C28x Floating Point Unit DSP Library

USER'S GUIDE

FPU-DSP-LIB-UG-V130

Copyright

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

Revision Information

This is version V130 of this document, last updated on January 03, 2012.

Table of Contents

1 Introduction

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

2 Library Structure

As installed, the C28x FPU Library is partitioned into a well-defined directory structure. By default, the library and source code is installed into the default controlSUITE directory,

```
C:\TI\controlSUITE\libs\dsp\FPU\VERSION
```

VERSION indicates the current revision of the FPU library. Figure. 2.1 shows the directory structure while the subsequent table 2.1 provides a description for each folder.

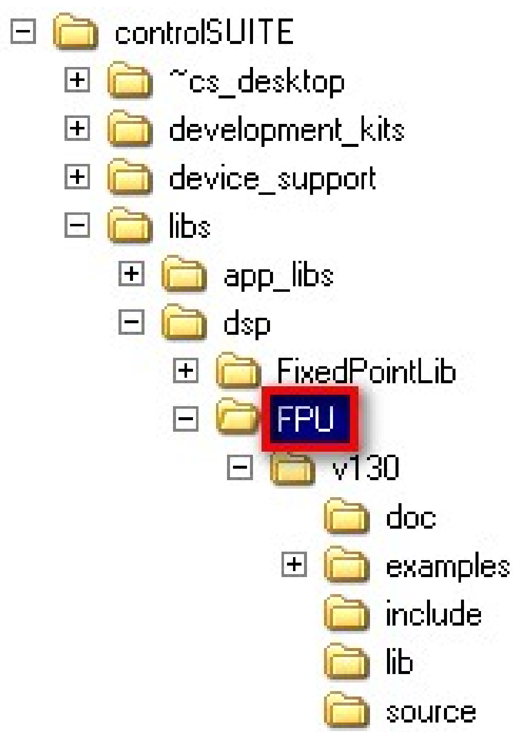

Figure 2.1: Directory Structure of the FPU Library

Folder	Description
<base>	Base install directory. By default this is C:/TI/controlSUITE/libs/dsp/FPU/VERSION For the rest of this document <base> will be omitted from the directory names
<base>/cmd	Linker command files used in the examples
<base>/doc	Documentation for the current revision of the library including revision history
<base>/examples	Examples that illustrate the library functions. At the time of writing these examples were built for the F2833x device using CCS4 but they can be imported into CCS5
<base>/include	Header files for the FPU library. These include function prototypes and structure definitions
<base>/lib	Pre-built FPU libraries
<base>/source	Source files and project for the library. Allows the user to reconfigure, modify and re-build the library to suit their particular needs
<*EXAMPLE*>/matlab	Matlab code for reference of debugging example project or lib source code. FFT results in example projects can be compared by the result in matlab code.

Table 2.1: FPU Library Directory Structure Description

2.1 Build Options used to build the library

The current version of the library was built with C28x Codegen Tools v6.0.1 with the following options:

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

2.2 Header Files

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

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

2.3 A Note about C functions and IQMath

Most of the functions contained in the C28x FPU library are c-callable assembly. A few functions may be written in C. These C functions are written using the IQMath pre-processor notation. This allows these functions to be easily ported from fixed point to floating-point math. The included IQMath header file, IQmathLib.h, controls whether the code is built for fixed point or floating-point.

You may choose to configure the file IQmathLib.h in the controlSUITE directory to generate floating-point code. i.e. the **MATH_TYPE** in the file is defined as **FLOAT_MATH**.

For more information on the IQMath notation, please refer to C28x IQMath Library - A Virtual Floating Point Engine (SPRC087) which can be downloaded from TI's website.

3 Function Descriptions

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_f32u	void CFFT_f32u(CFFT_F32_STRUCT *);
CFFT_f32_mag	void CFFT_f32_mag(CFFT_F32_STRUCT *);
CFFT_f32s_mag	void CFFT_f32s_mag(CFFT_F32_STRUCT *);
CFFT_f32_phase	void CFFT_f32_phase(CFFT_F32_STRUCT *);
CFFT_f32_sincostable	void CFFT_f32_sincostable(CFFT_F32_STRUCT *);
ICFFT_f32	void ICFFT_f32(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_f32s_mag	void RFFT_f32s_mag(RFFT_F32_STRUCT *);
RFFT_f32_phase	void RFFT_f32_phase(RFFT_F32_STRUCT *);
RFFT_f32_sincostable	void RFFT_f32_sincostable(RFFT_F32_STRUCT *);
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);
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);
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);
	Continued on next page

Table 3.1 – continued from previous page

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);
Math	
__ffsqrtf	inline static float32 __ffsqrtf(float32 x);
Utility	
memcpy_fast	void memcpy_fast(void *, const void *, Uint16);
memset_fast	void memset_fast(void*, int16, Uint16);

Table 3.1: List of Functions

The examples for each was built using **CGT 6.0.1** with the following options:

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

The only exception being the fast sqrt example 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

3.1 Complex Fast Fourier Transform

Description:

This module computes a 32-bit floating-point complex FFT including input bit reversing. This version of the function requires input buffer memory alignment. If you do not wish to align the input buffer, then use the **CFFT_f32u** function.

Header File:

FPU.h

Declaration:

```
VOID CFFT_f32 (CFFT_F32_STRUCT *)
```

Usage:

A pointer to the following structure is passed to the CFFT_f32 function:

```
typedef struct {
  float32    *InPtr;
  float32    *OutPtr;
  float32    *CoefPtr;
  float32    *CurrentInPtr;
  float32    *CurrentOutPtr;
  Uint16     Stages;
  Uint16     FFTSize;
} CFFT_F32_STRUCT;
```

Table 3.2 describes each element

Alignment Requirements:

The input buffer must be aligned to a multiple of the *2*FFTSize*sizeof(float)* i.e. *4*FFTSize*. For example, if the **FFTSize** is 128 you must align the buffer corresponding to **InPtr** to 4*128 = 512. An alignment to a smaller value will not work for the 128-pt complex FFT.

To align the input 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 this code example the buffer is assigned to the CFFTin1Buff section.

```
#define  CFFT_STAGES    7
#define  CFFT_SIZE     (1 << CFFT_STAGES)

//Buffer alignment for the input array,
//CFFT_f32u (optional), CFFT_f32 (required)
//Output of FFT overwrites input if
//CFFT_STAGES is ODD
#pragma DATA_SECTION(CFFTin1Buff,"CFFTdata1");
float   CFFTin1Buff[CFFT_SIZE*2];
```

In the project's linker command file, the **CFFTdata1** section is then aligned to a multiple of the FFTSize as shown below:

```
CFFTdata1        : > RAML4,     PAGE = 1, ALIGN(512)
```

The buffers referenced by **InPtr** and **OutPtr** are used in ping-pong fashion. At the first stage of the FFT InPtr and CurrentInPtr both point to the input buffer and OutPtr and CurrentOutPtr point

<table>
<tr><th>Item</th><th>Description</th><th>Format</th><th>Comment</th></tr>
<tr><td>InPtr</td><td>Input data</td><td>Pointer to 32-bit float array</td><td>Input buffer alignment is required. Refer to the alignment section.</td></tr>
<tr><td>OutPtr</td><td>Output buffer</td><td>Pointer to 32-bit float array</td><td></td></tr>
<tr><td>CoefPtr</td><td>Twiddle factors</td><td>Pointer to 32-bit float array</td><td>Calculate using CFFT_f32_cossintable ( ).</td></tr>
<tr><td>CurrentInPtr</td><td>Output Buffer</td><td>Pointer to 32-bit float array</td><td>Result of CFFT_f32. This buffer can then be used as the input to the magnitude and phase calculations. The output order for FFTSize = N is:
CurrentInPtr[0] = real[0]
CurrentInPtr[1] = imag[0]
CurrentInPtr[2] = real[1]
...
CurrentInPtr[N] = real[N/2]
CurrentInPtr[N+1] = imag[N/2]
...
CurrentInPtr[2N-3] = imag[N-2]
CurrentInPtr[2N-2] = real[N-1]
CurrentInPtr[2N-1] = imag[N-1]</td></tr>
<tr><td>CurrentOutPtr</td><td>Output Buffer</td><td>Pointer to 32-bit float array</td><td>Result of N-1 stage complex FFT.</td></tr>
<tr><td>Stages</td><td>Number of stages</td><td>Uint16</td><td>Stages = log2(FFTSize). Must larger than 3.</td></tr>
<tr><td>FFTSize</td><td>FFT size</td><td>Uint16</td><td>Must be a power of 2 greater than or equal to 8.</td></tr>
</table>

Table 3.2: Elements of the Data Structure

to the same output buffer. After bit reversing the input and computing the stage 1 butterflies the output is stored at the location pointed to be cfft.CurrentOutPtr. The next step is to switch the pointer cfft.CurrentInPtr with cfft.CurrentOutPtr so that the output from the n^{th} stage becomes the input to the $n + 1^{th}$ stage while the previous (n^{th}) stage's input buffer will be used as the output for the $n + 1^{th}$ stage. In this manner the CurrentInPtr and CurrentOutPtr are switched successively for each FFT stage.Therefore, If the number of stages is odd then at the end of all the coputation we get:

currentInPtr=InPtr, currentOutPtr=OutPtr.

If number of stages is even then,

currentInPtr=OutPtr, currentOutPtr=InPtr.

<table>
<tr><th rowspan="2"></th><th rowspan="2">Stage3</th><th rowspan="2">Stage4</th><th rowspan="2">Stage5</th><th rowspan="2">...</th><th colspan="2">Stage N</th></tr>
<tr><th>N = odd</th><th>N = even</th></tr>
<tr><td>InPtr (Buf1)</td><td>CurrentInPtr</td><td>CurrentOutPtr</td><td>CurrentInPtr</td><td>...</td><td>CurrentInPtr</td><td>CurrentOutPtr</td></tr>
<tr><td>OutPtr (Buf2)</td><td>CurrentOutPtr</td><td>CurrentInPtr</td><td>CurrentOutPtr</td><td>...</td><td>CurrentOutPtr</td><td>CurrentInPtr</td></tr>
<tr><td>Result Buf</td><td>Buf1</td><td>Buf2</td><td>Buf1</td><td>...</td><td>Buf1</td><td>Buf2</td></tr>
</table>

Table 3.3: Input and Output Buffer Pointer Allocations

Notes:

1. **If the input buffer is not properly aligned, then the output will be unpredictable.**
2. **If you do not wish to align the input buffer, then you must use the CFFT_f32u function. This version of the function does not have any input buffer alignment requirements. Using CFFT_f32u will, however, result in lower cycle performance.**
3. **All buffers and stack are placed in internal memory (zero-wait states in data space).**

Example:

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

```
#include "FPU.h"
#define  CFFT_STAGES    7
#define  CFFT_SIZE     (1 << CFFT_STAGES)

/* CFFTin1Buff section to 4*FFT_SIZE in the linker file     */
#pragma DATA_SECTION(CFFTin1Buff,"CFFTdata1");
float CFFTin1Buff[CFFT_SIZE*2];
#pragma DATA_SECTION(CFFToutBuff,"CFFTdata3");
float CFFToutBuff[CFFT_SIZE*2];
#pragma DATA_SECTION(CFFTF32Coef,"CFFTdata4");
float CFFTF32Coef[CFFT_SIZE];

CFFT_F32_STRUCT cfft;

main()
{
  cfft.InPtr   = CFFTin1Buff;  /* Input data buffer          */
  cfft.OutPtr  = CFFToutBuff;  /* FFT output buffer          */
  cfft.CoefPtr = CFFTF32Coef;  /* Twiddle factor buffer     */
  cfft.FFTSize = CFFT_SIZE;    /* FFT length                 */
  cfft.Stages  = CFFT_STAGES;  /* FFT Stages                 */
  ... ...
  CFFT_f32_sincostable(&cfft); /* Initialize twiddle buffer */
  CFFT_f32(&cfft);             /* Calculate output           */
}
```

Benchmark Information:

The following table provides benchmark numbers for the function. Note that these include the cycles used in the call/return from the function

FFTSize	C-Callable ASM (Cycle Count)
32	1121
64	2331
128	5029
256	11023
512	24249
1024	53219

Table 3.4: Benchmark Information

3.2 Complex Fast Fourier Transform (Unaligned)

Description:

This module computes a 32-bit floating-point complex FFT including input bit reversing. This version of the function does not have any buffer alignment requirements. If you can align the input buffer, then use the **CFFT_f32** function for improved performance

Header File:

FPU.h

Declaration:

```
VOID CFFT_f32u (CFFT_F32_STRUCT *)
```

Usage:

A pointer to the following structure is passed to the CFFT_f32 function.It is the same structure described in the **CFFT_f32** section:

```
typedef struct {
  float32   *InPtr;
  float32   *OutPtr;
  float32   *CoefPtr;
  float32   *CurrentInPtr;
  float32   *CurrentOutPtr;
  Uint16    Stages;
  Uint16    FFTSize;
} CFFT_F32_STRUCT;
```

Table 3.2 describes each element describes each element with the exception that the **input buffer does not require alignment**.

Alignment Requirements:

None

Notes:

1. **If you can align the input buffer to a *4*FFTSize*, then consider using the CFFT_f32 function which has input buffer alignment requirements, but it is more cycle efficient**
2. **All buffers and stack are placed in internal memory (zero-wait states in data space).**

Example:

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

```
#include "FPU.h"
#define  CFFT_STAGES    7
#define  CFFT_SIZE     (1 << CFFT_STAGES)

float CFFTin1Buff[CFFT_SIZE*2];
float CFFTin2Buff[CFFT_SIZE*2];
float CFFToutBuff[CFFT_SIZE*2];
float CFFTF32Coef[CFFT_SIZE];

CFFT_F32_STRUCT cfft;

main()
{
  cfft.InPtr   = CFFTin1Buff;  /* Input data buffer      */
  cfft.OutPtr  = CFFToutBuff;  /* FFT output buffer      */
  cfft.CoefPtr = CFFTF32Coef;  /* Twiddle factor buffer  */
  cfft.FFTSize = CFFT_SIZE;    /* FFT length             */
  cfft.Stages  = CFFT_STAGES;  /* FFT Stages             */
  ... ...
  CFFT_f32_sincostable(&cfft); /* Initialize twiddle     */
                               /* buffer                 */
  CFFT_f32u(&cfft);            /* Calculate output       */
}
```

Benchmark Information:

The following table provides benchmark numbers for the function. Note that these include the cycles used in the call/return from the function

FFTSize	C-Callable ASM (Cycle Count)
32	1351
64	2785
128	5931
256	12821
512	27839
1024	60393

Table 3.5: Benchmark Information

3.3 Complex Fast Fourier Transform Magnitude

Description:

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.

Header File:

FPU.h

Declaration:

```
VOID CFFT_f32_mag (CFFT_F32_STRUCT *)
```

Usage:

A pointer to the following structure is passed to the CFFT_f32_mag function.It is the same structure described in the **CFFT_f32** section:

```
typedef struct {
  float32   *InPtr;
  float32   *OutPtr;
  float32   *CoefPtr;
  float32   *CurrentInPtr;
  float32   *CurrentOutPtr;
  Uint16    Stages;
  Uint16    FFTSize;
} CFFT_F32_STRUCT;
```

Table 3.2 describes each element

Alignment Requirements:

The Magnitude buffer requires no alignment but the input buffer to the complex FFT routine will need alignment if using the **CFFT_f32()**

Notes:

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

Example:

The following sample code obtains the complex FFT magnitude.

```
#include "FPU.h"
#define  CFFT_STAGES    7
#define  CFFT_SIZE     (1 << CFFT_STAGES)

/* CFFTin1Buff section to 4*FFT_SIZE in the linker file */
#pragma DATA_SECTION(CFFTin1Buff,"CFFTdata1");
float CFFTin1Buff[CFFT_SIZE*2];
#pragma DATA_SECTION(CFFTin2Buff,"CFFTdata2");
float CFFTin2Buff[CFFT_SIZE*2];
#pragma DATA_SECTION(CFFToutBuff,"CFFTdata3");
float CFFToutBuff[CFFT_SIZE*2];
#pragma DATA_SECTION(CFFTF32Coef,"CFFTdata4");
float CFFTF32Coef[CFFT_SIZE];

CFFT_F32_STRUCT cfft;

main()
{
  cfft.InPtr   = CFFTin1Buff;  /* Input data buffer      */
  cfft.OutPtr  = CFFToutBuff;  /* FFT output buffer      */
  cfft.CoefPtr = CFFTF32Coef;  /* Twiddle factor buffer  */
  cfft.FFTSize = CFFT_SIZE;    /* FFT length             */
  cfft.Stages  = CFFT_STAGES;  /* FFT Stages             */
  ... ...
  CFFT_f32_sincostable(&cfft); /* Initialize twiddle     */
                               /* buffer                 */
  CFFT_f32(&cfft);             /* Calculate output       */
  CFFT_f32_mag(&cfft);         /* Calculate Magnitude    */
}
```

Benchmark Information:

The following table provides benchmark numbers for the function. Note that these include the cycles used in the call/return from the function

FFTSize	**C-Callable ASM (Cycle Count)**	
	Standard Runtime Lib	Fast Runtime Lib
32	2717	1436
64	5405	2844
128	10781	5660
256	21533	11292
512	43037	22556
1024	86045	45084

Table 3.6: Benchmark Information

3.4 Complex Fast Fourier Transform Magnitude (Scaled)

Description:

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_f32_mag matches the magnitude output from the FFT found in common mathematics software and Code Composer Studio FFT graphs.

Header File:

FPU.h

Declaration:

```
VOID CFFT_f32s_mag (CFFT_F32_STRUCT *)
```

Usage:

A pointer to the following structure is passed to the CFFT_f32s_mag function.It is the same structure described in the **CFFT_f32** section:

```
typedef struct {
  float32   *InPtr;
  float32   *OutPtr;
  float32   *CoefPtr;
  float32   *CurrentInPtr;
  float32   *CurrentOutPtr;
  Uint16    Stages;
  Uint16    FFTSize;
} CFFT_F32_STRUCT;
```

Table 3.2 describes each element

Alignment Requirements:

The Magnitude buffer requires no alignment but the input buffer to the complex FFT routine will need alignment if using the **CFFT_f32()**

Notes:

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

Example:

The following sample code obtains the scaled FFT magnitude.

```
#include "FPU.h"
#define  CFFT_STAGES    7
#define  CFFT_SIZE     (1 << CFFT_STAGES)

/* CFFTin1Buff section to 4*FFT_SIZE in the linker file */
#pragma DATA_SECTION(CFFTin1Buff,"CFFTdata1");
float CFFTin1Buff[CFFT_SIZE*2];
#pragma DATA_SECTION(CFFTin2Buff,"CFFTdata2");
float CFFTin2Buff[CFFT_SIZE*2];
#pragma DATA_SECTION(CFFToutBuff,"CFFTdata3");
float CFFToutBuff[CFFT_SIZE*2];
#pragma DATA_SECTION(CFFTF32Coef,"CFFTdata4");
float CFFTF32Coef[CFFT_SIZE];

CFFT_F32_STRUCT cfft;

main()
{
  cfft.InPtr   = CFFTin1Buff; /* Input data buffer      */
  cfft.OutPtr  = CFFToutBuff; /* FFT output buffer      */
  cfft.CoefPtr = CFFTF32Coef; /* Twiddle factor buffer  */
  cfft.FFTSize = CFFT_SIZE;   /* FFT length             */
  cfft.Stages  = CFFT_STAGES; /* FFT Stages             */
  ... ...
  CFFT_f32_sincostable(&cfft);/* Initialize twiddle     */
                              /* buffer                 */
  CFFT_f32(&cfft);            /* Calculate output       */
  CFFT_f32s_mag(&cfft);       /* Calculate Magnitude    */
}
```

Benchmark Information:

The following table provides benchmark numbers for the function. Note that these include the cycles used in the call/return from the function

FFTSize	**C-Callable ASM (Cycle Count)**	
	Standard Runtime Lib	Fast Runtime Lib
32	2906	1534
64	5760	3013
128	11462	5964
256	22860	11859
512	45650	23642
1024	91224	47201

Table 3.7: Benchmark Information

3.5 Complex Fast Fourier Transform Phase

Description:
This module computes FFT Phase.

Header File:
FPU.h

Declaration:
```
VOID CFFT_f32_phase (CFFT_F32_STRUCT *)
```

Usage:
A pointer to the following structure is passed to the CFFT_f32_phase function.It is the same structure described in the **CFFT_f32** section:

```
typedef struct {
  float32   *InPtr;
  float32   *OutPtr;
  float32   *CoefPtr;
  float32   *CurrentInPtr;
  float32   *CurrentOutPtr;
  Uint16    Stages;
  Uint16    FFTSize;
} CFFT_F32_STRUCT;
```

Table 3.2 describes each element

Alignment Requirements:
The Phase buffer requires no alignment but the input buffer to the complex FFT routine will need alignment if using the **CFFT_f32()**

Notes:

1. **All buffers and stack are placed in internal memory (zero-wait states in data space).**
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 run-time library rts2800_fpu32.lib.**

Example:

The following sample code obtains the Complex FFT phase

```
#include "FPU.h"
#define  CFFT_STAGES     7
#define  CFFT_SIZE      (1 << CFFT_STAGES)

/* CFFTin1Buff section to 4*FFT_SIZE in the linker file */
#pragma DATA_SECTION(CFFTin1Buff,"CFFTdata1");
float CFFTin1Buff[CFFT_SIZE*2];
#pragma DATA_SECTION(CFFTin2Buff,"CFFTdata2");
float CFFTin2Buff[CFFT_SIZE*2];
#pragma DATA_SECTION(CFFToutBuff,"CFFTdata3");
float CFFToutBuff[CFFT_SIZE*2];
#pragma DATA_SECTION(CFFTF32Coef,"CFFTdata4");
float CFFTF32Coef[CFFT_SIZE];

CFFT_F32_STRUCT cfft;

main()
{
  cfft.InPtr = CFFTin1Buff;   /* Input data buffer       */
  cfft.OutPtr = CFFToutBuff;  /* FFT output buffer       */
  cfft.CoefPtr = CFFTF32Coef; /* Twiddle factor buffer   */
  cfft.FFTSize = CFFT_SIZE;   /* FFT length              */
  cfft.Stages = CFFT_STAGES;  /* FFT Stages              */
  ... ...
  CFFT_f32_sincostable(&cfft);/* Initialize twiddle     */
                              /* buffer                  */
  CFFT_f32(&cfft);            /* Calculate output        */
  CFFT_f32_mag(&cfft);        /* Calculate Magnitude     */
  cfft.CurrentOutPtr=CFFTin2Buff;/* Change output buffer*/
  CFFT_f32_phase(&cfft);         /* Calculate phase     */
}
```

Benchmark Information:

The following table provides benchmark numbers for the function. Note that these include the cycles used in the call/return from the function

FFTSize	C-Callable ASM (Cycle Count)	
	Standard Runtime Lib	Fast Runtime Lib
32	29778	2141
64	63279	4253
128	110368	8477
256	242669	16925
512	485624	33821
1024	1002380	67613

Table 3.8: Benchmark Information

3.6 Complex Fast Fourier Transform Twiddle Factors

Description:
This module generates the twiddle factors used by the complex FFT.

Header File:
FPU.h

Declaration:
```
VOID CFFT_f32_sincostable (CFFT_F32_STRUCT *)
```

Usage:
A pointer to the following structure is passed to the CFFT_f32_sincostable function.It is the same structure described in the **CFFT_f32** section:

```
typedef struct {
  float32   *InPtr;
  float32   *OutPtr;
  float32   *CoefPtr;
  float32   *CurrentInPtr;
  float32   *CurrentOutPtr;
  Uint16    Stages;
  Uint16    FFTSize;
} CFFT_F32_STRUCT;
```

Table 3.2 describes each element

Alignment Requirements:
None

Example:

The following sample code obtains the scaled FFT magnitude.

```
#include "FPU.h"
#define  CFFT_STAGES    7
#define  CFFT_SIZE     (1 << CFFT_STAGES)

/* CFFTin1Buff section to 4*FFT_SIZE in the linker file     */
#pragma DATA_SECTION(CFFTin1Buff,"CFFTdata1");
float CFFTin1Buff[CFFT_SIZE*2];
#pragma DATA_SECTION(CFFToutBuff,"CFFTdata3");
float CFFToutBuff[CFFT_SIZE*2];
#pragma DATA_SECTION(CFFTF32Coef,"CFFTdata4");
float CFFTF32Coef[CFFT_SIZE];

CFFT_F32_STRUCT cfft;

main()
{
  cfft.InPtr   = CFFTin1Buff;  /* Input data buffer         */
  cfft.OutPtr  = CFFToutBuff;  /* FFT output buffer         */
  cfft.CoefPtr = CFFTF32Coef;  /* Twiddle factor buffer     */
  cfft.FFTSize = CFFT_SIZE;    /* FFT length                */
  cfft.Stages  = CFFT_STAGES;  /* FFT Stages                */
  CFFT_f32_sincostable(&cfft); /* Initialize twiddle buffer */
  CFFT_f32(&cfft);             /* Calculate output          */
}
```

Benchmark Information:

The CFFT_f32_sincostable function is written in C and not optimized.

3.7 Inverse Complex Fast Fourier Transform

Description:

This module computes a 32-bit floating-point Inverse complex FFT . This version of the function requires input buffer memory alignment.

Header File:

FPU.h

Declaration:

```
VOID ICFFT_f32 (CFFT_F32_STRUCT *)
```

Usage:

A pointer to the following structure is passed to the CFFT_f32 function:

```
typedef struct {
  float32   *InPtr;
  float32   *OutPtr;
  float32   *CoefPtr;
  float32   *CurrentInPtr;
  float32   *CurrentOutPtr;
  Uint16    Stages;
  Uint16    FFTSize;
} CFFT_F32_STRUCT;
```

Table 3.2 describes each element

Alignment Requirements:

The input buffer must be aligned to a multiple of the *2*FFTSize*sizeof(float)* i.e. *4*FFTSize*. For example, if the **FFTSize** is 256 you must align the buffer corresponding to **InPtr** to 4*256 = 1024. A smaller alignment will not work for a 256 IFFT.

To align the input 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 this code example the buffer is assigned to the INBUF section.

```
#define  CFFT_STAGES    8
#define  CFFT_SIZE     (1 << CFFT_STAGES)

// FFT input data buffer, alignment require
// Output of ICFFT overwrites input if
// CFFT_STAGES is ODD
#pragma DATA_SECTION(CFFTin1Buff,"CFFTdata1")
float32 CFFTin1Buff[CFFT_SIZE*2];
```

In the project's linker command file, the **INBUF** section is then aligned to a multiple of the FFTSize as shown below:

```
CFFTdata1        : > RAML4,      PAGE = 1, ALIGN(1024)
```

The buffers referenced by **InPtr** and **OutPtr** are used in ping-pong fashion. At the first stage of the IFFT InPtr and CurrentInPtr both point to the input buffer and OutPtr and CurrentOutPtr point to the same output buffer. After bit reversing the input and computing the stage 1 butterflies the output is stored at the location pointed to be cfft.CurrentOutPtr. The next step is to switch the pointer cfft.CurrentInPtr with cfft.CurrentOutPtr so that the output from the n^{th} stage becomes the input to the $n+1^{th}$ stage while the previous (n^{th}) stage's input buffer will be used as the output for the $n+1^{th}$ stage. In this manner the CurrentInPtr and CurrentOutPtr are switched successively for each IFFT stage.Therefore, If the number of stages is odd then at the end of all the coputation we get:

currentInPtr=InPtr, currentOutPtr=OutPtr.

If number of stages is even then,

currentInPtr=OutPtr, currentOutPtr=InPtr.

	Stage3	**Stage4**	**Stage5**	**...**	**Stage N**	
					N = odd	**N = even**
InPtr (Buf1)	CurrentInPtr	CurrentOutPtr	CurrentInPtr	...	CurrentInPtr	CurrentOutPtr
OutPtr (Buf2)	CurrentOutPtr	CurrentInPtr	CurrentOutPtr	...	CurrentOutPtr	CurrentInPtr
Result Buf	Buf1	Buf2	Buf1	...	Buf1	Buf2

Table 3.9: Input and Output Buffer Pointer Allocations

Notes:

1. **If the input buffer is not properly aligned, then the output will be unpredictable.**
2. **All buffers and stack are placed in internal memory (zero-wait states in data space).**

Example:

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

```
#include "FPU.h"
#define  CFFT_STAGES    3
#define  CFFT_SIZE     (1 << CFFT_STAGES)

/* CFFTin1Buff section to 4*FFT_SIZE in the linker file     */
#pragma DATA_SECTION(CFFTin1Buff,"CFFTdata1");
float CFFTin1Buff[CFFT_SIZE*2];
#pragma DATA_SECTION(CFFToutBuff,"CFFTdata2");
float CFFTin2Buff[CFFT_SIZE*2];
#pragma DATA_SECTION(CFFTF32Coef,"CFFTdata4");
float CFFTF32Coef[CFFT_SIZE];

CFFT_F32_STRUCT cfft;

main()
{
  cfft.InPtr   = CFFTin1Buff; /* Input data buffer          */
  cfft.OutPtr  = CFFToutBuff; /* FFT output buffer          */
  cfft.CoefPtr = CFFTF32Coef; /* Twiddle factor buffer      */
  cfft.FFTSize = CFFT_SIZE;   /* FFT length                 */
  cfft.Stages  = CFFT_STAGES; /* FFT Stages                 */
  ... ...
  CFFT_f32_sincostable(&cfft); /* Initialize twiddle buffer */
  CFFT_f32(&cfft);             /* Calculate output          */
  ... ...
  ICFFT_f32(&cfft);            /* Calculate Inverse FFT     */
}
```

Benchmark Information:

The following table provides benchmark numbers for the function. Note that these include the cycles used in the call/return from the function

FFTSize	C-Callable ASM (Cycle Count)
32	1370
64	2803
128	5948
256	12837
512	27854
1024	60411

Table 3.10: Benchmark Information

3.8 Real Fast Fourier Transform

Description:

This module computes a 32-bit floating-point real FFT including input bit reversing. This version of the function requires input buffer memory alignment. If you do not wish to align the input buffer, then use the **RFFT_f32u** function.

Header File:

FPU.h

Declaration:

```
VOID RFFT_f32 (RFFT_F32_STRUCT *)
```

Usage:

A pointer to the following structure is passed to the RFFT_f32 function:

```
typedef struct {
  float32    *InBuf;
  float32    *OutBuf;
  float32    *CosSinBuf;
  float32    *MagBuf;
  float32    *PhaseBuf;
  Uint16     FFTSize;
  Uint16     FFTStages;
} RFFT_F32_STRUCT;
```

Table 3.11 describes each element

Alignment Requirements:

The input buffer must be aligned to a multiple of the *2*FFTSize*. For example, if the FFTSize is 256 you must align the buffer corresponding to **InBuf** to 2*256 = 512. A smaller alignment will not work for a 256 RFFT.

To align the input 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 this code example the buffer is assigned to the **INBUF** section.

```
#define  RFFT_STAGES    8
#define  RFFT_SIZE     (1 << RFFT_STAGES)

//Buffer alignment for the input array,
//RFFT_f32u (optional), RFFT_f32 (required)
//Output of FFT overwrites input if
//RFFT_STAGES is ODD
#pragma DATA_SECTION(RFFTin1Buff,"RFFTdata1");
float32 RFFTin1Buff[RFFT_SIZE];
```

In the project's linker command file, the **RFFTdata1** section is then aligned to a multiple of the **FFTSize** as shown below:

```
RFFTdata1        : > RAML4,     PAGE = 1, ALIGN(512)
```

Item	Description	Format	Comment
InBuf	Input data	Pointer to 32-bit float array	Input buffer alignment is required. Refer to the alignment section.

OutBuf	Output buffer	Pointer to 32-bit float array	Result of RFFT_f32. This buffer can then be used as the input to the magnitude and phase calculations. The output order for FFTSize = N is:
			`OutBuf[0] = real[0]`
			`OutBuf[1] = real[1]`
			`OutBuf[2] = real[2]`
			`...`
			`OutBuf[N/2] = real[N/2]`
			`OutBuf[N/2+1] = imag[N/2-1]`
			`...`
			`OutBuf[N-3] = imag[3]`
			`OutBuf[N-2] = imag[2]`
			`OutBuf[N-1] = imag[1]`

CosSinBuf	Twiddle factors	Pointer to 32-bit float array	Calculate using **RFFT_f32_sincostable( )**.
FFTSize	FFT size	Uint16	Must be a power of 2 greater than or equal to 32.
FFTStages	Number of stages	Uint16	Stages = log2(FFTSize)
MagBuf	Magnitude buffer	Pointer to 32-bit float array	Output from the magnitude calculation if the magnitude functions is called. **FFTSize/2 in length**.
PhaseBuf	Phase buffer	Pointer to 32-bit float array	Output from the phase calculation if the phase function is called. **FFTSize/2 in length**.

Table 3.11: Elements of the Data Structure

Notes:

1. **If the input buffer is not properly aligned, then the output will be unpredictable.**
2. **If you do not wish to align the input buffer, then you must use the RFFT_f32u function. This version of the function does not have any input buffer alignment requirements. Using RFFT_f32u will, however, result in a lower cycle performance.**
3. **All buffers and stack are placed in internal memory (zero-wait states in data space)**

Example:

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

```
#include "FPU.h"
#define  RFFT_STAGES    8
#define  RFFT_SIZE     (1 << RFFT_STAGES)

/* RFFTin1Buff section to 2*FFT_SIZE in the linker file         */
#pragma DATA_SECTION(RFFTin1Buff,"RFFTdata1");
float32 RFFTin1Buff[RFFT_SIZE];
#pragma DATA_SECTION(RFFToutBuff,"RFFTdata2");
float32 RFFToutBuff[RFFT_SIZE];
#pragma DATA_SECTION(RFFTmagBuff,"RFFTdata3");
float32 RFFTmagBuff[RFFT_SIZE/2+1];
#pragma DATA_SECTION(RFFTF32Coef,"RFFTdata4");
float32 RFFTF32Coef[RFFT_SIZE];

RFFT_F32_STRUCT rfft;

main()
{
    rfft.FFTSize   = RFFT_SIZE;
    rfft.FFTStages = RFFT_STAGES;
    rfft.InBuf     = &RFFTin1Buff[0];  /* Input buffer           */
    rfft.OutBuf    = &RFFToutBuff[0];  /* Output buffer          */
    rfft.CosSinBuf = &RFFTF32Coef[0];  /* Twiddle factor buffer */
    rfft.MagBuf    = &RFFTmagBuff[0];  /* Magnitude buffer       */

    RFFT_f32_sincostable(&rfft);    /* Calculate twiddle factor */
    RFFT_f32(&rfft);                /* Calculate output         */
}
```

Benchmark Information:

The following table provides benchmark numbers for the function. Note that these include the cycles used in the call/return from the function

FFTSize	C-Callable ASM (Cycle Count)
32	611
64	1277
128	2775
256	6145
512	13675
1024	30357
2048	67007

Table 3.12: Benchmark Information

3.9 Real Fast Fourier Transform (Unaligned)

Description:
This module computes a 32-bit floating-point real FFT including input bit reversing. This version of the function does not have any buffer alignment requirements. If you can align the input buffer, then use the **RFFT_f32** function for improved performance.

Header File:
FPU.h

Declaration:
```
VOID RFFT_f32u (RFFT_F32_STRUCT *)
```

Usage:
A pointer to the following structure is passed to the RFFT_f32u function. It is the same structure described in the **RFFT_f32** section:

```
typedef struct {
  float32   *InBuf;
  float32   *OutBuf;
  float32   *CosSinBuf;
  float32   *MagBuf;
  float32   *PhaseBuf;
  Uint16    FFTSize;
  Uint16    FFTStages;
} RFFT_F32_STRUCT;
```

Table 3.11 describes each element with the exception that the **input buffer does not require alignment**.

Alignment Requirements:
None

Notes:

1. **If you can align the input buffer to a *2*FFTSize*, then consider using the RFFT_f32 function which has input buffer alignment requirements, but it is more cycle efficient**
2. **All buffers and stack are placed in internal memory (zero-wait states in data space).**

Example:

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

```
#include "FPU.h"
#define  RFFT_STAGES    8
#define  RFFT_SIZE     (1 << RFFT_STAGES)

/* RFFTin1Buff section to 2*FFT_SIZE in the linker file         */
#pragma DATA_SECTION(RFFTin1Buff,"RFFTdata1");
float32 RFFTin1Buff[RFFT_SIZE];
#pragma DATA_SECTION(RFFToutBuff,"RFFTdata2");
float32 RFFToutBuff[RFFT_SIZE];
#pragma DATA_SECTION(RFFTmagBuff,"RFFTdata3");
float32 RFFTmagBuff[RFFT_SIZE/2+1];
#pragma DATA_SECTION(RFFTF32Coef,"RFFTdata4");
float32 RFFTF32Coef[RFFT_SIZE];

RFFT_F32_STRUCT fft;

main()
{
    rfft.FFTSize   = RFFT_SIZE;
    rfft.FFTStages = RFFT_STAGES;
    rfft.InBuf     = &RFFTin1Buff[0];  /* Input buffer           */
    rfft.OutBuf    = &RFFToutBuff[0];  /* Output buffer          */
    rfft.CosSinBuf = &RFFTF32Coef[0];  /* Twiddle factor buffer */
    rfft.MagBuf    = &RFFTmagBuff[0];  /* Magnitude buffer       */

    RFFT_f32_sincostable(&rfft);    /* Calculate twiddle factor  */
    RFFT_f32u(&rfft);               /* Calculate output          */
}
```

Benchmark Information:

The following table provides benchmark numbers for the function. Note that these include the cycles used in the call/return from the function

FFTSize	C-Callable ASM (Cycle Count)
32	667
64	1389
128	2999
256	6593
512	14571
1024	32149
2048	70591

Table 3.13: Benchmark Information

3.10 Real Fast Fourier Transform with ADC Input

Description:

This module computes a 32-bit floating-point real FFT with 12-bit ADC input including input bit reversing. This version of the function requires input buffer memory alignment. If you do not wish to align the input buffer, then use the **RFFT_adc_f32u** function.

Header File:

FPU.h

Declaration:

```
VOID RFFT_adc_f32 (RFFT_ADC_F32_STRUCT *)
```

Usage:

A pointer to the following structure is passed to the RFFT_adc_f32 function:

```
typedef struct {
  Uint16    *InBuf;
  void      *Tail;
} RFFT_ADC_F32_STRUCT;

typedef struct {
  float32   *InBuf;
  float32   *OutBuf;
  float32   *CosSinBuf;
  float32   *MagBuf;
  float32   *PhaseBuf;
  Uint16    FFTSize;
  Uint16    FFTStages;
} RFFT_F32_STRUCT;
```

Table 3.14 describes each element of the structure RFFT_ADC_F32_STRUCT and table 3.15 describes the elements of RFFT_F32_STRUCT, but note that its **InBuf** pointer is not used

Item	Description	Format	Comment
InBuf	Input data	Pointer to 16-bit Uint16 array	Input buffer alignment is required. Refer to the alignment section.
Tail	Input structure	Null pointer to RFFT_F32_STRUCT	Null pointer is passed to OutBuf of RFFT_F32_STRUCT.

Table 3.14: Elements of the Data Structure RFFT_ADC_F32_STRUCT

Alignment Requirements:

The input buffer must be aligned to a multiple of the *2*FFTSize*. For example, if the FFTSize is 512 you must align the buffer corresponding to **InBuf** to 2*512 = 1024. A smaller alignment will not work.

To align the input 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 this code example the buffer is assigned to the **INBUF** section.

Item	Description	Format	Comment
InBuf	Input data	Pointer to 32-bit float array	Not Used.

OutBuf	Output buffer	Pointer to 32-bit float array	Result of RFFT_adc_f32. This buffer is then used as the input to the magnitude and phase calculations. The output order for FFTSize = N is:
			`OutBuf[0] = real[0]`
			`OutBuf[1] = real[1]`
			`OutBuf[2] = real[2]`
			`...`
			`OutBuf[N/2] = real[N/2]`
			`OutBuf[N/2+1] = imag[N/2-1]`
			`...`
			`OutBuf[N-3] = imag[3]`
			`OutBuf[N-2] = imag[2]`
			`OutBuf[N-1] = imag[1]`

CosSinBuf	Twiddle factors	Pointer to 32-bit float array	Calculate using **RFFT_f32_sincostable( )**.
FFTSize	FFT size	Uint16	Must be a power of 2 greater than or equal to 32.
FFTStages	Number of stages	Uint16	Stages = log2(FFTSize)
MagBuf	Magnitude buffer	Pointer to 32-bit float array	Output from the magnitude calculation if the magnitude functions is called. FFTSize/2 in length.
PhaseBuf	Phase buffer	Pointer to 32-bit float array	Output from the phase calculation if the phase function is called. FFTSize/2 in length.

Table 3.15: Elements of the Data Structure RFFT_F32_STRUCT

```
#define  RFFT_STAGES     9
#define  RFFT_SIZE      (1 << RFFT_STAGES)

//Buffer alignment for the input array,
//RFFT_adc_f32u (optional) RFFT_adc_f32 (required)
//Output of FFT overwrites input if
//RFFT_STAGES is ODD
#pragma DATA_SECTION(RFFTin1Buff,"RFFTdata1");
float32 RFFTin1Buff[RFFT_SIZE];
```

In the project's linker command file, the **RFFTdata1** section is then aligned to a multiple of the **FFTSize** as shown below:

```
RFFTdata1     : > RAML4,     PAGE = 1, ALIGN(1024)
```

Notes:

1. **If the input buffer is not properly aligned, then the output will be unpredictable.**
2. **If you do not wish to align the input buffer, then you must use the RFFT_adc_f32u function which does not have any input buffer alignment requirements. Using RFFT_adc_f32u will, however, result in a lower cycle performance.**
3. **All buffers and stack are placed in internal memory (zero-wait states in data space).**

Example:

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

```
#include "FPU.h"
#define  RFFT_STAGES    9
#define  RFFT_SIZE     (1 << RFFT_STAGES)

RFFT_ADC_F32_STRUCT rfft_adc;
RFFT_F32_STRUCT rfft;

/* RFFTin1Buff section to 2*FFT_SIZE in the linker file              */
#pragma DATA_SECTION(RFFTin1Buff,"RFFTdata1");
float32 RFFTin1Buff[RFFT_SIZE];
#pragma DATA_SECTION(RFFToutBuff,"RFFTdata2");
float32 RFFToutBuff[RFFT_SIZE];
#pragma DATA_SECTION(RFFTmagBuff,"RFFTdata3");
float32 RFFTmagBuff[RFFT_SIZE/2+1];
#pragma DATA_SECTION(RFFTF32Coef,"RFFTdata4");
float32 RFFTF32Coef[RFFT_SIZE];

main()
{
  rfft_adc.Tail  = &rfft.OutBuf;    /* Tail is passed to OutBuf      */
  rfft.FFTSize   = RFFT_SIZE;       /* FFT size                      */
  rfft.FFTStages = RFFT_STAGES;     /* FFT stages                    */
  rfft_adc.InBuf = &RFFTin1Buff[0]; /* Input buffer 12-bit ADC input */
  rfft.OutBuf    = &RFFToutBuff[0]; /* Output buffer                 */
  rfft.CosSinBuf = &RFFTF32Coef[0]; /* Twiddle factor                */
  rfft.MagBuf   = &RFFTmagBuff[0];  /* Magnitude output buffer       */
  ... ...
  RFFT_f32_sincostable(&rfft)       /* Initialize twiddle buffer     */
  RFFT_adc_f32(&rfft);              /* Calculate output              */
}
```

Benchmark Information:

The following table provides benchmark numbers for the function:

FFTSize	C-Callable ASM (Cycle Count)
32	628
64	1290
128	2764
256	6054
512	13360
1024	29466

Table 3.16: Benchmark Information

3.11 Real Fast Fourier Transform with ADC Input (Unaligned)

Description:

This module computes a 32-bit floating-point real FFT with 12-bit ADC input including input bit reversing. This version of the function does not have any buffer alignment requirements. If you can align the input buffer, then use the **RFFT_adc_f32** function for improved performance.

Header File:

FPU.h

Declaration:

```
VOID RFFT_adc_f32u (RFFT_F32_STRUCT *)
```

Usage:

A pointer to the following structure is passed to the RFFT_adc_f32u function:

```
typedef struct {
  Uint16    *InBuf;
  void      *Tail;
} RFFT_ADC_F32_STRUCT;

typedef struct {
  float32   *InBuf;
  float32   *OutBuf;
  float32   *CosSinBuf;
  float32   *MagBuf;
  float32   *PhaseBuf;
  Uint16    FFTSize;
  Uint16    FFTStages;
} RFFT_F32_STRUCT;
```

Table 3.14 describes each element of the structure RFFT_ADC_F32_STRUCT and table 3.15 describes the elements of RFFT_F32_STRUCT, but note that its **InBuf** pointer is not used

Alignment Requirements:

None

Notes:

1. **If you can align the input buffer to a 2*FFTSize, then consider using the RFFT_adc_f32 function. This version of the function has input buffer alignment requirements, but it is more cycle efficient**
2. **All buffers and stack are placed in internal memory (zero-wait states in data space).**

Example:

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

```
#include "FPU.h"
#define  RFFT_STAGES    9
#define  RFFT_SIZE     (1 << RFFT_STAGES)

RFFT_ADC_F32_STRUCT rfft_adc;
RFFT_F32_STRUCT rfft;

/* RFFTin1Buff section to 2*FFT_SIZE in the linker file              */
#pragma DATA_SECTION(RFFTin1Buff,"RFFTdata1");
float32 RFFTin1Buff[RFFT_SIZE];
#pragma DATA_SECTION(RFFToutBuff,"RFFTdata2");
float32 RFFToutBuff[RFFT_SIZE];
#pragma DATA_SECTION(RFFTmagBuff,"RFFTdata3");
float32 RFFTmagBuff[RFFT_SIZE/2+1];
#pragma DATA_SECTION(RFFTF32Coef,"RFFTdata4");
float32 RFFTF32Coef[RFFT_SIZE];

main()
{
  rfft_adc.Tail  = &rfft.OutBuf;    /* Tail is passed to OutBuf      */
  rfft.FFTSize   = RFFT_SIZE;       /* FFT size                      */
  rfft.FFTStages = RFFT_STAGES;     /* FFT stages                    */
  rfft_adc.InBuf = &RFFTin1Buff[0]; /* Input buffer 12-bit ADC input */
  rfft.OutBuf    = &RFFToutBuff[0]; /* Output buffer                 */
  rfft.CosSinBuf = &RFFTF32Coef[0]; /* Twiddle factor                */
  rfft.MagBuf   = &RFFTmagBuff[0];  /* Magnitude output buffer       */
  ... ...
  RFFT_f32_sincostable(&rfft)       /* Initialize twiddle buffer     */
  RFFT_adc_f32u(&rfft);             /* Calculate output              */
}
```

Benchmark Information:

The following table provides benchmark numbers for the function:

FFTSize	C-Callable ASM (Cycle Count)
32	698
64	1444
128	3102
256	6792
512	14962

Table 3.17: Benchmark Information

3.12 Real Fast Fourier Transform Magnitude

Description:

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.

Header File:

FPU.h

Declaration:

```
VOID RFFT_f32_mag (RFFT_F32_STRUCT *)
```

Usage:

A pointer to the following structure is passed to the RFFT_f32_mag function. It is the same structure described in the **RFFT_f32** section:

```
typedef struct {
  float32   *InBuf;
  float32   *OutBuf;
  float32   *CosSinBuf;
  float32   *MagBuf;
  float32   *PhaseBuf;
  Uint16    FFTSize;
  Uint16    FFTStages;
} RFFT_F32_STRUCT;
```

Table 3.11 describes each element.

Alignment Requirements:

None

Notes:

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

Example:

The following sample code obtains the FFT magnitude.

```
include "FPU.h"
#define  RFFT_STAGES    8
#define  RFFT_SIZE     (1 << RFFT_STAGES)

/* RFFTin1Buff section to 2*FFT_SIZE in the linker file         */
#pragma DATA_SECTION(RFFTin1Buff,"RFFTdata1");
float32 RFFTin1Buff[RFFT_SIZE];
#pragma DATA_SECTION(RFFToutBuff,"RFFTdata2");
float32 RFFToutBuff[RFFT_SIZE];
#pragma DATA_SECTION(RFFTmagBuff,"RFFTdata3");
float32 RFFTmagBuff[RFFT_SIZE/2+1];
#pragma DATA_SECTION(RFFTF32Coef,"RFFTdata4");
float32 RFFTF32Coef[RFFT_SIZE];

RFFT_F32_STRUCT rfft;

main()
{
    rfft.FFTSize   = RFFT_SIZE;
    rfft.FFTStages = RFFT_STAGES;
    rfft.InBuf     = &RFFTin1Buff[0];  /* Input buffer          */
    rfft.OutBuf    = &RFFToutBuff[0];  /* Output buffer         */
    rfft.CosSinBuf = &RFFTF32Coef[0];  /* Twiddle factor buffer */
    rfft.MagBuf    = &RFFTmagBuff[0];  /* Magnitude buffer      */

    RFFT_f32_sincostable(&rfft);   /* Calculate twiddle factor  */
    RFFT_f32(&rfft);               /* Calculate output          */
    RFFT_f32_mag(&rfft)            /* Calculate magnitude       */
}
```

Benchmark Information:

The following table provides benchmark numbers for the function. Note that these include the cycles used in the call/return from the function

FFTSize	C-Callable ASM (Cycle Count)	
	Standard Runtime Lib	Fast Runtime Lib
32	1324	694
64	2654	1382
128	5342	2758
256	10718	5510
512	21470	11014
1024	42974	22022
2048	85982	44038

Table 3.18: Benchmark Information

3.13 Real Fast Fourier Transform Magnitude (Scaled)

Description:

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.

Header File:

FPU.h

Declaration:

```
VOID RFFT_f32s_mag (RFFT_F32_STRUCT *)
```

Usage:

A pointer to the following structure is passed to the RFFT_f32s_mag function. It is the same structure described in the **RFFT_f32** section:

```
typedef struct {
  float32   *InBuf;
  float32   *OutBuf;
  float32   *CosSinBuf;
  float32   *MagBuf;
  float32   *PhaseBuf;
  Uint16    FFTSize;
  Uint16    FFTStages;
} RFFT_F32_STRUCT;
```

Table 3.11 describes each element.

Alignment Requirements:

None

Notes:

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

Example:

The following sample code obtains the FFT magnitude.

```
include "FPU.h"
#define  RFFT_STAGES    8
#define  RFFT_SIZE     (1 << RFFT_STAGES)

/* RFFTin1Buff section to 2*FFT_SIZE in the linker file         */
#pragma DATA_SECTION(RFFTin1Buff,"RFFTdata1");
float32 RFFTin1Buff[RFFT_SIZE];
#pragma DATA_SECTION(RFFToutBuff,"RFFTdata2");
float32 RFFToutBuff[RFFT_SIZE];
#pragma DATA_SECTION(RFFTmagBuff,"RFFTdata3");
float32 RFFTmagBuff[RFFT_SIZE/2+1];
#pragma DATA_SECTION(RFFTF32Coef,"RFFTdata4");
float32 RFFTF32Coef[RFFT_SIZE];

RFFT_F32_STRUCT rfft;

main()
{
    rfft.FFTSize   = RFFT_SIZE;
    rfft.FFTStages = RFFT_STAGES;
    rfft.InBuf     = &RFFTin1Buff[0];  /* Input buffer           */
    rfft.OutBuf    = &RFFToutBuff[0];  /* Output buffer          */
    rfft.CosSinBuf = &RFFTF32Coef[0];  /* Twiddle factor buffer */
    rfft.MagBuf    = &RFFTmagBuff[0];  /* Magnitude buffer       */

    RFFT_f32_sincostable(&rfft);   /* Calculate twiddle factor  */
    RFFT_f32(&rfft);               /* Calculate output          */
    RFFT_f32s_mag(&rfft)           /* Calculate magnitude       */
}
```

Benchmark Information:

The following table provides benchmark numbers for the function. Note that these include the cycles used in the call/return from the function

FFTSize	C-Callable ASM (Cycle Count)	
	Standard Runtime Lib	Fast Runtime Lib
32	1367	737
64	2749	1447
128	5507	2861
256	11017	5683
512	22031	11321
1024	44053	22591
2048	88091	45125

Table 3.19: Benchmark Information

3.14 Real Fast Fourier Transform Phase

Description:
This module computes FFT Phase.

Header File:
FPU.h

Declaration:
```
VOID RFFT_f32_phase (RFFT_F32_STRUCT *)
```

Usage:
A pointer to the following structure is passed to the RFFT_f32_phase function. It is the same structure described in the **RFFT_f32** section:

```
typedef struct {
  float32   *InBuf;
  float32   *OutBuf;
  float32   *CosSinBuf;
  float32   *MagBuf;
  float32   *PhaseBuf;
  Uint16    FFTSize;
  Uint16    FFTStages;
} RFFT_F32_STRUCT;
```

Table 3.11 describes each element.

Alignment Requirements:
None

Notes:

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

Example:

The following sample code obtains the FFT phase.

```
include "FPU.h"
#define  RFFT_STAGES    8
#define  RFFT_SIZE     (1 << RFFT_STAGES)

/* RFFTin1Buff section to 2*FFT_SIZE in the linker file         */
#pragma DATA_SECTION(RFFTin1Buff,"RFFTdata1");
float32 RFFTin1Buff[RFFT_SIZE];
#pragma DATA_SECTION(RFFToutBuff,"RFFTdata2");
float32 RFFToutBuff[RFFT_SIZE];
#pragma DATA_SECTION(RFFTmagBuff,"RFFTdata3");
float32 RFFTphaseBuff[RFFT_SIZE/2+1];
#pragma DATA_SECTION(RFFTF32Coef,"RFFTdata4");
float32 RFFTF32Coef[RFFT_SIZE];

main()
{
    rfft.FFTSize   = RFFT_SIZE;
    rfft.FFTStages = RFFT_STAGES;
    rfft.InBuf     = &RFFTin1Buff[0];  /* Input buffer            */
    rfft.OutBuf    = &RFFToutBuff[0];  /* Output buffer           */
    rfft.CosSinBuf = &RFFTF32Coef[0];  /* Twiddle factor buffer */
    rfft.PhaseBuf  = &RFFTphaseBuff[0];/* Phase buffer            */

    RFFT_f32_sincostable(&rfft);   /* Calculate twiddle factor  */
    RFFT_f32(&rfft);               /* Calculate output          */
    RFFT_f32_phase(&rfft)          /* Calculate phase           */
}
```

Benchmark Information:

The following table provides benchmark numbers for the function. Note that these include the cycles used in the call/return from the function

FFTSize	C-Callable ASM (Cycle Count)	
	Standard Runtime Lib	Fast Runtime Lib
32	14909	1105
64	29096	2152
128	59381	4239
256	106114	8406
512	237106	16733
1024	479424	33380
2048	852535	66667

Table 3.20: Benchmark Information

3.15 Real Fast Fourier Transform Twiddle Factors

Description:
This module generates the twiddle factors used by the real FFT.

Header File:
FPU.h

Declaration:
```
VOID RFFT_f32_sincostable (RFFT_F32_STRUCT *)
```

Usage:
A pointer to the following structure is passed to the RFFT_f32_sincostable function. It is the same structure described in the **RFFT_f32** section:

```
typedef struct {
  float32   *InBuf;
  float32   *OutBuf;
  float32   *CosSinBuf;
  float32   *MagBuf;
  float32   *PhaseBuf;
  Uint16    FFTSize;
  Uint16    FFTStages;
} RFFT_F32_STRUCT;
```

Table 3.11 describes each element.

Alignment Requirements:
None

Example:

The following sample code obtains the FFT phase.

```
#include "FPU.h"
#define  RFFT_STAGES    8
#define  RFFT_SIZE     (1 << RFFT_STAGES)

/* RFFTin1Buff section to 2*FFT_SIZE in the linker file         */
#pragma DATA_SECTION(RFFTin1Buff,"RFFTdata1");
float32 RFFTin1Buff[RFFT_SIZE];
#pragma DATA_SECTION(RFFToutBuff,"RFFTdata2");
float32 RFFToutBuff[RFFT_SIZE];
#pragma DATA_SECTION(RFFTmagBuff,"RFFTdata3");
float32 RFFTmagBuff[RFFT_SIZE/2+1];
#pragma DATA_SECTION(RFFTF32Coef,"RFFTdata4");
float32 RFFTF32Coef[RFFT_SIZE];

RFFT_F32_STRUCT rfft;

main()
{
    rfft.FFTSize   = RFFT_SIZE;
    rfft.FFTStages = RFFT_STAGES;
    rfft.InBuf     = &RFFTin1Buff[0];  /* Input buffer           */
    rfft.OutBuf    = &RFFToutBuff[0];  /* Output buffer          */
    rfft.CosSinBuf = &RFFTF32Coef[0];  /* Twiddle factor buffer */
    rfft.MagBuf    = &RFFTmagBuff[0];  /* Magnitude buffer       */

    RFFT_f32_sincostable(&rfft);   /* Calculate twiddle factor  */
    RFFT_f32(&rfft);               /* Calculate output          */
}
```

Benchmark Information:

The RFFT_f32_sincostable function is written in C and not optimized.

3.16 Finite Impulse Response Filter

Description:
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.

Header File:
FPU.h

Declaration:
```
void FIR_FP_calc(FIR_FP_handle)
```

Usage:
A pointer to the following structure is passed to the FIR_f32 function:

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

Table 3.21 describes each element

Item	Description	Format	Comment
coeff_ptr	Pointer to Filter coefficient	Pointer to 32-bit float array	Place the coefficients in a section (e.g. “coeffilt”) aligned to 2x number of coefficients
dbuffer_ptr	Delay buffer ptr	Pointer to 32-bit float array	Place the Delay in a section (e.g. “firldb”) aligned to an even number of words
cbindex	Circular Buffer Index	Uint16	Index to the delay buffer
order	Order of the Filter	Uint16	Order is number of coefficients minus one
input	Latest Input sample	32-bit float	can be assigned to an ADC input
output	Filter Output	32-bit float	
*init	Pointer to Init funtion	n/a	Points to FIR_FP_init
*calc	Pointer to calc funtion	n/a	Points to FIR_FP_calc

Table 3.21: Elements of the Data Structure

Alignment Requirements:

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.

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 this code example the buffer is assigned to the **firldb** section while the coefficients are assigned to the **coefffilt** section.

```
#define  FIR_ORDER    31
#pragma DATA_SECTION(dbuffer, "firldb")
float dbuffer[FIR_ORDER+1];

#pragma DATA_SECTION(coeff, "coefffilt");
float const coeff[FIR_ORDER+1]= FIR_FP_LPF32;
```

In the project's linker command file, the **firldb** section is then aligned to a value greater or equal to the minimum required as shown below:

```
firldb    ALIGN(0x100)      > RAML0   PAGE = 0
coefffilt ALIGN(0x100)      > RAML2   PAGE = 0
```

Notes:

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

Example:

The following sample code obtains the FIR response to a sample input.

```
#include FPU.h
#define  FIR_ORDER         31
#define  SIGNAL_LENGTH    (FIR_ORDER + 1)*2*2

#pragma DATA_SECTION(firFP, "firfilt")
FIR_FP  firFP = FIR_FP_DEFAULTS;

#pragma DATA_SECTION(dbuffer, "firldb")
float dbuffer[FIR_ORDER+1];

#pragma DATA_SECTION(sigIn, "sigIn");
#pragma DATA_SECTION(sigOut, "sigOut");
float sigIn[SIGNAL_LENGTH];
float sigOut[SIGNAL_LENGTH];

#pragma DATA_SECTION(coeff, "coefffilt");
float const coeff[FIR_ORDER+1]= FIR_FP_LPF32;

float   RadStep  = 0.062831853071f;
float   RadStep2 = 2.073451151f;
float Rad      = 0.0f;
float  Rad2     = 0.0f;

float xn,yn;
int count = 0;

void main()
{
    unsigned int i;
    /* FIR Generic Filter Initialisation    */
    firFP.order=FIR_ORDER;
    firFP.dbuffer_ptr=dbuffer;
    firFP.coeff_ptr=(float *)coeff;
    firFP.init(&firFP);

    for(i=0; i < SIGNAL_LENGTH; i++)
    {
      xn=0.5*sin(Rad) + 0.5*sin(Rad2); //Q15
      sigIn[i]=xn;
      firFP.input= xn;
      firFP.calc(&firFP);
      yn = firFP.output;
      sigOut[i]=yn;
      Rad = Rad + RadStep;
      Rad2 = Rad2 + RadStep2;       }
}
```

Benchmark Information:

The following table provides benchmark numbers for the function:

FIR order	C-Callable ASM (Cycle Count)
31	81
63	113
127	177
255	305
511	561

Table 3.22: Benchmark Information

3.17 Absolute Value of a Complex Vector

Description:

This module computes the absolute value of a complex Vector.

$$Y[i] \quad = \quad \sqrt{(}X_{re}[i]^2 + X_{im}[i]^2)$$

Header File:

FPU.h

Declaration:

```
abs_SP_CV(float32 *y, const complex_float *x, const Uint16 N)
```

Usage:

abs_SP_CV(x, y, N);

float32 *y
output array
complex_float *x
input array
Uint16 N
length of x and y arrays

The type “complex_float” is defined as

```
typedef struct{
  float32 dat[2];
}complex_float;
```

Element dat[0] is the real part, dat[1] is the imaginary part

Alignment Requirements:

None

Example:

```
#include "FPU.h"
#define  N    10
float32 y[N];
complex_float x[N];

main()
{
  abs_SP_CV(y, x, N);        // complex absolute value
}
```

Benchmark Information:

Number of Cycles = 28*N+9 cycles (including the call and return)

3.18 Absolute Value of an Even Length Complex Vector

Description:

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

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

Header File:

FPU.h

Declaration:

```
void abs_SP_CV_2(float32 *y, const complex_float *x, const Uint16 N)
```

Usage:

abs_SP_CV_2(x, y, N);

float32 *y
: output array

complex_float *x
: input array

Uint16 N
: length of x and y arrays

The type “complex_float” is defined as

```
typedef struct{
  float32 dat[2];
}complex_float;
```

Element dat[0] is the real part, dat[1] is the imaginary part

Alignment Requirements:

None

Notes:

1. **N must be EVEN**

Example:

```
#include "FPU.h"
#define  N    10
float32 y[N];
complex_float x[N];

main()
{
  abs_SP_CV_2(y, x, N);      // complex absolute value
}
```

Benchmark Information:

Number of Cycles = 18*N+22 cycles (including the call and return)

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

Description:

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

$$\begin{aligned} Y_{re}[i] &= X_{re}[i] + C_{re} \\ Y_{im}[i] &= X_{im}[i] + C_{im} \end{aligned}$$

Header File:

FPU.h

Declaration:

```
void add_SP_CSxCV(complex_float *y, const complex_float *x,
                        const complex_float c, const Uint16 N)
```

Usage:

add_SP_CSxCV(y, w, c, N);

complex_float *y
result complex array

complex_float *x
input complex array

complex_float c
input complex scalar

Uint16 N
length of x and y arrays

The inputs and return value are of type “complex_float” defined as

```
typedef struct{
  float32 dat[2];
}complex_float;
```

Element dat[0] is the real part, dat[1] is the imaginary part

Alignment Requirements:

None

Notes:

1. **N must be at least 2**

Example:

```
#include "FPU.h"
#define  N    4
complex_float w1[N], w2[N], z[N];

main()
{
  add_SP_CSxCV(z, w1, w2[0], N);
}
```

Benchmark Information:

Number of Cycles = 4*N + 18 cycles (including the call and return)

3.20 Addition of Two Complex Vectors

Description:

This module adds two complex vectors.

$$
\begin{aligned}
Y_{re}[i] &= W_{re}[i] + X_{re}[i] \\
Y_{im}[i] &= W_{im}[i] + X_{im}[i]
\end{aligned}
$$

Header File:

FPU.h

Declaration:

```
void add_SP_CVxCV(complex_float *y, const complex_float *w,
                        const complex_float *x, const Uint16 N)
```

Usage:

add_SP_CVxCV(y, w, x, N);

complex_float *y
result complex array
complex_float *w
input complex array 1
complex_float *x
input complex array 2
Uint16 N
length of w, x, and y arrays

The inputs and return value are of type “complex_float” defined as

```
typedef struct{
  float32 dat[2];
}complex_float;
```

Element dat[0] is the real part, dat[1] is the imaginary part.

Alignment Requirements:

None

Notes:

1. **N must be at least 2**

Example:

```
#include "FPU.h"
#define  N    4
complex_float w1[N], w2[N], z[N];

main()
{
  add_SP_CVxCV(z, w1, w2, N);
}
```

Benchmark Information:

Number of Cycles = 6*N + 15 cycles (including the call and return)

3.21 Inverse Absolute Value of a Complex Vector

Description:
This module computes the inverse absolute value of a complex vector

$$Y[i] \quad = \quad \frac{1}{\sqrt(X_{re}[i]^2 + X_{im}[i]^2)}$$

Header File:
FPU.h

Declaration:
```
void iabs_SP_CV(float32 *y, const complex_float *x, const Uint16 N)
```

Usage:
iabs_SP_CV(y, x, N);

float32 *y
output array
complex_float *x
input complex array
Uint16 N
length of x and y arrays

The type “complex_float” is defined as

```
typedef struct{
  float32 dat[2];
}complex_float;
```

Element dat[0] is the real part, dat[1] is the imaginary part.

Alignment Requirements:
None

Notes:
1. **N must be at least 2**

Example:
```
#include "FPU.h"
#define  N    4
float32 y[N];
complex_float x[N];

main()
{
  iabs_SP_CV(y, x, N);     // inverse complex absolute value

}
```

Benchmark Information:
Number of Cycles = 25*N + 13 cycles (including the call and return)

3.22 Inverse Absolute Value of an Even Length Complex Vector

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

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

Header File:
FPU.h

Declaration:
```
void iabs_SP_CV_2(float32 *y, const complex_float *x, const Uint16 N)
```

Usage:
iabs_SP_CV_2(y, x, N);

float32 *y
output array
complex_float *x
input complex array
Uint16 N
length of x and y arrays (must be even)

The type “complex_float” is defined as

```
typedef struct{
  float32 dat[2];
}complex_float;
```

Element dat[0] is the real part, dat[1] is the imaginary part.

Alignment Requirements:
None

Notes:
1. **N must be EVEN**

Example:
```
#include "FPU.h"
#define  N    4
float32 y[N];
complex_float x[N];

main()
{
  iabs_SP_CV_2(y, x, N);    // inverse complex absolute value
}
```

Benchmark Information:
Number of Cycles = 15*N + 22 cycles (including the call and return)

3.23 Index of Maximum Value of an Even Length Real Array

Description:
This module finds the index of maximum value of an even length real array

Header File:
FPU.h

Declaration:
```
Uint16 maxidx_SP_RV_2(float32 *x, Uint16 N)
```

Usage:
index = maxidx_SP_RV_2(x, N);

float32 x
input array
Uint16 N
length of x
Uint16 index
index of maximum value in x

NOTE:

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 occurence (lowest index value)**

Alignment Requirements:
None

Example:
```
#include "FPU.h"
#define  N    10
float32 x[N]={-4.3, 3.2, 0.0, -2.4, 4.2,
               5.9, 6.1, -3.3, 6.1, 2.2};
volatile Uint16 index;

main()
{
    index = 999;
    index = maxidx_SP_RV_2(x, N);
}
```

Benchmark Information:
Number of Cycles = 3*N + 21 cycles (including the call and return)

3.24 Mean of Real and Imaginary Parts of a Complex Vector

Description:

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

$$
\begin{aligned}
Y_{re} &= \frac{\Sigma X_{re}}{N} \\
Y_{im} &= \frac{\Sigma X_{im}}{N}
\end{aligned}
$$

Header File:

FPU.h

Declaration:

```
complex_float mean_SP_CV_2(const complex_float *x, const Uint16 N)
```

Usage:

y = mean_SP_CV_2(x, N);

complex_float *x
input complex array
Uint16 N
length of x array
complex_float y
result

The type “complex_float” is defined as

```
typedef struct{
  float32 dat[2];
}complex_float;
```

Element dat[0] is the real part, dat[1] is the imaginary part.

Alignment Requirements:

None

Notes:

1. **N must be EVEN and a minimum of 4.**

Example:

```
#include "FPU.h"
#define  N    4
complex_float y;
complex_float x[N];

main()
{
    y = mean_SP_CV_2(x, N);
}
```

Benchmark Information:

Number of Cycles = 2*N + 34 cycles (including the call and return)

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

Description:
This module computes the median of a real valued array of floats. The input array is preserved. This function calls median_SP_RV() and memcpy_fast()

Header File:
FPU.h

Declaration:
```
float32 median_noreorder_SP_RV(const float32 *, Uint16)
```

Usage:
z = median_noreorder_SP_CV(x, N);

float32 *x
pointer to array of real input values

Uint16 N
size of x array

float32 z
the median of x[]

The type "complex_float" is defined as

```
typedef struct{
  float32 dat[2];
}complex_float;
```

Element dat[0] is the real part, dat[1] is the imaginary part.

Alignment Requirements:
None

Notes:

1. **This function simply makes a local copy of the input array, and then calls median_SP_CV() using the copy**
2. **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 greated than K, memory corruption will result. Be sure to recompile the library with an appropriate value $K >= N$ before executing this code**

Example:

```
#include "FPU.h"
#define  N    256
float32  y[N];
float32  x[N];
float32  z;

main()
{
  memcpy_fast(x, y, N<<1);         // Start with same data
  z = median_noreorder_SP_RV(x, N);
}
```

Benchmark Information:

This is a C function and is not benchmarked

3.26 Median of a real array of floats

Description:

This module computes the median of a real array of floats. The Input array is NOT preserved. This function calls qsort_SP_CV(), which is an optimized version of the RTS library qsort() for floats.

Header File:

FPU.h

Declaration:

```
float32 median_SP_RV(float32 *, Uint16)
```

Usage:

z = median_SP_RV(x, N);

float32 *x
: input array

Uint16 N
: length of x array

float32 z
: result

Alignment Requirements:

None

Notes:

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().**

Example:

```
#include "FPU.h"
#define  N    256
float32  y[N];
float32  x[N];
float32  z;

main()
{
  memcpy_fast(x, y, N<<1);       // Start with same data
  z = median_SP_RV(x, N);
}
```

Benchmark Information:

This is a C function and is not benchmarked

3.27 Complex Multiply of Two Floating Point Numbers

Description:
This module multiplies two 32-bit floating point complex values

$$\begin{aligned} Y_{re} &= W_{re} * X_{re} - W_{im} * X_{im} \\ Y_{im} &= W_{re} * X_{im} + W_{im} * X_{re} \end{aligned}$$

Header File:
FPU.h

Declaration:
```
complex_float mpy_SP_CSxCS(complex_float w, complex_float x)
```

Usage:
y = mpy_SP_CSxCS(w,x);

complex_float w
input 1
complex_float x
input 2
complex_float y
result

The type “complex_float” is defined as

```
typedef struct{
  float32 dat[2];
}complex_float;
```

Element dat[0] is the real part, dat[1] is the imaginary part.

Alignment Requirements:
None

Example:
```
#include "FPU.h"
// Global variables for complex multiply test
complex_float w,x,y;

main()
{
  y = mpy_SP_CSxCS(w,x);   // complex multiply
}
```

Benchmark Information:
Number of Cycles = 19 cycles (including the call and return)

3.28 Complex Multiply of Two Complex Vectors

Description:

This module performs complex multiplication on two input complex vectors

$$\begin{aligned} Y_{re}[i] &= W_{re}[i] * X_{re}[i] - W_{im}[i] * X_{im}[i] \\ Y_{im}[i] &= W_{re}[i] * X_{im}[i] + W_{im}[i] * X_{re}[i] \end{aligned}$$

Header File:

FPU.h

Declaration:

```
void mpy_SP_CVxCV(complex_float *y, const complex_float *w,
                        const complex_float *x, const Uint16 N)
```

Usage:

mpy_SP_CVxCV(y, w, x, N);

complex_float *y
result complex array

complex_float *w
input complex array 1

complex_float *x
input complex array 2

Uint16 N
length of w, x, and y arrays

The type “complex_float” is defined as

```
typedef struct{
  float32 dat[2];
}complex_float;
```

Element dat[0] is the real part, dat[1] is the imaginary part.

Alignment Requirements:

None

Example:

```
#include "FPU.h"
#define  N    4
complex_float w1[N], w2[N], z[N];

main()
{
  mpy_SP_CVxCV(z, w1, w2, N);
}
```

Benchmark Information:

Number of Cycles = 10*N + 16 cycles (including the call and return)

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

Description:

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

$$\begin{aligned} X^*_{re}[i] &= X_{re}[i] \\ X^*_{im}[i] &= -X_{im}[i] \\ Y_{re}[i] &= W_{re}[i] * X_{re}[i] - W_{im}[i] * X^*_{im}[i] \\ Y_{im}[i] &= W_{re}[i] * X^*_{im}[i] + W_{im}[i] * X_{re}[i] \end{aligned}$$

Header File:

FPU.h

Declaration:

```
void mpy_SP_CVxCVC(complex_float *y, const complex_float *w,
                        const complex_float *x, const Uint16 N)
```

Usage:

mpy_SP_CVxCVC(y, w, x, N);

complex_float *y
: result complex array

complex_float *w
: input complex array 1

complex_float *x
: input complex array 2

Uint16 N
: length of w, x, and y arrays

The type “complex_float” is defined as

```
typedef struct{
  float32 dat[2];
}complex_float;
```

Element dat[0] is the real part, dat[1] is the imaginary part.

Alignment Requirements:

None

Example:

```
#include "FPU.h"
#define  N    4
complex_float w1[N], w2[N], z[N];
main()
{
  mpy_SP_CVxCVC(z, w1, w2, N);
}
```

Benchmark Information:

Number of Cycles = 11*N + 16 cycles (including the call and return)

3.30 Multiplication of a Real scalar and a Real Vector

Description:

This module multiplies a real scalar and a real vector

$$Y[i] \quad = \quad C * X[i]$$

Header File:

FPU.h

Declaration:

```
void mpy_SP_RSxRV_2(float32 *y, const float32 *x,
                        const float32 c, const Uint16 N)
```

Usage:

mpy_SP_RSxRV_2(y, x, c, N);

float32 *y
: result real array

float32 *x
: input real array

float32 c
: input real scalar

Uint16 N
: length of x and y array

Alignment Requirements:

None

Notes:

1. **N must be EVEN and a minimum of 4.**

Example:

```
#include "FPU.h"
#define  N   10
float32 x1[N], y[N];
float32 c;

main()
{
  mpy_SP_RSxRV_2(y, x1, c, N);
}
```

Benchmark Information:

Number of Cycles = 2*N + 15 cycles (including the call and return)

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

Description:
This module multiplies a real scalar with a real vector and another real vector

$$Y[i] \quad = \quad C * W[i] * X[i]$$

Header File:
FPU.h

Declaration:
```
void mpy_SP_RSxRVxRV_2(float32 *y, const float32 *w,
                        const float32 *x, const float32 c, const Uint16 N)
```

Usage:
mpy_SP_RSxRVxRV_2(y, w, x, c, N);

float32 *y
result real array
float32 *w
input real array 1
float32 *x
input real array 2
float32 c
input real scalar
Uint16 N
length of w, x and y arrays

Alignment Requirements:
None

Notes:
1. **N must be EVEN and a minimum of 4.**

Example:
```
#include "FPU.h"
#define  N    4
float32 x1[N], x2[N], y[N];
float32 c;

main()
{
  mpy_SP_RSxRVxRV_2(y, x1, x2, c, N);
}
```

Benchmark Information:
Number of Cycles = 3*N + 22 cycles (including the call and return)

3.32 Multiplication of a Real Vector and a Complex Vector

Description:

This module multiplies a real vector and a complex vector

$$Y_{re}[i] = X[i] * W_{re}[i]$$
$$Y_{im}[i] = X[i] * W_{im}[i]$$

Header File:

FPU.h

Declaration:

```
void mpy_SP_RVxCV(complex_float *y, const complex_float *w,
                          const float32 *x, const Uint16 N)
```

Usage:

mpy_SP_RVxCV(y, x, c, N);

complex_float *y
result complex array
complex_float *w
input complex array
float32 *x
input real array
Uint16 N
length of w, x, and y arrays

The type "complex_float" is defined as

```
typedef struct{
  float32 dat[2];
}complex_float;
```

Element dat[0] is the real part, dat[1] is the imaginary part.

Alignment Requirements:

None

Notes:

1. **N must be at least 2**

Example:

```
#include "FPU.h"
#define  N    4
float32 x1[N];
complex_float w1[N], z[N];

main()
{
    mpy_SP_RVxCV(z, w1, x1, N);
}
```

Benchmark Information:

Number of Cycles = 5*N + 15 cycles (including the call and return)

3.33 Multiplication of a Real Vector and a Real Vector

Description:
This module multiplies two real vectors

$$Y[i] = W[i] * X[i]$$

Header File:
FPU.h

Declaration:
```
void mpy_SP_RVxRV_2(float32 *y, const float32 *w,
                        const float32 *x, const Uint16 N)
```

Usage:
mpy_SP_RVxRV_2(y, w, x, N);

float32 *y
result real array
float32 *w
input real array 1
float32 *x
input real array 2
Uint16 N
length of w, x and y arrays

The type “complex_float” is defined as

```
typedef struct{
  float32 dat[2];
}complex_float;
```

Element dat[0] is the real part, dat[1] is the imaginary part.

Alignment Requirements:
None

Notes:
1. **N must be EVEN and a minimum of 4.**

Example:
```
#include "FPU.h"
#define  N    4
float32 x1[N], x2[N], y[N];
float32 c;

main()
{
  mpy_SP_RVxRV_2(y, x1, x2, N);
}
```

Benchmark Information:
Number of Cycles = 3*N + 17 cycles (including the call and return)

3.34 Sort an Array of Floats

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

Header File:
FPU.h

Declaration:
```
void qsort_SP_RV(void *, Uint16)
```

Usage:
qsort_SP_RV(x, N);

void *x
input array of floats

Uint16 N
size of x array

Alignment Requirements:
None

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

Example:
```
#include "FPU.h"
#define  N    4
float x1[N];

main()
{
  qsort_SP_RV(x1,N);
}
```

Benchmark Information:
This is a C function and is not benchmarked

3.35 Rounding (Unbiased) of a Floating Point Scalar

Description:
This module performs the unbiased rounding of a 32-bit floating point scalar

Header File:
FPU.h

Declaration:
```
float32 rnd_SP_RS(float32 x)
```

Usage:
y = rnd_SP_RS(x);

float32 x
input value

float32 y
result

Alignment Requirements:
None

Notes:

1. **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

Example:
```
#include "FPU.h"
#define  N    6

main()
{
  volatile float32 x[N] = {4.4, -4.4, 4.5,
                           -4.5, 4.6, -4.6};
  volatile float32 y[N];
  Uint16 i;

  for(i=0; i<N; i++)
  {
    y[i] = rnd_SP_RS(x[i]);
  }
}
```

Benchmark Information:
Number of Cycles = 18 cycles (including the call and return)

3.36 Subtraction of a Complex Scalar from a Complex Vector

Description:
This module subtracts a complex scalar from a complex vector

$$\begin{aligned} Y_{re}[i] &= X_{re}[i] - C_{re} \\ Y_{im}[i] &= X_{im}[i] - C_{im} \end{aligned}$$

Header File:
FPU.h

Declaration:
```
void sub_SP_CSxCV(complex_float *y, const complex_float *x,
                        const complex_float c, const Uint16 N)
```

Usage:
sub_SP_CSxCV(y, w, c, N);

complex_float *y
result complex array
complex_float *x
input complex array
complex_float c
input complex scalar
Uint16 N
length of x and y arrays

The type “complex_float” is defined as

```
typedef struct{
  float32 dat[2];
}complex_float;
```

Element dat[0] is the real part, dat[1] is the imaginary part.

Alignment Requirements:
None

Notes:
1. **N must be at least 2**

Example:
```
#include "FPU.h"
#define  N   4
complex_float w1[N], w2[N], z[N];

main()
{
  sub_SP_CSxCV(z, w1, w2[0], N);
}
```

Benchmark Information:
Number of Cycles = 4*N + 18 cycles (including the call and return)

3.37 Subtraction of a Complex Vector and another Complex Vector

Description:
This module subtracts a complex vector from another complex vector

$$
\begin{aligned}
Y_{re}[i] &= W_{re}[i] - X_{re}[i] \\
Y_{im}[i] &= W_{im}[i] - X_{im}[i]
\end{aligned}
$$

Header File:
FPU.h

Declaration:
```
void sub_SP_CVxCV(complex_float *y, const complex_float *w,
                        const complex_float *x, const Uint16 N)
```

Usage:
sub_SP_CVxCV(y, w, x, N);

complex_float *y
result complex array
complex_float *w
input complex array 1
complex_float *x
input complex array 2
Uint16 N
length of w, x and y arrays

The type “complex_float” is defined as

```
typedef struct{
  float32 dat[2];
}complex_float;
```

Element dat[0] is the real part, dat[1] is the imaginary part.

Alignment Requirements:
None

Notes:
1. **N must be at least 2**

Example:
```
#include "FPU.h"
#define  N    4
complex_float w1[N], w2[N], z[N];

main()
{
  sub_SP_CVxCV(z, w1, w2, N);
}
```

Benchmark Information:
Number of Cycles = 6*N + 15 cycles (including the call and return)

3.38 Fast Square Root

Description:

This function is an inline optmized fast square root function using two iterations of the newton raphson method to achieve an accurate result

Header File:

FPU.h

Declaration:

```
inline static float32 __ffsqrtf(float32 x)
```

Usage:

__ffsqrtf(x);

void x

input variable

Alignment Requirements:

None

Notes:

1. **Performance is best with -o2, -mn compiler options (cgtools v6.0.1)**

Example:

```
#include "FPU.h"

float32 fInput,fOutput;

main()
{
  fOutput = __ffsqrtf(fInput);
}
```

Benchmark Information:

This is a C function and is not benchmarked

3.39 Optimized Memory Copy

Description:

Header File:
FPU.h

Declaration:
This module performs optimized memory copies

```
void memcpy_fast(void* dst, const void* src, Uint16 N)
```

Usage:
memcpy_fast(dst, src, N);

void* dst
pointer to destination

const void* src
pointer to source

Uint16 N
number of 16-bit words to copy

Alignment Requirements:
None

Notes:

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

Example:

```
#include "FPU.h"
#define  N    256

float32 y[N];
float32 x[N];
float32 z;

main()
{
  memcpy_fast(x, y, N<<1);          // Start with same data
  z = median_noreorder_SP_RV(x, N);
}
```

Benchmark Information:
Number of Cycles = 1 cycle per copy + 20 cycles of overhead (including the call and return).
This assumes src and dst are located in different internal RAM blocks.

3.40 Optimized Memory Set

Description:
This module performs optimized memory sets

Header File:
FPU.h

Declaration:
```
void memset_fast(void*, int16, Uint16)
```

Usage:
memset_fast(dst, value, N);

void* dst
pointer to destination
int16 value
initialization value
Uint16 N
number of 16-bit words to initialize

Alignment Requirements:
None

Notes:
1. **The function checks for the case of N=0 and just returns if true.**

Example:
```
#include "FPU.h"
#define  N    10
int x1[N];

main()
{
  memset_fast(x1, 4, N);
}
```

Benchmark Information:
Number of Cycles = 1 cycle per copy + 20 cycles of overhead (including the call and return). This assumes src and dst are located in different internal RAM blocks.

4 Revision History

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

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