

Real FFT Using a Complex FFT

Samples	Cycles
64	761
128	1465
256	2873
512	5688
1024	11320

Table 6.8: Performance Data

6.3.4.36 void ICFFT_f64 (CFFT_f64_Handle *hndCFFT_f64*)

Inverse Complex FFT (Swap Method).

This routine computes the 64-bit floating-point Inverse FFT for an N-pt ($N = 2^n, n = 5 : 10$) complex input. It uses the forward FFT to do this by first swapping the real and imaginary parts of the input, running the forward FFT and then swapping the real and imaginary parts of the output to get the final answer.

The inverse FFT is given by:

$$x(n) = \sum_{k=0}^{N-1} (X(k)e^{j.2.pi.k.n/N})$$

Note that

$$\begin{aligned} -jX(k) &= -j(X_r(k) + jX_i(k)) \\ &= (X_i(k) - jX_r(k)) \\ &= (swap_{r\leftrightarrow i}(X(k)))^* \\ &= X^*_{sri}(k) \end{aligned}$$

and

$$\begin{aligned} jX^*_{sri}(k) &= j(X_i(k) - jX_r(k)) \\ &= (X_r(k) + jX_i(k)) \end{aligned}$$

Multiplying and dividing the original equation by -j

$$x(n) = \sum_{k=0}^{N-1} j(-jX(k)e^{j.2.pi.k.n/N})$$

$$= j(\sum_{k=0}^{N-1}(X^{*}_{sri}(k)e^{j.2.pi.k.n/N})$$

$$= j(\sum_{k=0}^{N-1}(X_{sri}(k)e^{-j.2.pi.k.n/N})^{*}$$

$$= j.(CFFT(swap_{r\leftrightarrow i}(X(k))))^{*}$$

$$= swap_{r\leftrightarrow i}(CFFT(swap_{r\leftrightarrow i}(X(k))))$$

The routine uses two buffers in ping-pong fashion i.e. after each FFT stage the output and input buffers become the input and output buffers respectively for the next stage. The CFFT_f64_Struct object uses two pointers, CurrentInPtr and CurrentOutPtr to keep track of the switching. The user can determine the address of the final output by looking at the CurrentInPtr.

Parameters:

hndCFFT_f64 Pointer to the CFFT_f64_Struct object

Attention:

1. The routine requires the use of two buffers, each of size 2N and type float64_t (64-bit), for computation; the input buffer must be aligned to a memory address of 8N words (16-bit). Refer to the FFT linker command file to see an example of this.
2. The maximum size of this inverse complex FFT is 4096

Samples	Cycles
64	4537
128	10132
256	22559
512	49927
1024	109746

Table 6.9: Performance Data

6.3.4.37 void ICFFT_f64u (CFFT_f64_Handle *hndCFFT_f64*)

Unaligned Inverse Complex FFT (Swap Method).

This routine computes the 64-bit floating-point Inverse FFT for an N-pt ($N = 2^n, n = 5 : 10$) complex input. It uses the forward FFT to do this by first swapping the real and imaginary parts of the input, running the forward FFT and then swapping the real and imaginary parts of the output to get the final answer.

The inverse FFT is given by:

$$x(n) = \sum_{k=0}^{N-1}(X(k)e^{j.2.pi.k.n/N})$$

Note that

$$-jX(k) = -j(X_r(k) + jX_i(k))$$

$$= (X_i(k) - jX_r(k))$$

$$= (swap_{r\leftrightarrow i}(X(k)))^*$$
$$= X^*_{sri}(k)$$

and

$$jX^*_{sri}(k) = j(X_i(k) - jX_r(k))$$
$$= (X_r(k) + jX_i(k))$$

Multiplying and dividing the original equation by -j

$$x(n) = \sum_{k=0}^{N-1} j(-jX(k)e^{j.2.pi.k.n/N})$$

$$= j(\sum_{k=0}^{N-1}(X^*_{sri}(k)e^{j.2.pi.k.n/N})$$

$$= j(\sum_{k=0}^{N-1}(X_{sri}(k)e^{-j.2.pi.k.n/N})^*$$

$$= j.(CFFT(swap_{r\leftrightarrow i}(X(k))))^*$$

$$= swap_{r\leftrightarrow i}(CFFT(swap_{r\leftrightarrow i}(X(k))))$$

The routine uses two buffers in ping-pong fashion i.e. after each FFT stage the output and input buffers become the input and output buffers respectively for the next stage. The CFFT_f64_Struct object uses two pointers, CurrentInPtr and CurrentOutPtr to keep track of the switching. The user can determine the address of the final output by looking at the CurrentInPtr.

Parameters:

hndCFFT_f64 Pointer to the CFFT_f64_Struct object

Attention:

1. The routine requires the use of two buffers, each of size 2N and type float64_t (64-bit), for computation; the input buffer need not be aligned to any boundary; if it is possible to align the input buffer to an 8N word boundary the user may use the faster ICFFT_f64 function
2. The maximum size of this inverse complex FFT is 4096

Samples	Cycles
64	4876
128	10808
256	23905
512	52619
1024	115129

Table 6.10: Performance Data

6.3.4.38 void RFFT_f64 (CFFT_f64_Handle *hndCFFT_f64*)

Real Fast Fourier Transform (Alternate).

This routine computes the 64-bit double precision FFT for an N-pt ($N = 2^n, n = 5 : 10$) real input. This function reorders the input in bit-reversed format as part of the stage 1,2 and 3 computations. The routine uses two buffers in ping-pong fashion i.e. after each FFT stage the output and input buffers become the input and output buffers respectively for the next stage. This algorithm only allocates memory, and performs computation, for the non-zero elements of the input (the real part). The complex conjugate nature of the spectrum (for real only data) affords savings in space and computation, therefore the algorithm only calculates the spectrum from the 0th bin to the Nyquist bin (included).

Another approach to calculate the real FFT would be to treat the real N-point data as N/2 complex, run the forward complex N/2 point FFT followed by an "unpack" function

Parameters:

hndCFFT_F64 Pointer to the CFFT_f64_Struct object

Attention:

1. The routine requires the use of two buffers, each of size N and type float64_t (64-bit), for computation; the input buffer must be aligned to a memory address of 4N words (16-bit). Refer to the RFFT linker command file to see an example of this.
2. If alignment is not possible the user can use the alternative, albeit slower, function RFFT_f64_u_calc

Warning:

This function is not re-entrant as it uses global variables to store certain parameters

Attention:

The maximum size of the real FFT is 16384

See also:

Real FFT Using a Complex FFT

Samples	Cycles
64	1729
128	3914
256	8915
512	20219
1024	45476

Table 6.11: Performance Data

6.3.4.39 void RFFT_f64u (CFFT_f64_Handle *hndCFFT_f64*)

Unaligned Real Fast Fourier Transform (Alternate).

This routine computes the 64-bit double precision FFT for an N-pt ($N = 2^n, n = 5 : 10$) real input. This function reorders the input in bit-reversed format as part of the stage 1,2 and 3 computations. The routine uses two buffers in ping-pong fashion i.e. after each FFT stage the output and input buffers become the input and output buffers respectively for the next stage. This algorithm only allocates memory, and performs computation, for the non-zero elements of the input (the real part).

The complex conjugate nature of the spectrum (for real only data) affords savings in space and computation, therefore the algorithm only calculates the spectrum from the 0th bin to the Nyquist bin (included).

Another approach to calculate the real FFT would be to treat the real N-point data as N/2 complex, run the forward complex N/2 point FFT followed by an "unpack" function

Parameters:

hndCFFT_F64 Pointer to the CFFT_f64_Struct object

Attention:

1. The routine requires the use of two buffers, each of size N and type float64_t (64-bit), for computation; the input buffer need not be aligned.
2. If alignment is possible the user can use the faster alternative function RFFT_f64_calc

Warning:

This function is not re-entrant as it uses global variables to store certain parameters

Attention:

The maximum size of the real FFT is 16384

See also:

Real FFT Using a Complex FFT

Samples	Cycles
64	1903
128	4256
256	9593
512	21569
1024	48170

Table 6.12: Performance Data

6.3.4.40 void RFFT_adc_f64 (CFFT_ADC_f64_Handle *hndCFFT_ADC_f64*)

Real FFT with ADC Input.

This routine computes the 64-bit double precision FFT for an N-pt ($N = 2^n, n = 5 : 10$) real 12-bit ADC input. This function reorders the input in bit-reversed format as part of the stage 1,2 and 3 computations. The routine uses two buffers in ping-pong fashion i.e. after each FFT stage the output and input buffers become the input and output buffers respectively for the next stage. This algorithm only allocates memory, and performs computation, for the non-zero elements of the input (the real part). The complex conjugate nature of the spectrum (for real only data) affords savings in space and computation, therefore the algorithm only calculates the spectrum from the 0th bin to the Nyquist bin (included).

Another approach to calculate the real FFT would be to treat the real N-point data as N/2 complex, run the forward complex N/2 point FFT followed by an "unpack" function

Parameters:

hndCFFT_ADC_f64 Pointer to the CFFT_ADC_f64_Struct object

Attention:

1. The routine requires the use of two buffers, the input of size 4N and type uint16_t, the output of size N and type float64_t, for computation; the input buffer must be aligned to a memory address of N words (16-bit). Refer to the RFFT linker command file to see an example of this. If it is not possible to align the input buffer to an N word boundary you may use the unaligned version RFFT_f64_u_adc_calc().

Warning:

This function is not re-entrant as it uses global variables to store certain parameters

Attention:

The maximum size of the real FFT is 16384

See also:

Real FFT Using a Complex FFT

Samples	Cycles
64	2046
128	4551
256	10195
512	22780
1024	50597

Table 6.13: Performance Data

6.3.4.41 void RFFT_adc_f64u (CFFT_ADC_f64_Handle *hndCFFT_ADC_f64*)

Unaligned Real FFT with ADC Input.

This routine computes the 64-bit double precision FFT for an N-pt ($N = 2^n, n = 5 : 10$) real 12-bit ADC input. This function reorders the input in bit-reversed format as part of the stage 1,2 and 3 computations. The routine uses two buffers in ping-pong fashion i.e. after each FFT stage the output and input buffers become the input and output buffers respectively for the next stage. This algorithm only allocates memory, and performs computation, for the non-zero elements of the input (the real part). The complex conjugate nature of the spectrum (for real only data) affords savings in space and computation, therefore the algorithm only calculates the spectrum from the 0th bin to the nyquist bin (included).

Another approach to calculate the real FFT would be to treat the real N-point data as N/2 complex, run the forward complex N/2 point FFT followed by an "unpack" function

Parameters:

hndCFFT_ADC_f64 Pointer to the CFFT_ADC_f64_Struct object

Attention:

1. The routine requires the use of two buffers, the input of size 4N and type uint16_t, the output of size N and type float64_t, for computation; the input buffer needn't be aligned. If it is possible to align the input buffer to an N word boundary consider using the faster RFFT_f64_adc_calc().

Warning:

This function is not re-entrant as it uses global variables to store certain parameters

Attention:
The maximum size of the real FFT is 16384

See also:
Real FFT Using a Complex FFT

Samples	Cycles
64	2180
128	4813
256	10709
512	23806
1024	52647

Table 6.14: Performance Data

6.3.4.42 void RFFT_adc_f64_win (uint16_t ∗ *pBuffer*, const uint16_t ∗ *pWindow*, const uint16_t *size*)

Windowing function for the 64-bit real FFT with ADC Input.

Parameters:

pBuffer pointer to the uint16_t buffer that needs to be windowed

pWindow pointer to the float windowing table

size size of the buffer This function applies the window to a 2N point real data buffer that has not been reordered in the bit-reversed format

Note:
This function is identical to _RFFT_adc_f32_win. It is replicated here so as to avoid conflicting ISA issues when including both the fpu32 and fpu64 libraries.

Attention:
1. The routine requires the window to be unsigned int (16-bit). The user must take the desired floating point window from the header files (e.g. HANN1024 from fpu32/fpu_fft_hann.h)and convert it to Q16 by multiplying by 2^16 and then flooring the value before converting it to uint16_t

Samples	Cycles
64	346
128	666
256	1306
512	2586
1024	5146

Table 6.15: Performance Data

6.3.4.43 void RFFT_f64_mag (CFFT_f64_Handle *hndCFFT_f64*)

Real FFT Magnitude.

This module computes the real FFT magnitude. The output from RFFT_f64_mag matches the magnitude output from the FFT found in common mathematics software and Code Composer Studio

FFT graphs. If instead a normalized magnitude like that performed by the fixed-point TMS320C28x IQmath FFT library is required, then the RFFT_f64s_mag function can be used. In fixed-point algorithms scaling is performed to avoid overflowing data. Floating-point calculations do not need this scaling to avoid overflow and therefore the RFFT_f64_mag function can be used instead.

Parameters:

hndCFFT_f64 Pointer to the CFFT_f64_Struct object

Attention:

1. The macro RFFT_MAG_IN_PLACE must be set to 1 if the input and output arrays are the same, i.e. the magnitude operation is done in-place. If the input and output arrays are different set this macro to 0. Always rebuild the library when changing this macro
2. The Magnitude buffer does not require memory alignment to a boundary

Samples	Cycles
64	1332
128	2628
256	5220
512	10402
1024	20770

Table 6.16: Performance Data

6.3.4.44 void RFFT_f64s_mag (CFFT_f64_Handle *hndCFFT_f64*)

Real FFT Magnitude (Scaled).

This module computes the scaled real FFT magnitude. The scaling is $\frac{1}{[2^{FFT\_STAGES-1}]}$, and is done to match the normalization performed by the fixed-point TMS320C28x IQmath FFT library for overflow avoidance. Floating-point calculations do not need this scaling to avoid overflow and therefore the RFFT_f64_mag function can be used instead. The output from RFFT_f64_s_mag matches the magnitude output from the FFT found in common mathematics software and Code Composer Studio FFT graphs.

Parameters:

hndCFFT_f64 Pointer to the CFFT_f64_Struct object

Attention:

1. The macro RFFT_MAG_IN_PLACE must be set to 1 if the input and output arrays are the same, i.e. the magnitude operation is done in-place. If the input and output arrays are different set this macro to 0. Always rebuild the library when changing this macro
2. The Magnitude buffer does not require memory alignment to a boundary

Samples	Cycles
64	1403
128	2725
256	5449
512	10818
1024	21575

Table 6.17: Performance Data

6.3.4.45 void RFFT_f64_phase (CFFT_f64_Handle *hndCFFT_f64*)

Real FFT Phase.

Parameters:

hndCFFT_f64 Pointer to the CFFT_f64_Struct object

Attention:

1. The Phase buffer does not require memory alignment to a boundary
2. The phase function calls the atan2 function in the runtime-support library. The phase function has not been optimized at this time.
3. The use of the atan2 function in the FPU fast RTS library will speed up this routine.

Samples	Cycles (fastRTS)	Cycles (RTS)
64	3280	N/A
128	6448	N/A
256	12784	N/A
512	25454	N/A
1024	50798	N/A

Table 6.18: Performance Data

6.3.5 Variable Documentation

6.3.5.1 float64_t FPU64CFFTtwiddleFactors[768]

Complex FFT Twiddle Factor Table

6.4 Real FFT Using a Complex FFT

It is possible to run the Fast Fourier Transform on a sequence of real data using the complex FFT. For a 2N point real sequence, the user would treat the data as N-pt complex (no rearrangement required) and run it through an N point complex FFT. In order to derive the correct spectrum, you would have to “unpack” the output. The derivations can be found here:

http://www.engineeringproductivitytools.com/stuff/T0001/PT10.HTM

Similarly, to run an inverse Real FFT, the user would “pack” the data and run it through an N-point Forward Complex FFT and then conjugate its complex output to get the original 2N point signal.

Note 1 When running an inverse real FFT after the forward real FFT, the user must take care to first switch the **Input** and **Output** pointers in the FFT object before calling the FFT routine again.

Note 2 Refer to the project, **rfft_f64**, in the examples folder for a demonstration of the use of a complex FFT followed by the unpack function to compute the forward real FFT. The unpack routine yields the spectrum of the real input data; the spectrum has a real part that is symmetric, and an imaginary part that is antisymmetric about the nyquist frequency. We only need calculate half the spectrum, up to the nyquist bin, while the latter half can be derived from the first half using the conjugate symmetry properties of the spectrum.

Note 3 The project **rfft_alt_f64**, on the other hand, demonstrates the real FFT function, **R**FFT_f64_calc, which runs a 2N point complex FFT on the real input data treating the imaginary portion as zeros, computing only half the spectrum - the spectrum of real data is complex conjugate about the nyquist point - and preserving only the real parts of certain points in the butterfly groups (in every stage) that are necessarily real.

Note 4 Refer to the project, **irfft_f64**, in the examples folder for a demonstration of the use of the pack function followed by a complex FFT to compute the inverse real FFT.

See also:
CFFT_f64_pack, CFFT_f64_unpack

6.5 Filters (Double Precision)

Data Structures

- FIR_f64
- IIR_f64

Functions

- static void FIR_f64_setCoefficientsPtr (FIR_f64_Handle fh, const float64_t *pc)
- static float64_t * FIR_f64_getCoefficientsPtr (FIR_f64_Handle fh)
- static void FIR_f64_setDelayLinePtr (FIR_f64_Handle fh, const float64_t *pdl)
- static float64_t * FIR_f64_getDelayLinePtr (FIR_f64_Handle fh)
- static void FIR_f64_setInputPtr (FIR_f64_Handle fh, const float64_t *pi)
- static float64_t * FIR_f64_getInputPtr (FIR_f64_Handle fh)
- static void FIR_f64_setOutputPtr (FIR_f64_Handle fh, const float64_t *po)
- static float64_t * FIR_f64_getOutputPtr (FIR_f64_Handle fh)
- static void FIR_f64_setOrder (FIR_f64_Handle fh, const uint16_t order)
- static uint16_t FIR_f64_getOrder (FIR_f64_Handle fh)
- static void FIR_f64_setInitFunction (FIR_f64_Handle fh, const v_pfn_v pfn)
- static v_pfn_v FIR_f64_getInitFunction (FIR_f64_Handle fh)
- static void FIR_f64_setCalcFunction (FIR_f64_Handle fh, const v_pfn_v pfn)
- static v_pfn_v FIR_f64_getCalcFunction (FIR_f64_Handle fh)
- static void IIR_f64_setCoefficientsAPtr (IIR_f64_Handle fh, const float64_t *pca)
- static float64_t * IIR_f64_getCoefficientsAPtr (IIR_f64_Handle fh)
- static void IIR_f64_setCoefficientsBPtr (IIR_f64_Handle fh, const float64_t *pcb)
- static float64_t * IIR_f64_getCoefficientsBPtr (IIR_f64_Handle fh)
- static void IIR_f64_setDelayLinePtr (IIR_f64_Handle fh, const float64_t *pdl)
- static float64_t * IIR_f64_getDelayLinePtr (IIR_f64_Handle fh)
- static void IIR_f64_setInputPtr (IIR_f64_Handle fh, const float64_t *pi)
- static float64_t * IIR_f64_getInputPtr (IIR_f64_Handle fh)
- static void IIR_f64_setOutputPtr (IIR_f64_Handle fh, const float64_t *po)
- static float64_t * IIR_f64_getOutputPtr (IIR_f64_Handle fh)
- static void IIR_f64_setScalePtr (IIR_f64_Handle fh, const float64_t *psv)
- static float64_t * IIR_f64_getScalePtr (IIR_f64_Handle fh)
- static void IIR_f64_setOrder (IIR_f64_Handle fh, const uint16_t order)
- static uint16_t IIR_f64_getOrder (IIR_f64_Handle fh)
- static void IIR_f64_setInitFunction (IIR_f64_Handle fh, const v_pfn_v pfn)
- static v_pfn_v IIR_f64_getInitFunction (IIR_f64_Handle fh)
- static void IIR_f64_setCalcFunction (IIR_f64_Handle fh, const v_pfn_v pfn)
- static v_pfn_v IIR_f64_getCalcFunction (IIR_f64_Handle fh)
- void FIR_f64_calc (FIR_f64_Handle hndFIR_f64)
- void FIR_f64_init (FIR_f64_Handle hndFIR_f64)
- void IIR_f64_calc (IIR_f64_Handle hndIIR_f64)
- void IIR_f64_init (IIR_f64_Handle hndIIR_f64)

6.5.1 Data Structure Documentation

6.5.1.1 FIR_f64

Definition:

```
typedef struct
{
    float64_t *p_coeff;
    float64_t *p_dbuffer;
    float64_t *p_input;
    float64_t *p_output;
    uint16_t order;
    void (*init)(void *);
    void (*calc)(void *);
}
FIR_f64
```

Members:

p_coeff Pointer to the filter coefficients.
p_dbuffer Delay buffer pointer.
p_input Pointer to the latest input sample.
p_output Pointer to the filter output.
order Order of the filter.
init Pointer to the initialization function.
calc Pointer to the calculation function.

Description:

Structure for the double precision Finite Impulse Response filter

6.5.1.2 IIR_f64

Definition:

```
typedef struct
{
    float64_t *p_coeff_A;
    float64_t *p_coeff_B;
    float64_t *p_dbuffer;
    float64_t *p_input;
    float64_t *p_output;
    float64_t *p_scale;
    uint16_t order;
    void (*init)(void *);
    void (*calc)(void *);
}
IIR_f64
```

Members:

p_coeff_A Pointer to the denominator coefficients.
p_coeff_B Pointer to the numerator coefficients.
p_dbuffer Delay buffer pointer.

p_input Pointer to the latest input sample.
p_output Pointer to the filter output.
p_scale Pointer to the biquad(s) scale values.
order Order of the filter.
init Pointer to the initialization function.
calc Pointer to the calculation function.

Description:
Structure for the double precision Infinite Impulse Response filter

6.5.2 Function Documentation

6.5.2.1 FIR_f64_setCoefficientsPtr

Set the coefficients pointer.

Prototype:
```
static void
FIR_f64_setCoefficientsPtr(FIR_f64_Handle fh,
                           const float64_t *pc) [inline, static]
```

Parameters:
← ***fh*** handle to the 'FIR_f64' object
← ***pc*** pointer to the coefficients

6.5.2.2 static float64_t* FIR_f64_getCoefficientsPtr (FIR_f64_Handle *fh*) `[inline, static]`

Get the coefficients pointer.

Parameters:
← ***fh*** handle to the 'FIR_f64' object

Returns:
pc pointer to the coefficients

6.5.2.3 static void FIR_f64_setDelayLinePtr (FIR_f64_Handle *fh*, const float64_t * *pdl*) `[inline, static]`

Set the delay line pointer.

Parameters:
← ***fh*** handle to the 'FIR_f64' object
← ***pdl*** pointer to the delay line

6.5.2.4 static float64_t* FIR_f64_getDelayLinePtr (FIR_f64_Handle *fh*) [inline, static]

Get the delay line pointer.

Parameters:
← ***fh*** handle to the 'FIR_f64' object

Returns:
pdl pointer to the delay line

6.5.2.5 static void FIR_f64_setInputPtr (FIR_f64_Handle *fh*, const float64_t * *pi*) [inline, static]

Set the input pointer.

Parameters:
← ***fh*** handle to the 'FIR_f64' object
← ***pi*** pointer to the current input

6.5.2.6 static float64_t* FIR_f64_getInputPtr (FIR_f64_Handle *fh*) [inline, static]

Get the input pointer.

Parameters:
← ***fh*** handle to the 'FIR_f64' object

Returns:
pi pointer to the current input

6.5.2.7 static void FIR_f64_setOutputPtr (FIR_f64_Handle *fh*, const float64_t * *po*) [inline, static]

Set the output pointer.

Parameters:
← ***fh*** handle to the 'FIR_f64' object
← ***po*** pointer to the current output

6.5.2.8 static float64_t* FIR_f64_getOutputPtr (FIR_f64_Handle *fh*) [inline, static]

Get the output pointer.

Parameters:
← ***fh*** handle to the 'FIR_f64' object

Returns:
po pointer to the current output

6.5.2.9 static void FIR_f64_setOrder (FIR_f64_Handle *fh*, const uint16_t *order*) `[inline, static]`

Set the order of the filter.

Parameters:

← ***fh*** handle to the 'FIR_f64' object

← ***order*** Order of the filter

6.5.2.10 static uint16_t FIR_f64_getOrder (FIR_f64_Handle *fh*) `[inline, static]`

Get the order of the filter.

Parameters:

← ***fh*** handle to the 'FIR_f64' object

Returns:

order Order of the filter

6.5.2.11 static void FIR_f64_setInitFunction (FIR_f64_Handle *fh*, const v_pfn_v *pfn*) `[inline, static]`

Set the init function.

Parameters:

← ***fh*** handle to the 'FIR_f64' object

← ***pfn*** pointer to the init function

6.5.2.12 static v_pfn_v FIR_f64_getInitFunction (FIR_f64_Handle *fh*) `[inline, static]`

Get the init function.

Parameters:

← ***fh*** handle to the 'FIR_f64' object

Returns:

pfn pointer to the init function

6.5.2.13 static void FIR_f64_setCalcFunction (FIR_f64_Handle *fh*, const v_pfn_v *pfn*) `[inline, static]`

Set the calc function.

Parameters:

← ***fh*** handle to the 'FIR_f64' object

← ***pfn*** pointer to the calc function

6.5.2.14 static v_pfn_v FIR_f64_getCalcFunction (FIR_f64_Handle *fh*) `[inline, static]`

Get the calc function.

Parameters:
← ***fh*** handle to the 'FIR_f64' object

Returns:
pfn pointer to the calc function

6.5.2.15 static void IIR_f64_setCoefficientsAPtr (IIR_f64_Handle *fh*, const float64_t ∗ *pca*) `[inline, static]`

Set the denominator coefficients pointer.

Parameters:
← ***fh*** handle to the 'IIR_f64' object
← ***pca*** pointer to the denominator coefficients

6.5.2.16 static float64_t∗ IIR_f64_getCoefficientsAPtr (IIR_f64_Handle *fh*) `[inline, static]`

Get the denominator coefficients pointer.

Parameters:
← ***fh*** handle to the 'IIR_f64' object

Returns:
pca pointer to the denominator coefficients

6.5.2.17 static void IIR_f64_setCoefficientsBPtr (IIR_f64_Handle *fh*, const float64_t ∗ *pcb*) `[inline, static]`

Set the numerator coefficients pointer.

Parameters:
← ***fh*** handle to the 'IIR_f64' object
← ***pcb*** pointer to the numerator coefficients

6.5.2.18 static float64_t∗ IIR_f64_getCoefficientsBPtr (IIR_f64_Handle *fh*) `[inline, static]`

Get the numerator coefficients pointer.

Parameters:
← ***fh*** handle to the 'IIR_f64' object

Returns:
pca pointer to the numerator coefficients

6.5.2.19 static void IIR_f64_setDelayLinePtr (IIR_f64_Handle *fh*, const float64_t ∗ *pdl*) `[inline, static]`

Set the delay line pointer.

Parameters:
← ***fh*** handle to the 'IIR_f64' object
← ***pdl*** pointer to the delay line

6.5.2.20 static float64_t∗ IIR_f64_getDelayLinePtr (IIR_f64_Handle *fh*) `[inline, static]`

Get the delay line pointer.

Parameters:
← ***fh*** handle to the 'IIR_f64' object

Returns:
pdl pointer to the delay line

6.5.2.21 static void IIR_f64_setInputPtr (IIR_f64_Handle *fh*, const float64_t ∗ *pi*) `[inline, static]`

Set the input pointer.

Parameters:
← ***fh*** handle to the 'IIR_f64' object
← ***pi*** pointer to the current input

6.5.2.22 static float64_t∗ IIR_f64_getInputPtr (IIR_f64_Handle *fh*) `[inline, static]`

Get the input pointer.

Parameters:
← ***fh*** handle to the 'IIR_f64' object

Returns:
pi pointer to the current input

6.5.2.23 static void IIR_f64_setOutputPtr (IIR_f64_Handle *fh*, const float64_t * *po*) [inline, static]

Set the output pointer.

Parameters:
← ***fh*** handle to the 'IIR_f64' object
← ***po*** pointer to the current output

6.5.2.24 static float64_t* IIR_f64_getOutputPtr (IIR_f64_Handle *fh*) [inline, static]

Get the output pointer.

Parameters:
← ***fh*** handle to the 'IIR_f64' object

Returns:
po pointer to the current output

6.5.2.25 static void IIR_f64_setScalePtr (IIR_f64_Handle *fh*, const float64_t * *psv*) [inline, static]

Set the scale value pointer.

Parameters:
← ***fh*** handle to the 'IIR_f64' object
← ***psv*** pointer to the scale values for the biquads

6.5.2.26 static float64_t* IIR_f64_getScalePtr (IIR_f64_Handle *fh*) [inline, static]

Get the scale value pointer.

Parameters:
← ***fh*** handle to the 'IIR_f64' object

Returns:
psv pointer to the scale values for the biquads

6.5.2.27 static void IIR_f64_setOrder (IIR_f64_Handle *fh*, const uint16_t *order*) [inline, static]

Set the order of the filter.

Parameters:
← ***fh*** handle to the 'IIR_f64' object
← ***order*** Order of the filter

6.5.2.28 static uint16_t IIR_f64_getOrder (IIR_f64_Handle *fh*) [inline, static]

Get the order of the filter.

Parameters:
← ***fh*** handle to the 'IIR_f64' object

Returns:
order Order of the filter

6.5.2.29 static void IIR_f64_setInitFunction (IIR_f64_Handle *fh*, const v_pfn_v *pfn*) [inline, static]

Set the init function.

Parameters:
← ***fh*** handle to the 'IIR_f64' object
← ***pfn*** pointer to the init function

6.5.2.30 static v_pfn_v IIR_f64_getInitFunction (IIR_f64_Handle *fh*) [inline, static]

Get the init function.

Parameters:
← ***fh*** handle to the 'IIR_f64' object

Returns:
pfn pointer to the init function

6.5.2.31 static void IIR_f64_setCalcFunction (IIR_f64_Handle *fh*, const v_pfn_v *pfn*) [inline, static]

Set the calc function.

Parameters:
← ***fh*** handle to the 'IIR_f64' object
← ***pfn*** pointer to the calc function

6.5.2.32 static v_pfn_v IIR_f64_getCalcFunction (IIR_f64_Handle *fh*) [inline, static]

Get the calc function.

Parameters:
← ***fh*** handle to the 'IIR_f64' object

Returns:
pfn pointer to the calc function

6.5.2.33 void FIR_f64_calc (FIR_f64_Handle *hndFIR_f64*)

Finite Impulse Response Filter.

This routine implements the non-recursive difference equation of an all-zero filter (FIR), of order N. All the coefficients of all-zero filter are assumed to be less than 1 in magnitude.

Parameters:
hndFIR_f64 Handle to the FIR_f64 object

Attention:
1. The delay and coefficients buffer must be assigned a minimum of 4 x (order + 1) words. For example, if the filter order is 31, it will have 32 taps or coefficients each a 64-bit floating point value. A minimum of $(4 * 32) = 128$ words will need to be allocated for the delay and coefficients buffer.
2. In the code example the buffer is assigned to the .ebss section while the coefficients are assigned to the .econst section.

Number of Taps (Order + 1)	Cycles
28	217
59	403
117	751

Table 6.19: Performance Data

6.5.2.34 void FIR_f64_init (FIR_f64_Handle *hndFIR_f64*)

Finite Impulse Response Filter Initialization.

Zeros out the delay line

Parameters:
hndFIR_f64 Handle to the FIR_f64 object

Attention:
Please see the description of FIR_f64_calc for more details on the space requirements for the delay line and coefficients

Number of Taps (Order + 1)	Cycles
28	130
59	254
117	486

Table 6.20: Performance Data

6.5.2.35 void IIR_f64_calc (IIR_f64_Handle *hndIIR_f64*)

Infinite Impulse Response Filter.

This routine implements the Transposed Direct form II recursive difference equation of an N pole-zero filter(IIR).

Parameters:

hndIIR_f64 Handle to the IIR_f64 object

Attention:

1. The delay line buffer must be 4∗(n_biquads∗n_delay_elements_per_biquad), since there are 4 delay elements per biquad that are double precision (64-bits) we require a total of 16∗n_biquads words For example, if the filter is an 8th order filter it would require 4 biquads (each biqaud is a 2nd order construct) hence 16∗4 = 64 words If the filter were a 9th order filter, it would require 5 biquads; the first four would be quadratic while the last is linear. The last biquad will be implemented with the B[2] and A[2] coefficients zero. We would require a total of 16∗5 = 80 words
2. In the code example the buffer is assigned to the .ebss section while the coefficients are assigned to the .econst section.

Filter Order	Number of Biquads	Cycles
2	1	89
6	3	173
12	6	299

Table 6.21: Performance Data

6.5.2.36 void IIR_f64_init (IIR_f64_Handle *hndIIR_f64*)

Infinite Impulse Response Filter Initialization.

Zeros out the delay line

Parameters:

hndIIR_f64 Handle to the IIR_f64 object

Attention:

Please see the description of IIR_f64_calc for more details on the space requirements for the delay line and coefficients

Filter Order	Number of Biquads	Cycles
2	1	38
6	3	70
12	6	118

Table 6.22: Performance Data

6.6 Vector Operations (Double Precision)

Data Structures

- complexf64_t

Functions

- void abs_DP_CV (float64_t *y, const complexf64_t *x, const uint16_t N)
- void abs_DP_CV_2 (float64_t *y, const complexf64_t *x, const uint16_t N)
- void add_DP_CSxCV (complexf64_t *y, const complexf64_t *x, const complexf64_t *c, const uint16_t N)
- void add_DP_CVxCV (complexf64_t *y, const complexf64_t *w, const complexf64_t *x, const uint16_t N)
- void iabs_DP_CV (float64_t *y, const complexf64_t *x, const uint16_t N)
- void iabs_DP_CV_2 (float64_t *y, const complexf64_t *x, const uint16_t N)
- complexf64_t mac_DP_i16RVxCV (const complexf64_t *w, const int16_t *x, const uint16_t N)
- complexf64_t mac_DP_CVxCV (const complexf64_t *w, const complexf64_t *x, const uint16_t N)
- complexf64_t mac_DP_RVxCV (const complexf64_t *w, const float64_t *x, const uint16_t N)
- uint16_t maxidx_DP_RV_2 (const float64_t *x, const uint16_t N)
- complexf64_t mean_DP_CV_2 (const complexf64_t *x, const uint16_t N)
- complexf64_t mpy_DP_CSxCS (const complexf64_t *w, const complexf64_t *x)
- void mpy_DP_CVxCV (complexf64_t *y, const complexf64_t *w, const complexf64_t *x, const uint16_t N)
- void mpy_DP_CVxCVC (complexf64_t *y, const complexf64_t *w, const complexf64_t *x, const uint16_t N)
- void mpy_DP_RMxRM (float64_t *y, const float64_t *w, const float64_t *x, const uint16_t m, const uint16_t n, const uint16_t p)
- void mpy_DP_RMxRM_2 (float64_t *y, const float64_t *w, const float64_t *x, const uint16_t m, const uint16_t n, const uint16_t p)
- void mpy_DP_RSxRV_2 (float64_t *y, const float64_t *x, const float64_t c, const uint16_t N)
- void mpy_DP_RSxRVxRV_2 (float64_t *y, const float64_t *w, const float64_t *x, const float64_t c, const uint16_t N)
- void mpy_DP_RVxCV (complexf64_t *y, const complexf64_t *w, const float64_t *x, const uint16_t N)
- void mpy_DP_RVxRV_2 (float64_t *y, const float64_t *w, const float64_t *x, const uint16_t N)
- float64_t rnd_DP_RS (const float64_t x)
- void sub_DP_CSxCV (complexf64_t *y, const complexf64_t *x, const complexf64_t *c, const uint16_t N)
- void sub_DP_CVxCV (complexf64_t *y, const complexf64_t *w, const complexf64_t *x, const uint16_t N)

6.6.1 Data Structure Documentation

6.6.1.1 complexf64_t

Definition:

```
typedef struct
{
    float64_t real;
    float64_t imag;
}
complexf64_t
```

Members:

real Real part of the data point.

imag Imaginary part of the data point.

Description:

Structure for the double precision complex data type

6.6.2 Function Documentation

6.6.2.1 abs_DP_CV

Absolute Value of a Complex Vector (Double Precision).

Prototype:

```
void
abs_DP_CV(float64_t *y,
          const complexf64_t *x,
          const uint16_t N)
```

Description:

This module computes the absolute value of a complex vector. If N is even, use abs_SP_CV_2() for better performance.

$$y[i] = \sqrt{(}x_{re}[i]^2 + x_{im}[i]^2)$$

Parameters:

y pointer to the output vector

x pointer to the input vector

N length of the x and y vectors

Cycles	Comment
73∗N + 8	Cycle count includes the call and return

Table 6.23: Performance Data

6.6.2.2 void abs_DP_CV_2 (float64_t ∗ *y*, const complexf64_t ∗ *x*, const uint16_t *N*)

Absolute Value of an Even Length Complex Vector (Double Precision).

This module computes the absolute value of an even length complex vector.

$$y[i] = \sqrt{(x_{re}[i]^2 + x_{im}[i]^2)}$$

Parameters:

y pointer to the output vector
x pointer to the input vector
N length of the x and y vectors

Attention:

N must be even

Cycles	Comment
79∗N/2 + 17	Cycle count includes the call and return

Table 6.24: Performance Data

6.6.2.3 add_DP_CSxCV

Addition (Element-Wise) of a Complex Scalar to a Complex Vector (Double Precision).

Prototype:

```
void
add_DP_CSxCV(complexf64_t *y,
             const complexf64_t *x,
             const complexf64_t *c,
             const uint16_t N)
```

Description:

This module adds a complex scalar element-wise to a complex vector.

$$y_{re}[i] = x_{re}[i] + c_{re}$$

$$y_{im}[i] = x_{im}[i] + c_{im}$$

Parameters:

y pointer to the complex output vector
x pointer to the complex input vector
c pointer to the complex input scalar
N length of the x and y vectors

Note:

The output vector must be placed in a separate RAM block to avoid memory stalls from writes that immediately follow reads to the same physical RAM block

Cycles	Comment
8∗N + 21	Cycle count includes the call and return

Table 6.25: Performance Data

6.6.2.4 add_DP_CVxCV

Addition of Two Complex Vectors (Double Precision).

Prototype:

```
void
add_DP_CVxCV(complexf64_t *y,
             const complexf64_t *w,
             const complexf64_t *x,
             const uint16_t N)
```

Description:

This module adds two complex vectors.

$$y_{re}[i] = w_{re}[i] + x_{re}[i]$$

$$y_{im}[i] = w_{im}[i] + x_{im}[i]$$

Parameters:

- ***y*** pointer to the complex output vector
- ***w*** pointer to the first complex input vector
- ***x*** pointer to the second complex input vector
- ***N*** length of the w, x and y vectors

Note:

The output vector must be placed in a separate RAM block to avoid memory stalls from writes that immediately follow reads to the same physical RAM block

Cycles	Comment
12∗N + 15	Cycle count includes the call and return

Table 6.26: Performance Data

6.6.2.5 iabs_DP_CV

Inverse Absolute Value of a Complex Vector (Double Precision).

Prototype:

```
void
iabs_DP_CV(float64_t *y,
           const complexf64_t *x,
           const uint16_t N)
```

Description:

This module computes the inverse absolute value of a complex vector.

$$y[i] = \frac{1}{\sqrt{(x_{re}[i]^2 + x_{im}[i]^2)}}$$

Parameters:

y pointer to the output vector

x pointer to the input vector

N length of the x and y vectors

Attention:

N must be at least 2

Cycles	Comment
69∗N + 8	Cycle count includes the call and return

Table 6.27: Performance Data

6.6.2.6 iabs_DP_CV_2

Inverse Absolute Value of an Even Length Complex Vector (Double Precision).

Prototype:

```
void
iabs_DP_CV_2(float64_t *y,
             const complexf64_t *x,
             const uint16_t N)
```

Description:

This module calculates the inverse absolute value of an even length complex vector.

$$y[i] = \frac{1}{\sqrt{(x_{re}[i]^2 + x_{im}[i]^2)}}$$

Parameters:

y pointer to the output vector

x pointer to the input vector

N length of the x and y vectors

Attention:

N must be even

Cycles	Comment
74∗N/2 + 17	Cycle count includes the call and return

Table 6.28: Performance Data

6.6.2.7 mac_DP_i16RVxCV

Multiply-and-Accumulate of a Real Vector (integer) and a Complex Vector (Double Precision).

Prototype:

```
complexf64_t
mac_DP_i16RVxCV(const complexf64_t *w,
```

```
const int16_t *x,
const uint16_t N)
```

Description:

This module multiplies and accumulates a 16-bit integer real vector and a floating pt. complex vector.

$$y_{re} = sum(x[i] * w_{re}[i])$$

$$y_{im} = sum(x[i] * w_{im}[i])$$

Parameters:

- ***w*** pointer to the complex input vector
- ***x*** pointer to the real input vector
- ***N*** length of the w and x vectors

Returns:

complex double precision accumulation result

Attention:

N must be a minimum of 5

Cycles	Comment
	Cycle count includes the call and return
6*N + 36	N is even
6*N + 29	N is odd

Table 6.29: Performance Data

6.6.2.8 mac_DP_CVxCV

Multiply-and-Accumulate of a Complex Vector and a Complex Vector (Double Precision).

Prototype:

```
complexf64_t
mac_DP_CVxCV(const complexf64_t *w,
             const complexf64_t *x,
             const uint16_t N)
```

Description:

This module multiplies and accumulates a complex vector and another complex vector.

$$y_{re} = \sum(w_{re}[i] * x_{re}[i] - w_{im}[i] * x_{im}[i])$$

$$y_{im} = \sum(w_{re}[i] * x_{im}[i] + w_{im}[i] * x_{re}[i])$$

Parameters:

- ***y*** complex result
- ***w*** pointer to the first complex input vector
- ***x*** pointer to the second complex input vector
- ***N*** length of the w and x vectors

Attention:
N must be a minimum of 3

Cycles	Comment
9∗N + 27	Cycle count includes the call and return

Table 6.30: Performance Data

6.6.2.9 mac_DP_RVxCV

Multiply-and-Accumulate of a Real Vector and a Complex Vector (Double Precision).

Prototype:

```
complexf64_t
mac_DP_RVxCV(const complexf64_t *w,
             const float64_t *x,
             const uint16_t N)
```

Description:
This module multiplies and accumulates a real vector and a complex vector.

$$y_{re} = \sum(x[i] * w_{re}[i])$$

$$y_{im} = \sum(x[i] * w_{im}[i])$$

Parameters:
- ***y*** complex result
- ***w*** pointer to the complex input vector
- ***x*** pointer to the real input vector
- ***N*** length of the w and x vectors

Attention:
N must be a minimum of 5

Cycles	Comment
	Cycle count includes the call and return
6∗N + 34	N is even
6∗N + 28	N is odd

Table 6.31: Performance Data

6.6.2.10 maxidx_DP_RV_2

Index of Maximum Value of an Even Length Real Array (Double Precision).

Prototype:

```
uint16_t
maxidx_DP_RV_2(const float64_t *x,
               const uint16_t N)
```

Parameters:

x pointer to the input vector

N length of the x vector

Attention:

1. N must be even
2. If more than one instance of the max value exists in x[], the function will return the index of the first occurrence (lowest index value)

Cycles	Comment
4∗N + 22	Cycle count includes the call and return

Table 6.32: Performance Data

6.6.2.11 complexf64_t mean_DP_CV_2 (const complexf64_t ∗ *x*, const uint16_t *N*)

Mean of Real and Imaginary Parts of a Complex Vector (Double Precision).

This module calculates the mean of real and imaginary parts of a complex vector.

$$y_{re} = \frac{\Sigma x_{re}}{N}$$

$$y_{im} = \frac{\Sigma x_{im}}{N}$$

Parameters:

x pointer to the input vector

N length of the x vector

Attention:

N must be even

Cycles	Comment
5∗N + 37	Cycle count includes the call and return

Table 6.33: Performance Data

6.6.2.12 complexf64_t mpy_DP_CSxCS (const complexf64_t ∗ *w*, const complexf64_t ∗ *x*)

Complex Multiply of Two Double Precision Numbers.

This module multiplies two double precision complex values.

$$y_{re} = w_{re} * x_{re} - w_{im} * x_{im}$$

$$y_{im} = w_{re} * x_{im} + w_{im} * x_{re}$$

Parameters:

w pointer to the first complex input

x pointer to the second complex input

Returns:

complex product of the first and second complex input

Cycles	Comment
28	Cycle count includes the call and return

Table 6.34: Performance Data

6.6.2.13 void mpy_DP_CVxCV (complexf64_t ∗ *y*, const complexf64_t ∗ *w*, const complexf64_t ∗ *x*, const uint16_t *N*)

Complex Multiply of Two Complex Vectors (Double Precision).

This module performs complex multiplication on two input complex vectors.

$$y_{re}[i] = w_{re}[i] * x_{re}[i] - w_{im}[i] * x_{im}[i]$$

$$y_{im}[i] = w_{re}[i] * x_{im}[i] + w_{im}[i] * x_{re}[i]$$

Parameters:

y pointer to the complex product of the first and second complex vectors

w pointer to the first complex input vector

x pointer to the second complex input vector

N length of the w, x and y vectors

Note:

The output vector must be placed in a separate RAM block to avoid memory stalls from writes that immediately follow reads to the same physical RAM block

Cycles	Comment
20∗N + 16	Cycle count includes the call and return

Table 6.35: Performance Data

6.6.2.14 void mpy_DP_CVxCVC (complexf64_t ∗ *y*, const complexf64_t ∗ *w*, const complexf64_t ∗ *x*, const uint16_t *N*)

Multiplication of a Complex Vector and the Complex Conjugate of another Vector (Double Precision).

This module multiplies a complex vector (w) and the complex conjugate of another complex vector (x).

$$x^*_{re}[i] = x_{re}[i]$$

$$x^*_{im}[i] = -x_{im}[i]$$

$$y_{re}[i] = w_{re}[i] * x_{re}[i] - w_{im}[i] * x^*_{im}[i]$$

$$= w_{re}[i] * x_{re}[i] + w_{im}[i] * x_{im}[i]$$

$$y_{im}[i] = w_{re}[i] * x^*_{im}[i] + w_{im}[i] * x_{re}[i]$$

$$= w_{im}[i] * x_{re}[i] - w_{re}[i] * x_{im}[i]$$

Parameters:

y pointer to the complex conjugate product of the first and second complex vectors

w pointer to the first complex input vector
x pointer to the second complex input vector
N length of the w, x and y vectors

Note:
The output vector must be placed in a separate RAM block to avoid memory stalls from writes that immediately follow reads to the same physical RAM block

Cycles	Comment
20∗N + 16	Cycle count includes the call and return

Table 6.36: Performance Data

6.6.2.15 void mpy_DP_RMxRM (float64_t ∗ *y*, const float64_t ∗ *w*, const float64_t ∗ *x*, const uint16_t *m*, const uint16_t *n*, const uint16_t *p*)

Multiplication of Two Real Matrices (Double Precision).

This module multiplies two real matrices.

$$y[] = w[] * x[]$$

Parameters:
y pointer to result matrix
w pointer to 1st source matrix
x pointer to 2nd source matrix
m number of rows in the first and output matrices
n number of columns in the first and rows in the second matrix
p number of columns in the second and output matrices

Attention:
1. There are no restrictions on the values for n, m, and pwith this function.
2. If n is even and at least 4, you can use mpy_DP_RMxRM_2() for better performance if desired.

Cycles	Comment
8∗m∗n∗p + overhead	Cycle count includes the call and return
m=2 n=8, p=2 takes ∼362 cycles	versus 8∗m∗n∗p = 256
m=8, n=8, p=8 takes ∼5162 cycles	versus 8∗m∗n∗p = 4096
m=16, n=16, p=16 takes ∼36794 cycles	versus 8∗m∗n∗p = 32768

Table 6.37: Performance Data

6.6.2.16 void mpy_DP_RMxRM_2 (float64_t ∗ *y*, const float64_t ∗ *w*, const float64_t ∗ *x*, const uint16_t *m*, const uint16_t *n*, const uint16_t *p*)

Multiplication of Two Real Matrices (n even, Double Precision).

This module multiplies two real matrices.

$$y[] = w[] * x[]$$

Parameters:

y pointer to result matrix
w pointer to 1st source matrix
x pointer to 2nd source matrix
m number of rows in the first and output matrices
n number of columns in the first and rows in the second matrix
p number of columns in the second and output matrices

Attention:

1. n must be even and at least 4. If not, use mpy_DP_RMxRM().
2. There are no restrictions on the values of m and p with this function.

Cycles	Comment
4.5∗m∗n∗p + overhead	Cycle count includes the call and return
m=2 n=8, p=2 takes ∼287 cycles	versus 4.5∗m∗n∗p = 144
m=8, n=8, p=8 takes ∼3947 cycles	versus 4.5∗m∗n∗p = 2304
m=16, n=16, p=16 takes ∼26299 cycles	versus 4.5∗m∗n∗p = 18432

Table 6.38: Performance Data

6.6.2.17 void mpy_DP_RSxRV_2 (float64_t ∗ *y*, const float64_t ∗ *x*, const float64_t *c*, const uint16_t *N*)

Multiplication of a Real scalar and a Real Vector (Double Precision).

This module multiplies a real scalar and a real vector.

$$y[i] = c * x[i]$$

Parameters:

y pointer to the product of the scalar and a real vector
x pointer to the real input vector
c scalar multiplier
N length of the x and y vectors

Attention:

N must be even and a minimum of 4.

Note:

The output vector must be placed in a separate RAM block to avoid memory stalls from writes that immediately follow reads to the same physical RAM block

Cycles	Comment
4∗N + 18	Cycle count includes the call and return

Table 6.39: Performance Data

6.6.2.18 void mpy_DP_RSxRVxRV_2 (float64_t ∗ *y*, const float64_t ∗ *w*, const float64_t ∗ *x*, const float64_t *c*, const uint16_t *N*)

Multiplication of a Real Scalar, a Real Vector, and another Real Vector (Double Precision).

This module multiplies a real scalar with a real vector and another real vector.

$$y[i] = c * w[i] * x[i]$$

Parameters:

- ***y*** pointer to the product of the scalar and two real vectors
- ***w*** pointer to the first real input vector
- ***x*** pointer to the second real input vector
- ***c*** scalar multiplier
- ***N*** length of the w, x and y vectors

Attention:

N must be even and a minimum of 4.

Note:

The output vector must be placed in a separate RAM block to avoid memory stalls from writes that immediately follow reads to the same physical RAM block

Cycles	Comment
6∗N + 27	Cycle count includes the call and return

Table 6.40: Performance Data

6.6.2.19 void mpy_DP_RVxCV (complexf64_t ∗ *y*, const complexf64_t ∗ *w*, const float64_t ∗ *x*, const uint16_t *N*)

Multiplication of a Real Vector and a Complex Vector (Double Precision).

This module multiplies a real vector and a complex vector.

$$y_{re}[i] = x[i] * w_{re}[i]$$

$$y_{im}[i] = x[i] * w_{im}[i]$$

Parameters:

- ***y*** pointer to the product of the real and complex vectors
- ***w*** pointer to the complex input vector
- ***x*** pointer to the real input vector
- ***N*** length of the w, x and y vectors

Attention:
N must be at least 2

Note:
The output vector must be placed in a separate RAM block to avoid memory stalls from writes that immediately follow reads to the same physical RAM block

Cycles	Comment
10∗N + 15	Cycle count includes the call and return

Table 6.41: Performance Data

6.6.2.20 void mpy_DP_RVxRV_2 (float64_t ∗ *y*, const float64_t ∗ *w*, const float64_t ∗ *x*, const uint16_t *N*)

Multiplication of a Real Vector and a Real Vector.

This module multiplies two real vectors (Double Precision).

$$y[i] = w[i] * x[i]$$

Parameters:
- ***y*** pointer to the product of two real vectors
- ***w*** pointer to the first real input vector
- ***x*** pointer to the second real input vector
- ***N*** length of the w, x and y vectors

Attention:
N must be even and a minimum of 4.

Note:
The output vector must be placed in a separate RAM block to avoid memory stalls from writes that immediately follow reads to the same physical RAM block

Cycles	Comment
6∗N + 17	Cycle count includes the call and return

Table 6.42: Performance Data

6.6.2.21 float64_t rnd_DP_RS (const float64_t *x*)

Rounding (Unbiased) of a Floating Point Scalar (Double Precision).

numerical examples: rnd_DP_RS(+4.4) = +4.0 \ rnd_DP_RS(-4.4) = -4.0 \ rnd_DP_RS(+4.5) = +5.0 \ rnd_DP_RS(-4.5) = -5.0 \ rnd_DP_RS(+4.6) = +5.0 \ rnd_DP_RS(-4.6) = -5.0 \

Parameters:
- ***x*** input value

Returns:
rounded

Cycles	Comment
26	Cycle count includes the call and return

Table 6.43: Performance Data

6.6.2.22 void sub_DP_CSxCV (complexf64_t * *y*, const complexf64_t * *x*, const complexf64_t * *c*, const uint16_t *N*)

Subtraction of a Complex Scalar from a Complex Vector (Double Precision).

This module subtracts a complex scalar from a complex vector.

$$y_{re}[i] = x_{re}[i] - c_{re}$$

$$y_{im}[i] = x_{im}[i] - c_{im}$$

Parameters:

y pointer to the difference of a complex scalar from a complex vector
x pointer to the complex input vector
c pointer to the complex input scalar
N length of the x and y vectors

Attention:

N must be at least 2

Note:

The output vector must be placed in a separate RAM block to avoid memory stalls from writes that immediately follow reads to the same physical RAM block

Cycles	Comment
8*N + 21	Cycle count includes the call and return

Table 6.44: Performance Data

6.6.2.23 void sub_DP_CVxCV (complexf64_t * *y*, const complexf64_t * *w*, const complexf64_t * *x*, const uint16_t *N*)

Subtraction of a Complex Vector and another Complex Vector (Double Precision).

This module subtracts a complex vector from another complex vector.

$$y_{re}[i] = w_{re}[i] - x_{re}[i]$$

$$y_{im}[i] = w_{im}[i] - x_{im}[i]$$

Parameters:

y pointer to the difference of two complex vectors
w pointer to the first complex input vector
x pointer to the second complex input vector
N length of the w, x and y vectors

Attention:

N must be at least 2

Note:

The output vector must be placed in a separate RAM block to avoid memory stalls from writes that immediately follow reads to the same physical RAM block

Cycles	Comment
12∗N + 15	Cycle count includes the call and return

Table 6.45: Performance Data

7 Benchmarks

The benchmarks for the single precision library routines were obtained with the following compiler settings:

```
-v28 -mt -ml -g --diag_warning=225 --float_support=fpu32 --tmu_support=tmu0
```

while the benchmarks for the double precision library routines were obtained with the following compiler settings:

```
-v28 -mt -ml -g --diag_warning=225 --float_support=fpu64
```

Table. 7.1 summarizes the performance metrics for the single precision library routines. These numbers were obtained using profiling the code in the examples directory by simulating it on pre silicon platform.

Single Precision Routine	Constraints	Cycles [1]
	FFT	
CFFT_f32	N = 64	2334
	N = 128	5032
	N = 256	11024
	N = 512	24250
	N = 1024	53220
CFFT_f32_brev	N = 64	1621
	N = 128	3217
	N = 256	6352
	N = 512	13968
	N = 1024	28500
CFFT_f32i	N = 64	2407
	N = 128	5248
	N = 256	11591
	N = 512	25648
	N = 1024	56535
CFFT_f32t	N = 64	2334
	N = 128	5032
	N = 256	11026
	N = 512	24250
	N = 1024	53220
CFFT_f32it	N = 64	2407
	N = 128	5248
	N = 256	11593
	N = 512	25648
	N = 1024	56535
CFFT_f32u	N = 64	2787
	N = 128	5933
	N = 256	12821
	N = 512	27839
	N = 1024	60393

Continued on next page

Table 7.1 – continued from previous page

Single Precision Routine	Constraints	Cycles [1]
CFFT_f32ut	N = 64	2787
	N = 128	5933
	N = 256	12821
	N = 512	27839
	N = 1024	60393
CFFT_f32_win	N = 64	316
	N = 128	604
	N = 256	1180
	N = 512	2332
	N = 1024	4636
CFFT_f32_win_dual	N = 64	595
	N = 128	1171
	N = 256	2323
	N = 512	4627
	N = 1024	9235
CFFT_f32_sincostable [2]	N/A	N/A
CFFT_f32_mag	N = 64	1181
	N = 128	2333
	N = 256	4635
	N = 512	9243
	N = 1024	18459
CFFT_f32_mag_TMU0	N = 64	342
	N = 128	662
	N = 256	1302
	N = 512	2582
	N = 1024	5142
CFFT_f32s_mag	N = 64	1284
	N = 128	2506
	N = 256	4942
	N = 512	9812
	N = 1024	19546
CFFT_f32s_mag_TMU0	N = 64	412
	N = 128	769
	N = 256	1476
	N = 512	2889
	N = 1024	5710
CFFT_f32_phase [3]	N = 64	63223 / 3797
	N = 128	110204 / 7573
	N = 256	242449 / 14866
	N = 512	485200 / 29714
	N = 1024	1001691 / 60434
CFFT_f32_phase_TMU0	N = 64	480
	N = 128	928
	N = 256	1824
	N = 512	3615
	N = 1024	7199
CFFT_f32_pack	N = 64	564
	N = 128	1076

Continued on next page

Table 7.1 – continued from previous page

Single Precision Routine	Constraints	Cycles [1]
	N = 256	2100
	N = 512	4148
	N = 1024	8243
CFFT_f32_unpack	N = 64	562
	N = 128	1136
	N = 256	2098
	N = 512	4399
	N = 1024	8241
ICFFT_f32	N = 64	2808
	N = 128	5954
	N = 256	12843
	N = 512	27861
	N = 1024	60415
ICFFT_f32t	N = 64	2921
	N = 128	6180
	N = 256	13549
	N = 512	29271
	N = 1024	64256
RFFT_f32	N = 64	1281
	N = 128	2779
	N = 256	6149
	N = 512	13674
	N = 1024	30356
	N = 2048	67002
RFFT_f32_win	N = 64	308
	N = 128	596
	N = 256	1172
	N = 512	2324
	N = 1024	4628
	N = 2048	8210
RFFT_f32u	N = 64	1393
	N = 128	3003
	N = 256	6597
	N = 512	14570
	N = 1024	32148
	N = 2048	70586
RFFT_adc_f32	N = 64	1295
	N = 128	2769
	N = 256	6059
	N = 512	13360
	N = 1024	29466
	N = 2048	64709
RFFT_adc_f32_win	N = 64	346
	N = 128	666
	N = 256	1306
	N = 512	2586
	N = 1024	5146
RFFT_adc_f32u	N = 64	1415

Continued on next page

Table 7.1 – continued from previous page

Single Precision Routine	**Constraints**	**Cycles** [1]
	N = 128	3009
	N = 256	6539
	N = 512	14320
	N = 1024	31386
	N = 2048	68549
RFFT_f32_mag	N = 64	635
	N = 128	1243
	N = 256	2459
	N = 512	4888
	N = 1024	9752
	N = 2048	19476
RFFT_f32_mag_TMU0	N = 64	161
	N = 128	289
	N = 256	545
	N = 512	1055
	N = 1024	2079
	N = 2048	4123
RFFT_f32s_mag	N = 64	723
	N = 128	1336
	N = 256	2557
	N = 512	4990
	N = 1024	9859
	N = 2048	19588
RFFT_f32s_mag_TMU0	N = 64	268
	N = 128	466
	N = 256	856
	N = 512	1375
	N = 1024	2661
	N = 2048	5223
RFFT_f32_phase [3]	N = 64	28470 / 1949
	N = 128	58674 / 3933
	N = 256	104929 / 7901
	N = 512	235592 / 16835
	N = 1024	477211 / 31707
	N = 2048	846597 / 61402
RFFT_f32_phase_TMU0	N = 64	279
	N = 128	535
	N = 256	1047
	N = 512	2068
	N = 1024	4116
	N = 2048	8208
RFFT_f32_sincostable [2]	N/A	N/A
	Vector	
abs_SP_CV	N (vector size)	28*N + 9
abs_SP_CV_2	N (vector size)	18*N + 22
abs_SP_CV_TMU0	N (vector size)	30, N = 1
	1<N<8 and N even	7.5*(N)+21
		Continued on next page

Table 7.1 – continued from previous page

Single Precision Routine	Constraints	Cycles [1]
	1<N<8 and N odd	7.5*(N-1)+38
	N>=8 and N even	4*(N-6)+56
	N>=8 and N odd	4*(N-7)+73
add_SP_CSxCV	N (vector size)	4*N + 19
add_SP_CVxCV	N (vector size)	6*N + 15
iabs_SP_CV	N (vector size)	25*N + 13
iabs_SP_CV_2	N (vector size)	15*N + 22
iabs_SP_CV_TMU0	N =1 (vector size)	35
	1<N<8 and N even	10*(N)+24
	1<N<8 and N odd	10*(N-1)+46
	N>=8 and N even	5*(N-6)+67
	N>=8 and N odd	5*(N-7)+89
mac_SP_CVxCV	N (vector size)	5*N + 24
mac_SP_RVxCV	N (vector size)	3*N + 27
mac_SP_i16RVxCV	N (vector size, even) N (vector size, odd)	3*N + 28 3*N + 29
maxidx_SP_RV_2	N (vector size)	3*N + 21
mean_SP_CV_2	N (vector size)	2*N + 34
median_noreorder_SP_RV [4]	N/A	N/A
median_SP_RV [4]	N/A	N/A
mpy_SP_CSxCS	N = 1	19
mpy_SP_CVxCV	N (vector size)	10*N + 16
mpy_SP_CVxCVC	N (vector size)	11*N + 16
mpy_SP_RSxRV_2	N (vector size)	2*N + 15
mpy_SP_RSxRVxRV_2	N (vector size)	3*N + 22
mpy_SP_RVxCV	N (vector size)	5*N + 15
mpy_SP_RVxRV_2	N (vector size)	3*N + 17
qsort_SP_RV [4]	N/A	N/A
rnd_SP_RS	N = 1	18
sub_SP_CSxCV	N (vector size)	4*N + 19
sub_SP_CVxCV	N (vector size)	6*N + 15
Filter		
FIR_f32_init [5]	N = 28	79
	N = 59	141
	N = 117	257
FIR_f32_calc [5]	N = 28	103
	N = 59	165
	N = 117	281
IIR_f32_init [6]	2 /1	30
	6 /3	46
	12/6	70
IIR_f32_calc [6]	2 /1	68
	6 /3	116
	12/6	188
Utility		

Continued on next page

Table 7.1 – continued from previous page

Single Precision Routine	Constraints	Cycles [1]
memcpy_fast [8]	N (memory size)	N + 20
memcpy_fast_far [9]	N (memory size)	N/A
memset_fast [8]	N (memory size)	N + 20

Table 7.1: Benchmark for the Single Precision FPU Library Routines.

Table. 7.2 summarizes the performance metrics for the critical single precision library routines used in their corresponding example projects. These numbers were obtained using profiling the code in the examples directory by executing on hardware in RAM configuration and the default input vector sizes as seen in the package.

Executable file	Function name	Cycles
cfft_f32.out	CFFT_f32 [1]	4998
cfft_f32_unaligned.out	CFFT_f32u [1]	5935
cfft_f32_inplace.out	CFFT_f32i [1]	5245
cfft_f32_windowed.out	CFFT_f32_win_dual [1]	4998
icfft_f32.out	ICFFT_f32 [1]	12846
irfft_f32.out	ICFFT_f32t [2]	5955
rfft_adc_f32.out	RFFT_adc_f32 [1]	6060
rfft_adc_f32_unaligned.out	RFFT_adc_f32u [1]	6540
rfft_adc_f32_windowed.out	RFFT_adc_f32_win [1]	6061
rfft_alt_f32.out	RFFT_f32 [2]	6151
rfft_alt_f32_unaligned.out	RFFT_f32u [2]	6599
rfft_alt_f32_windowed.out CFFT_f32t [2]	RFFT_f32 7097	6151 rfft_f32.out
fir_f32.out	FIR_f32_calc	156
iir_f32.out	IIR_f32_calc	107
abs_SP_CV.out	abs_SP_CV	1828
add_SP_CSxCV.out	add_SP_CSxCV	389
add_SP_CVxCV.out	add_SP_CVxCV	391
iabs_SP_CV.out	iabs_SP_CV	1668
mac_SP_CVxCV.out	mac_SP_CVxCV	337
mac_SP_i16RVxCV.out	mac_SP_i16RVxCV	217

Continued on next page

[1] Includes call and return instructions.
[2] This function is written in C and not optimized.
[3] Numbers to the left of / were obtained using the standard run time support library while those to the right were with the fast runtime support library.
[4] The cycles for this function are data dependent and therefore the benchmark cannot be provided.
[5] N is the number of filter taps. For e.g. N = 28, cycle count = 103.
[6] To the left of the slash is the filter order, to the right the number of biquads.
[7] This assumes source and destination are located in different internal RAM blocks.
[8] Refer to the API documentation of this function for benchmark details

Table 7.2 – continued from previous page

Executable file	Function name	Cycles
mac_SP_RVxCV.out	mac_SP_RVxCV	273
maxidx_SP_RV_2.out	maxidx_SP_RV_2	267
mean_SP_CV_2.out	mean_SP_CV_2	214
mpy_SP_CSxCS.out	mpy_SP_CSxCS	9
mpy_SP_CVxCV.out	mpy_SP_CVxCV	648
mpy_SP_CVxCVC.out	mpy_SP_CVxCVC	712
mpy_SP_RMxRM.out	mpy_SP_RMxRM	8880
mpy_SP_RSxRV_2.out	mpy_SP_RSxRV_2	195
mpy_SP_RVxCV.out	mpy_SP_RVxCV	327
mpy_SP_RVxRV_2.out	mpy_SP_RVxRV_2	261
rnd_SP_RS.out	rnd_SP_RS	11
sub_SP_CSxCV.out	sub_SP_CSxCV	389
sub_SP_CVxCV.out	sub_SP_CVxCV	391

Table 7.2: Benchmark for the Single Precision FPU Library Routines used in their corresponding example projects

Table. 7.3 summarizes the performance metrics for the double precision library routines. These numbers were obtained using profiling the code in the examples directory by simulating it on pre silicon platform.

Double Precision Routine	Constraints	Cycles [1]
	FFT	
CFFT_f64_calc	N = 64	3805
	N = 128	8649
	N = 256	19571
	N = 512	43935
	N = 1024	98683
CFFT_f64_u_calc	N = 64	4107
	N = 128	9254
	N = 256	20783
	N = 512	46362
	N = 1024	102648
CFFT_f64_mag	N = 64	2696
	N = 128	5288
	N = 256	10470
	N = 512	20838
	N = 1024	41574
CFFT_f64_s_mag	N = 64	2771
	N = 128	5427
		Continued on next page

[1] Cycles are shown for 256-point operation

[2] Cycles are shown for 512-point operation

Table 7.3 – continued from previous page

Double Precision Routine	**Constraints**	**Cycles** [1]
	N = 256	10738
	N = 512	21362
	N = 1024	42610
CFFT_f64_phase [3]	N = 64	
	N = 128	
	N = 256	
	N = 512	
	N = 1024	
CFFT_f64_pack	N = 64	761
	N = 128	1465
	N = 256	2873
	N = 512	5688
	N = 1024	11320
CFFT_f64_unpack	N = 64	756
	N = 128	1460
	N = 256	2868
	N = 512	5683
	N = 1024	11315
ICFFT_f64_s_calc	N = 64	4537
	N = 128	10132
	N = 256	22559
	N = 512	49927
	N = 1024	109746
ICFFT_f64_su_calc	N = 64	4876
	N = 128	10808
	N = 256	23905
	N = 512	52619
	N = 1024	115129
RFFT_f64_calc	N = 64	1729
	N = 128	3914
	N = 256	8915
	N = 512	20219
	N = 1024	45476
RFFT_f64_u_calc	N = 64	1903
	N = 128	4256
	N = 256	9593
	N = 512	21569
	N = 1024	48170
RFFT_f64_adc_calc	N = 64	2046
	N = 128	4551
	N = 256	10195
	N = 512	22780
	N = 1024	50597
RFFT_f64_u_adc_calc	N = 64	2180
	N = 128	4813
	N = 256	10709
	N = 512	23806
	N = 1024	52647

Continued on next page

Table 7.3 – continued from previous page

Double Precision Routine	Constraints	Cycles [1]
RFFT_f64_adc_win	N = 64	346
	N = 128	666
	N = 256	1306
	N = 512	2586
	N = 1024	5146
RFFT_f64_mag	N = 64	1332
	N = 128	2628
	N = 256	5220
	N = 512	10402
	N = 1024	20770
RFFT_f64_s_mag	N = 64	1403
	N = 128	2725
	N = 256	5449
	N = 512	10818
	N = 1024	21575
RFFT_f64_phase [3]	N = 64	
	N = 128	
	N = 256	
	N = 512	
	N = 1024	
Vector		
abs_DP_CV	N (vector size)	73*N + 8
abs_DP_CV_2	N (vector size)	79*N/2 + 17
add_DP_CSxCV	N (vector size)	8*N + 21
add_DP_CVxCV	N (vector size)	12*N + 15
iabs_DP_CV	N (vector size)	69*N + 8
iabs_DP_CV_2	N (vector size)	74*N/2 + 17
mac_DP_CVxCV	N (vector size)	9*N + 27
mac_DP_RVxCV	N (vector size, even) N (vector size, odd)	6*N + 34 6*N + 28
mac_DP_i16RVxCV	N (vector size, even) N (vector size, odd)	6*N + 36 6*N + 29
maxidx_DP_RV_2	N (vector size)	4*N + 22
mean_DP_CV_2	N (vector size)	5*N + 37
mpy_DP_CSxCS		28
mpy_DP_CVxCV	N (vector size)	20*N + 16
mpy_DP_CVxCVC	N (vector size)	20*N + 16
mpy_DP_RMxRM	m, n, p (array dimensions)	8*m*n*p + overhead
mpy_DP_RMxRM	m, n, p (array dimensions)	4.5*m*n*p + overhead
mpy_DP_RSxRV_2	N (vector size)	4*N + 18
mpy_DP_RSxRVxRV_2	N (vector size)	6*N + 27
mpy_DP_RVxCV	N (vector size)	10*N + 15
mpy_DP_RVxRV_2	N (vector size)	6*N + 17
rnd_DP_RS		26
sub_DP_CSxCV	N (vector size)	8*N + 21
sub_DP_CVxCV	N (vector size)	12*N + 15

Continued on next page

Table 7.3 – continued from previous page

Double Precision Routine	**Constraints**	**Cycles** [1]
Filter		
FIR_f64_init [5]	N = 28	130
	N = 59	254
	N = 117	486
FIR_f64_calc [5]	N = 28	217
	N = 59	403
	N = 117	751
IIR_f64_init [6]	2 /1	38
	6 /3	70
	12/6	118
IIR_f64_calc [6]	2 /1	89
	6 /3	173
	12/6	299

Table 7.3: Benchmark for the Double Precision FPU Library Routines.

Table. 7.4 summarizes the performance metrics for the critical double precision library routines used in their corresponding example projects. These numbers were obtained using profiling the code in the examples directory by executing on hardware.

Executable file	**Function name**	**Cycles**
cfft_f64.out	CFFT_f64 [1]	43928
cfft_f64_unaligned.out	CFFT_f64u [1]	46355
icfft_f64.out	ICFFT_f64 [1]	49921
icfft_f64_unaligned.out	ICFFT_f64u [1]	52613
irfft_f64.out	ICFFT_f64t [1]	22550
rfft_adc_f64.out	RFFT_adc_f64 [1]	22772
rfft_adc_f64_unaligned.out	RFFT_adc_f64u [1]	23798
rfft_adc_f64_windowed.out	RFFT_adc_f64_win [1]	2577
rfft_alt_f64.out	RFFT_f64 [1]	20211
rfft_alt_f64_unaligned.out	RFFT_f64u [1]	21561
rfft_f64.out	CFFT_f64t [1]	10462
fir_f64.out	FIR_f64_calc	391
iir_f64.out	IIR_f64_calc	164
abs_DP_CV.out	abs_DP_CV	4736

Continued on next page

[1] Includes call and return instructions.

[3] Numbers to the left of / were obtained using the standard run time support library while those to the right were with the fast runtime support library.

[5] N is the number of filter taps. For e.g. N = 28, cycle count = 103.

[6] To the left of the slash is the filter order, to the right the number of biquads.

Table 7.4 – continued from previous page

Executable file	Function name	Cycles
add_DP_CSxCV.out	add_DP_CSxCV	527
add_DP_CVxCV.out	add_DP_CVxCV	775
iabs_DP_CV.out	iabs_DP_CV	4480
mac_DP_CVxCV.out	mac_DP_CVxCV	594
mac_DP_i16RVxCV.out	mac_DP_i16RVxCV	412
mac_DP_RVxCV.out	mac_DP_RVxCV	410
maxidx_DP_RV_2.out	maxidx_DP_RV_2	271
mean_DP_CV_2.out	mean_DP_CV_2	353
mpy_DP_CSxCS.out	mpy_DP_CSxCS	18
mpy_DP_CVxCV.out	mpy_DP_CVxCV	1288
mpy_DP_CVxCVC.out	mpy_DP_CVxCVC	1288
mpy_DP_RMxRM.out	mpy_DP_RMxRM	12209
mpy_DP_RSxRV_2xRV_2.out	mpy_DP_RSxRV_2xRV_2	401
mpy_DP_RSxRV_2.out	mpy_DP_RSxRV_2	263
mpy_DP_RVxCV.out	mpy_DP_RVxCV	647
mpy_DP_RVxRV_2.out	mpy_DP_RVxRV_2	393
rnd_DP_RS.out	rnd_DP_RS	18
sub_DP_CSxCV.out	sub_DP_CSxCV	527
sub_DP_CVxCV.out	sub_DP_CVxCV	775

Table 7.4: Benchmark for the Double Precision FPU Library Routines used in their corresponding example projects

[1] Cycles are shown for 512-point operation

8 Revision History

V2.11.00.00: Minor Update

Bug fix in 2837x$_{r}fft_{a}dc_{r}texample(datasectionforphasebuf)$

V2.10.00.00: Minor Update

Added support for F28P55x in cfft$_{f}32.$

V2.09.00.00: Minor Update

Fixes for errors in some examples

V2.08.00.00: Minor Update

Added support for F28P65x in cfft$_{f}32.$

V2.07.00.00: Minor Update

Fixes for errors in some examples

V2.06.00.00: Minor Update

Fixes for errors in some examples

V2.05.00.00: Minor Update

Added support for f280015xx devices for all single precision examples. Migrated to tools 22.6.0 for all the examples.

V2.04.00.00: Minor Update

Added support for f280013x devices for all single precision examples.

V2.03.00.00: Minor Update

Added support for f28003x devices for all single precision examples. Migrated to tools 20.2.1 for all the examples.

V2.02.00.00: Minor Update

Added support for f28002x devices for all single precision examples.

V2.01.00.00: Major Update

- All the libraries and F2838x examples have been migrated from COFF to EABI configuration.
- All the FFT examples are provided with RAM_ROMTABLES configuration to make use pre-stored FFT tables in ROM while computation.
- FLASH configuation is added to the examples.
- F2837xD examples are also restored exactly (without any modification) from the last release.
- Added double precision version of existing algorithms
  - **FFT**
    - cfft_f64.asm
    - cfft_f64_mag.asm
    - cfft_f64_pack.asm
    - cfft_f64_phase.asm
    - cfft_f64_scaled_mag.asm
    - cfft_f64_unaligned.asm
    - cfft_f64_unpack.asm
    - fft_tables.asm
    - icfft_f64.asm
    - icfft_f64_unaligned.asm
    - rfft_f64.asm
    - rfft_f64_adc.asm
    - rfft_f64_adc_unaligned.asm
    - rfft_f64_adc_window.asm

    * rfft_f64_mag.asm
    * rfft_f64_phase.asm
    * rfft_f64_scaled_mag.asm
    * rfft_f64_unaligned.asm
  - **Filter**
    * fir_f64.asm
    * iir_f64.asm
  - **Vector**
    * abs_DP_CV.asm
    * abs_DP_CV_2.asm
    * add_DP_CSxCV.asm
    * add_DP_CVxCV.asm
    * iabs_DP_CV.asm
    * iabs_DP_CV_2.asm
    * mac_DP_CVxCV.asm
    * mac_DP_i16RVxCV.asm
    * mac_DP_RVxCV.asm
    * maxidx_DP_RV_2.asm
    * mean_DP_CV_2.asm
    * mpy_DP_CSxCS.asm
    * mpy_DP_CVxCV.asm
    * mpy_DP_CVxCVC.asm
    * mpy_DP_RMxRM.asm
    * mpy_DP_RMxRM_2.asm
    * mpy_DP_RSxRVxRV_2.asm
    * mpy_DP_RSxRV_2.asm
    * mpy_DP_RVxCV.asm
    * mpy_DP_RVxRV_2.asm
    * rnd_DP_RS.asm
    * sub_DP_CSxCV.asm
    * sub_DP_CVxCV.asm
- Added new single precision algorithms
  - iir_f32.asm
  - mac_SP_CVxCV
  - CFFT_f32_brev.asm
  - CFFT_f32i.asm
  - CFFT_f32it.asm
  - CFFT_f32_unpack.asm
  - CFFT_f32_pack.asm
  - RFFT_f32_adc_win.asm
- Added examples for all the double and single precision algorithms that can easily support all compatible hardware targets
- The benchmarks for all the double and single precision routines are updated and included in this user guide

V1.50.00.00: Moderate Update

- Added TMU0 supported phase and magnitude functions

  - CFFT_f32_mag_TMU0
  - CFFT_f32s_mag_TMU0
  - RFFT_f32_mag_TMU0
  - RFFT_f32s_mag_TMU0
  - CFFT_f32_phase_TMU0
  - RFFT_f32_phase_TMU0
- Corrected issue with global pointer alignment (causing data corruption) for the FFT functions (CFFT_f32, CFFT_f32u, ICFFT_f32)
- Optimized magnitude functions (non-TMU)
  - CFFT_f32_mag
  - CFFT_f32s_mag
  - RFFT_f32_mag
  - RFFT_f32s_mag
- Removed scaling prior to arc-tangent call in the RFFT phase function, RFFT_f32_phase
- Library now has a single build configuration, ISA_C28FPU32
- Added a memcpy function to copy data from far (> 22-bit) memory, memcpy_fast_far
- Added windowing functions and tables, along with supporting examples
  - CFFT32_f32_win
  - CFFT32_f32_win_dual
  - RFFT_f32_win
- Added routine for a Multiply-Accumulate of a real vector and a complex vector (both single precision floating point), mac_SP_RVxCV
- Added routine for a Multiply-Accumulate of a real vector (16-bit signed integer) and a complex vector (single precision float), mac_SP_i16RVxCV
- Added twiddle factor tables and alternate CFFT (aligned and unaligned), ICFFT implementations to use this table
  - CFFT_f32t
  - CFFT_f32ut
  - ICFFT_f32t
- Updated all examples to work on the F2837xD

V1.40.00.00: Moderate Update
- Revised documentation
- Re-factored all library and example projects to use CGT v6.2.4
- Updated all examples to work with CCS v5
- Added TMU0 build configuration to the library and an example to demonstrate functions that use the TMU
- Corrected circular buffer limitation (256 words) for the FIR filter implementation by using C2xLP addressing mode which permits a circular buffer up to a maximum size of 65536 words

V1.31: Minor Update
- Revised documentation
- Updated median_SP_RV() routine

V1.30: Moderate Update
- Added vector and matrix functions and examples
- Added Inverse complex FFT and example

- Revised benchmark numbers
- Revised alignment requirements for FFT's

V1.20: Moderate Update
Added equiripple FIR filter function

V1.10: Moderate Update
Includes the complex FFT and real FFT with 12-bit ADC fixed-point input supporting functions

V1.00: Initial Release

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