TMS320C67x DSP Library Programmer's Reference Guide

Literature Number: SPRU657C
January 2010

TEXAS INSTRUMENTS

Preface

Read This First

About This Manual

Welcome to the TMS320C67x digital signal processor (DSP) Library or DSPLIB, for short. The DSPLIB is a collection of 64 high-level optimized DSP functions for the TMS320C67x device. This source code library includes C-callable functions (ANSI-C language compatible) for general signal processing math and vector functions.

This document contains a reference for the DSPLIB functions and is organized as follows:

- Overview − an introduction to the TI C67x DSPLIB
- Installation − information on how to install and rebuild DSPLIB
- DSPLIB Functions − a quick reference table listing of routines in the library
- DSPLIB Reference − a description of all DSPLIB functions complete with calling convention, algorithm details, special requirements and implementation notes
- Information about performance, fractional Q format and customer support

How to Use This Manual

The information in this document describes the contents of the TMS320C67x DSPLIB in several different ways.

- Chapter 1 provides a brief introduction to the TI C67x DSPLIB, shows the organization of the routines contained in the library, and lists the features and benefits of the DSPLIB.
- Chapter 2 provides information on how to install, use, and rebuild the TI C67x DSPLIB.
- Chapter 3 provides a quick overview of all DSPLIB functions in table format for easy reference. The information shown for each function includes the syntax, a brief description, and a page reference for obtaining more detailed information.

- Chapter 4 provides a list of the routines within the DSPLIB organized into functional categories. The functions within each category are listed in alphabetical order and include arguments, descriptions, algorithms, benchmarks, and special requirements.
- Appendix A describes performance considerations related to the C67x DSPLIB and provides information about the Q format used by DSPLIB functions.
- Appendix B provides information about software updates and customer support.

Notational Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays are shown in a `special typeface`.
- In syntax descriptions, the function or macro appears in a **bold typeface** and the parameters appear in plainface within parentheses. Portions of a syntax that are in **bold** should be entered as shown; portions of syntax that are within parentheses describe the type of information that should be entered.
- Macro names are written in uppercase text; function names are written in lowercase.
- The TMS320C67x is also referred to in this reference guide as the C67x.

Related Documentation From Texas Instruments

The following books describe the TMS320C6x devices and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477-8924. When ordering, please identify the book by its title and literature number. Many of these documents can be found on the Internet at http://www.ti.com.

TMS320C62x/C67x Technical Brief (literature number SPRU197) gives an introduction to the ’C62x/C67x digital signal processors, development tools, and third-party support.

TMS320C6000 CPU and Instruction Set Reference Guide (literature number SPRU189) describes the C6000 CPU architecture, instruction set, pipeline, and interrupts for these digital signal processors.

TMS320C6000 Peripherals Reference Guide (literature number SPRU190) describes common peripherals available on the TMS320C6000 digital signal processors. This book includes information on the internal data and program memories, the external memory interface (EMIF), the host port interface (HPI), multichannel buffered serial ports (McBSPs), direct memory access (DMA), enhanced DMA (EDMA), expansion bus, clocking and phase-locked loop (PLL), and the power-down modes.

TMS320C6000 Programmer's Guide (literature number SPRU198) describes ways to optimize C and assembly code for the TMS320C6000 DSPs and includes application program examples.

TMS320C6000 Assembly Language Tools User's Guide (literature number SPRU186) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the C6000 generation of devices.

TMS320C6000 Optimizing C Compiler User's Guide (literature number SPRU187) describes the C6000 C compiler and the assembly optimizer. This C compiler accepts ANSI standard C source code and produces assembly language source code for the C6000 generation of devices. The assembly optimizer helps you optimize your assembly code.

TMS320C6000 Chip Support Library (literature number SPRU401) describes the application programming interfaces (APIs) used to configure and control all on-chip peripherals.

TMS320C62x Image/Video Processing Library (literature number SPRU400) describes the optimized image/video processing functions including many C-callable, assembly-optimized, general-purpose image/video processing routines.

Trademarks

TMS320C6000, TMS320C62x, TMS320C67x, and Code Composer Studio are trademarks of Texas Instruments.

Other trademarks are the property of their respective owners.

Contents

Figures

Tables

Chapter 1

Introduction

This chapter provides a brief introduction to the TI C67x™ DSP Library (DSPLIB), shows the organization of the routines contained in the library, and lists the features and benefits of the DSPLIB.

Topic **Page**

1.1 Introduction to the TI C67x DSPLIB

The TI C67x DSPLIB is an optimized DSP Function Library for C programmers using TMS320C67x devices. It includes C-callable, assembly-optimized general-purpose signal-processing routines. These routines are typically used in computationally intensive real-time applications where optimal execution speed is critical. By using these routines, you can achieve execution speeds considerably faster than equivalent code written in standard ANSI C language. In addition, by providing ready-to-use DSP functions, TI DSPLIB can significantly shorten your DSP application development time.

The TI DSPLIB includes commonly used DSP routines. Source code is provided that allows you to modify functions to match your specific needs.

The routines contained in the library are first classified in to single- and double-precision functions and then they are organized into seven different functional categories.

- Single-precision funtions:
  - Adaptive filtering
    - DSPF_sp_lms
  - Correlation
    - DSPF_sp_autocor
  - FFT
    - DSPF_sp_bitrev_cplx
    - DSPF_sp_cfftr4_dif
    - DSPF_sp_cfftr2_dit
    - DSPF_sp_fftSPxSP
    - DSPF_sp_ifftSPxSP
    - DSPF_sp_icfftr2_dif
  - Filtering and convolution
    - DSPF_sp_fir_cplx
    - DSPF_sp_fir_gen
    - DSPF_sp_fir_r2
    - DSPF_sp_fircirc
    - DSPF_sp_biquad
    - DSPF_sp_iir
    - DSPF_sp_iirlat
    - DSPF_sp_convol

- Math
  - DSPF_sp_dotp_sqr
  - DSPF_sp_dotprod
  - DSPF_sp_dotp_cplx
  - DSPF_sp_maxval
  - DSPF_sp_maxidx
  - DSPF_sp_minval
  - DSPF_sp_vecrecip
  - DSPF_sp_vecsum_sq
  - DSPF_sp_w_vec
  - DSPF_sp_vecmul
- Matrix
  - DSPF_sp_mat_mul
  - DSPF_sp_mat_trans
  - DSPF_sp_mat_mul_cplx
- Miscellaneous
  - DSPF_sp_blk_move
  - DSPF_sp_blk_eswap16
  - DSPF_sp_blk_eswap32
  - DSPF_sp_blk_eswap64
  - DSPF_fltoq15
  - DSPF_sp_minerr
  - DSPF_q15tofl

- Double-precision funtions:
  - Adaptive filtering
    - DSPF_dp_lms
  - Correlation
    - DSPF_dp_autocor
  - FFT
    - DSPF_dp_bitrev_cplx
    - DSPF_dp_cfftr4_dif
    - DSPF_dp_cfftr2
    - DSPF_dp_icfftr2

- Filtering and convolution
  - DSPF_dp_fir_cplx
  - DSPF_dp_fir_gen
  - DSPF_dp_fir_r2
  - DSPF_dp_fircirc
  - DSPF_dp_biquad
  - DSPF_dp_iir
  - DSPF_dp_iirlat
  - DSPF_dp_convol
- Math
  - DSPF_dp_dotp_sqr
  - DSPF_dp_dotprod
  - DSPF_dp_dotp_cplx
  - DSPF_dp_maxval
  - DSPF_dp_maxidx
  - DSPF_dp_minval
  - DSPF_dp_vecrecip
  - DSPF_dp_vecsum_sq
  - DSPF_dp_w_vec
  - DSPF_dp_vecmul
- Matrix
  - DSPF_dp_mat_mul
  - DSPF_dp_mat_trans
  - DSPF_dp_mat_mul_cplx
- Miscellaneous
  - DSPF_dp_blk_move

1.2 Features and Benefits

- Hand-coded assembly-optimized routines
- C and linear assembly source code
- C-callable routines, fully compatible with the TI C6x compiler
- Fractional Q.15-format operands supported on some benchmarks
- Benchmarks (time and code)
- Tested against the C model

Chapter 2

Installing and Using DSPLIB

This chapter provides information on how to install, use, and rebuild the TI C67x DSPLIB.

Topic **Page**

2.1 How to Install the DSP Library

Note: Please read the README.TXT file for specific details of the release.

1) Unzip the C67xDSPLIB_v200.exe file to a temp directory.

2) Double click the file to launch the Install Shield Wizard,

3) Answer all remaining questions presented in the Install Shield dialogue boxes.

You may change the install directory if necessary.

The installation program will install the C67x DSP Library with the following directory structure:

```
c6700
    |
    +-- lib
    |
    +-- include
    |
    +-- bin
    |
    +-- support
    |
    +-- examples
```

2.2 Using DSPLIB

2.2.1 DSPLIB Arguments and Data Types

DSPLIB Types

Table 2−1 shows the data types handled by the DSPLIB.

Table 2−1. DSPLIB Data Types

Name	Size (bits)	Type	Minimum	Maximum
short	16	Integer	−32768	32767
int	32	Integer	−2147483648	2147483647
long	40	Integer	−549755813888	549755813887
pointer	32	Address	0000:0000h	FFFF:FFFFh
Q.15	16	Fraction	−1.0	0.9999694824...
IEEE float	32	Floating point	1.17549435e−38	3.40282347e+38
IEEE double	64	Floating point	2.2250738585072014e−308	1.7976931348623157e+308

DSPLIB Arguments

TI DSPLIB functions typically operate over vector operands for greater efficiency. Even though these routines can be used to process smaller arrays, or even scalars (unless a minimum size requirement is noted), they will be slower for these cases.

- Vector stride is always equal to 1: Vector operands are composed of vector elements held in consecutive memory locations (vector stride equal to 1).
- Complex elements are assumed to be stored in consecutive memory locations with Real data followed by Imaginary data.
- In-place computation is *not* allowed, unless specifically noted: Source and destination arrays should not overlap.

2.2.2 Calling a DSPLIB Function From C

In addition to correctly installing the DSPLIB software, you must follow these steps to include a DSPLIB function in your code:

- Include the function header file corresponding to the DSPLIB function
- Link your code with dsp67x.lib
- Use a correct linker command file for the platform you use. Remember most functions in dsp67x.lib are written assuming little-endian mode of operation.

For example, if you want to call the single precision Autocorrelation DSPLIB function, you would add:

```
#include <dspf_sp_autocor.h>
```

in your C file and compile and link using

```
cl6x main.c –z –o autocor_drv.out –lrts6700.lib –
ldsp67x.lib
```

Code Composer Studio Users

Assuming your C_DIR environment is correctly set up (as mentioned in section 2.1), you would have to add DSPLIB under the Code Composer Studio environment by choosing dsp67x.lib from the menu *Project → Add Files to Project*. Also, you should make sure that you link with the run-time support library, rts6700.lib.

2.2.3 Calling a DSP Function From Assembly

The C67x DSPLIB functions were written to be used from C. Calling the functions from assembly language source code is possible as long as the calling function conforms to the Texas Instruments C6x C compiler calling conventions. Here, the corresponding .h67 header files located in the ‘include’ directory must be included using the ‘.include’ directive. For more information, refer to section 8 (Runtime Environment) of the *TMS320C6000 Optimizing C Compiler User’s Guide* (SPRU187).

2.2.4 How DSPLIB is Tested − Allowable Error

DSPLIB is tested under the Code Composer Studio environment against a reference C implementation. Because of floating point calculation order change for these two implementations, they differ in the results with an allowable tolerance for that particular kernel. Thus every kernel’s test routine (in the driver file) has error tolerance variable defined that gives the maximum value that is acceptable as the error difference.

For example:

```
#define R_TOL    (1e-05)
```

Here, 0.00001 is the maximum difference allowed for output array “r” forreference C code and any other implementation (like serial assembly, intrinsic C, or hand-optimized asm).

The error tolerance is therefore different for different functions.

2.2.5 How DSPLIB Deals With Overflow and Scaling Issues

The DSPLIB functions implement the same functionality of the reference C code. The user is expected to conform to the range requirements specified in the API function, and in addition, take care to restrict the input range in such a way that the outputs do not overflow.

2.2.6 Interrupt Behavior of DSPLIB Functions

Most DSPLIB functions are interrupt-tolerant but not interruptible. The cycle count formula provided for each function can be used to estimate the number of cycles during which interrupts cannot be taken.

2.3 How to Rebuild DSPLIB

If you would like to rebuild DSPLIB (for example, because you modified the source file contained in the archive), you will have to use the mk6x utility as follows:

```
mk6x dsp67x.src –l dsp67x.lib
```

Chapter 3

DSPLIB Function Tables

This chapter provides tables containing all DSPLIB functions, a brief description of each, and a page reference for more detailed information.

Topic **Page**

3.1 Arguments and Conventions Used

The following convention has been followed when describing the arguments for each individual function:

Table 3−1. Argument Conventions

Argument	Description
x,y	Argument reflecting input data vector
r	Argument reflecting output data vector
nx,ny,nr	Arguments reflecting the size of vectors x,y, and r, respectively. For functions in the case nx = ny = nr, only nx has been used across.
h	Argument reflecting filter coefficient vector (filter routines only)
nh	Argument reflecting the size of vector h
w	Argument reflecting FFT coefficient vector (FFT routines only)

3.2 DSPLIB Functions

The routines included in the DSP library — both single- and double-precision function — are organized into seven functional categories and are listed below in alphabetical order.

- Adaptive filtering
- Correlation
- FFT
- Filtering and convolution
- Math
- Matrix
- Miscellaneous

3.3 DSPLIB Function Tables

3.3.1 Single-Precision Functions

Table 3−2. Adaptive Filtering

Functions	Description	Page
float DSPF_sp_lms (float *x, float *h, float *desired, float *r, float adaptrate, float error, int nh, int nr)	LMS adaptive filter	4-2

Table 3−3. Correlation

Functions	Description	Page
void DSPF_sp_autocor (float *r, float*x, int nx, int nr)	Autocorrelation	4-4

Table 3−4. FFT

Functions	Description	Page
void DSPF_sp_bitrev_cplx (double *x, short *index, int nx)	Complex bit reverse	4-5
void DSPF_sp_cfftr4_dif (float *x, float *w, short n)	Complex radix 4 FFT using DIF	4-9
void DSPF_sp_cfftr2_dit (float *x, float *w, short n)	Complex radix 2 FFT using DIT	4-13
void DSPF_sp_fftSPxSP (int N, float *ptr_x, float *ptr_w, float *ptr_y, unsigned char *brev, int n_min, int offset, int n_max)	Cache optimized mixed radix FFT with digit reversal	4-17
void DSPF_sp_ifftSPxSP (int N, float *ptr_x, float *ptr_w, float *ptr_y, unsigned char *brev, int n_min, int offset, int n_max)	Cache optimized mixed radix inverse FFT with complex input	4-25
void DSPF_sp_icfftr2_dif (float *x, float *w, short n)	Complex radix 2 inverse FFT using DIF	4-34

Table 3−5. Filtering and Convolution

Functions	Description	Page
void DSPF_sp_fir_cplx (float *x, float *h, float *r, int nh, int nr)	Complex FIR filter (radix 2)	4-38
void DSPF_sp_fir_gen (float *x, float *h, float *r, int nh, int nr)	FIR filter (general purpose)	4-40
void DSPF_sp_fir_r2 (float *x, float *h, float *r, int nh, int nr)	FIR filter (radix 2)	4-42
void DSPF_sp_fircirc (float x[], float h[], float r[], int index, int csize, int nh, int nr)	FIR filter with circularly addressed input	4-43
void DSPF_sp_biquad (float x[], float b[], float a[], float delay[], float r[], int nx)	Biquad filter (IIR of second order)	4-45
void DSPF_sp_iir (float *r1, float *x, float *r2, float *h2, float *h1, int nr)	IIR filter (used in VSELP vocoder)	4-47
void DSPF_sp_iirlat (float *x, int nx, float *k, int nk, float *b, float *r)	All-pole IIR lattice filter	4-49
void DSPF_sp_convol (float *x, float *h, float *r, int nh, int nr)	Convolution	4-50

Table 3−6. Math

Functions	Description	Page
float DSPF_sp_dotp_sqr (float G, float *x, float *y, float *r, int nx)	Vector dot product and square	4-52
float DSPF_sp_dotprod (float*x, float*y, int nx)	Vector dot product	4-53
void DSPF_sp_dotp_cplx (float *x, float *y, int n, float *re, float *im)	Complex vector dot product	4-54
float DSPF_sp_maxval (float *x, int nx)	Maximum value of a vector	4-56
int DSPF_sp_maxidx (float *x, int nx)	Index of the maximum element of a vector	4-57
float DSPF_sp_minval (float *x, int nx)	Minimum value of a vector	4-58
void DSPF_sp_vecrecip (float *x, float *r, int n)	Vector reciprocal	4-60
float DSPF_sp_vecsum_sq (float *x, int n)	Sum of squares	4-61
void DSPF_sp_w_vec (float *x, float *y, float m, float *r, int nr)	Weighted vector sum	4-62
void DSPF_sp_vecmul (float *x, float *y, float *r, int n)	Vector multiplication	4-63

Table 3−7. Matrix

Functions	Description	Page
void DSPF_sp_mat_mul (float *x, int r1, int c1, float *y, int c2, float *r)	Matrix multiplication	4-64
void DSPF_sp_mat_trans (float *x, int rows, int cols, float *r)	Matrix transpose	4-66
void DSPF_sp_mat_mul_cplx (float *x, int r1, int c1, float *y, int c2, float *r)	Complex matrix multiplication	4-67

Table 3−8. Miscellaneous

Functions	Description	Page
void DSPF_sp_blk_move (float*x, float*r, int nx)	Move a block of memory	4-69
void DSPF_blk_eswap16 (void *x, void *r, int nx)	Endianswap a block of 16-bit values	4-70
void DSPF_blk_eswap32 (void *x, void *r, int nx)	Endian-swap a block of 32-bit values	4-72
void DSPF_blk_eswap64 (void *x, void *r, int nx)	Endian-swap a block of 64-bit values	4-74
void DSPF_fltoq15 (float *x, short *r, int nx)	Float to Q15 conversion	4-76
float DSPF_sp_minerr (float *GSP0_TABLE,float *errCoefs, int *max_index)	Minimum energy error search	4-77
void DSPF_q15tofl (short *x, float *r, int nx)	Q15 to float conversion	4-78

3.3.2 Double-Precision Functions

Table 3−9. Adaptive Filtering

Functions	Description	Page
double DSPF_dp_lms (double *x, double *h, double *desired, double *r, double adaptrate, double error, int nh, int nr)	LMS adaptive filter	4-80

Table 3−10. Correlation

Functions	Description	Page
void DSPF_dp_autocor (double *r, double*x, int nx, int nr)	Autocorrelation	4-82

Table 3−11.FFT

Functions	Description	Page
void DSPF_dp_bitrev_cplx (double *x, short *index, int n)	Complex bit reverse	4-83
void DSPF_dp_cfftr4_dif (double *x, double *w, short n)	Complex radix 4 FFT using DIF	4-87
void DSPF_dp_cfftr2 (short n, double *x, double *w, short n_min)	Cache optimized radix 2 FFT with complex input	4-91
void DSPF_dp_icfftr2 (short n, double *x, double *w, short n_min)	Cache optimized radix 2 Inverse FFT with complex input	4-96

Table 3−12. Filtering and Convolution

Functions	Description	Page
void DSPF_dp_fir_cplx (double *x, double *h, double *r, int nh, int nr)	Complex FIR filter (radix 2)	4-101
void DSPF_dp_fir_gen (double *x, double *h, double *r, int nh, int nr)	FIR filter (general purpose)	4-103
void DSPF_dp_fir_r2 (double *x, double *h, double *r, int nh, int nr)	FIR filter (radix 2)	4-104
void DSPF_dp_fircirc (double *x, double *h, double *r, int index, int csize, int nh, int nr)	FIR filter with circularly addressed input	4-106
void DSPF_dp_biquad (double *x, double *b, double *a, double *delay, double *r, int nx)	Biquad filter (IIR of second order)	4-108
void DSPF_dp_iir (double *r1, double *x, double *r2, double *h2, double *h1, int nr)	IIR filter (used in VSELP vocoder)	4-109
void DSPF_dp_iirlat (double *x, int nx, double *k, int nk, double *b, double *r)	All-pole IIR lattice filter	4-111
void DSPF_dp_convol (double *x, double *h, double *r, int nh, int nr)	Convolution	4-112

Table 3−13. Math

Functions	Description	Page
double DSPF_dp_dotp_sqr (double G, double *x, double *y, double *r, int nx)	Vector dot product and square	4-114
double DSPF_dp_dotprod (double*x, double*y, int nx)	Vector dot product	4-115
void DSPF_dp_dotp_cplx (double *x, double *y, int n, double *re, double *im)	Complex vector dot product	4-116
double DSPF_dp_maxval (double *x, int nx)	Maximum value of a vector	4-117
int DSPF_dp_maxidx (double *x, int nx)	Index of the maximum element of a vector	4-119
double DSPF_dp_minval (double *x, int nx)	Minimum value of a vector	4-120
void DSPF_dp_vecrecip (double *x, double *r, int n)	Vector reciprocal	4-121
double DSPF_dp_vecsum_sq (double *x, int n)	Sum of squares	4-122
void DSPF_dp_w_vec (double *x, double *y, double m, double *r, int nr)	Weighted vector sum	4-123
void DSPF_dp_vecmul (double *x, double *y, double *r, int n)	Vector multiplication	4-124

Table 3−14. Matrix

Functions	Description	Page
void DSPF_dp_mat_mul (double *x, int r1, int c1, double *y, int c2, double *r)	Matrix multiplication	4-126
void DSPF_dp_mat_trans (double *x, int rows, int col, double *r)	Matrix transpose	4-128
void DSPF_dp_mat_mul_cplx (double *x, int r1, int c1, double *y, int r2, double *r)	Complex matrix multiplication	4-129

Table 3−15. Miscellaneous

Functions	Description	Page
void DSPF_dp_blk_move (double*x, double*r, int nx)	Move a block of memory	4-131

Chapter 4

DSPLIB Reference

This chapter provides a list of the single- and double-precision functions within the DSP library (DSPLIB) organized into functional categories. The functions within each category are listed in alphabetical order and include arguments, descriptions, algorithms, benchmarks, and special requirements.

Topic **Page**

4.1 Single-Precision Functions

4.1.1 Adaptive Filtering

DSPF_sp_lms *Single-precision floating-point LMS algorithm*

Function float DSPF_sp_lms (float *x, float *h, float *desired, float *r, float adapt rate, float error, int nh, int nr)

Arguments

x	Pointer to input samples
h	Pointer to the coefficient array
desired	Pointer to the desired output array
r	Pointer to filtered output array
adapt rate	Adaptation rate
error	Initial error
nh	Number of coefficients
nr	Number of output samples

Description The DSPF_sp_lms implements an LMS adaptive filter. Given an actual input signal and a desired input signal, the filter produces an output signal, the final coefficient values, and returns the final output error signal.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
float DSPF_sp_lms(float *x,float *h,float *y, int nh,
       float *d,float ar, short nr, float error)
{
    int i,j;
    float sum;
    for (i = 0; i < nr; i++)
    {
        for (j = 0; j < nh; j++)
        {
            h[j] = h[j] + (ar*error*x[i+j-1]);
        }
```

```
        sum = 0.0f;
        for (j = 0; j < nh; j++)
        {
            sum += h[j] * x[i+j];
        }
        y[i] = sum;
        error = d[i] - sum;
    }
    return error;
}
```

Special Requirements

- The inner-loop counter must be a multiple of 6 and ≥6.
- Little endianness is assumed.
- Extraneous loads are allowed in the program.
- The coefficient array is assumed to be in reverse order; i.e., h(nh−1), h(nh−2), ..., h(0) will hold coefficients h0, h1, ..., hnh−1, respectively.
- The x[−1] value is assumed to be 0.

Implementation Notes

- The inner loop is unrolled six times to allow update of six coefficients in the kernel.
- The outer loop has been unrolled twice to enable use of LDDW for loading the input coefficients.
- LDDW instruction is used to load the coefficients.
- Register sharing is used to make optimal use of available registers.
- The outer loop instructions are scheduled in parallel with epilog and prolog wherever possible.
- The error term needs to be computed in the outer loop before a new iteration of the inner loop can start. As a result the prolog cannot be placed in parallel with epilog (after the loop kernel).
- Pushing and popping variables from the stack does not really add any overhead except increase stack size. This is because the pops and pushes are done in the delay slots of the outer loop instructions.
- ***Endianness:*** This code is little endian.

- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	(nh + 35) nr + 21 e.g., for nh = 36 and nr = 64 cycles = 4565
Code size (in bytes)	1376

4.1.2 Correlation

DSPF_sp_autocor *Single-precision autocorrelation*

Function void DSPF_sp_autocor (float * restrict r, const float * restrict x, int nx, int nr)

Arguments

r	Pointer to output array of autocorrelation of length nr.
x	Pointer to input array of length nx+nr. Input data must be padded with nr consecutive zeros at the beginning.
nx	Length of autocorrelation vector.
nr	Length of lags.

Description This routine performs the autocorrelation of the input array x. It is assumed that the length of the input array, x, is a multiple of 2 and the length of the output array, r, is a multiple of 4. The assembly routine computes 4 output samples at a time. It is assumed that input vector x is padded with nr no of zeros in the beginning.

Algorithm This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_sp_autocor(float * restrict r, const float *
     restrict x, int nx, int nr)
    {
        int i,k;
        float sum;
        for (i = 0; i < nr; i++)
        {
            sum = 0;
```

```
            for (k = nr; k < nx+nr; k++)
               sum += x[k] * x[k-i];
            r[i] = sum ;
        }
    }
```

Special Requirements

- The value of nx is a multiple of 2 and greater than or equal to 4.
- The value of nr is a multiple of 4 and greater than or equal to 4.
- The value of nx is greater than or equal to nr.
- The x array is double-word aligned.

Implementation Notes

- The inner loop is unrolled twice and the outer loop is unrolled four times.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	(nx/2) * nr + (nr/2) * 5 + 10 − (nr * nr)/4 + nr For nx=64 and nr=64, cycles=1258 For nx=60 and nr=32, cycles=890
Code size (in bytes)	512

4.1.3 FFT

DSPF_sp_bitrev_cplx *Bit reversal for single-precision complex numbers*

Function void DSPF_sp_bitrev_cplx (double *x, short *index, int nx)

Arguments

x	Complex input array to be bit reversed. Contains 2*nx floats.
index	Array of size ~sqrt(nx) created by the routine bitrev_index to allow the fast implementation of the bit reversal.
nx	Number of elements in array x[]. Must be power of 2.

Description

This routine performs the bit-reversal of the input array x[], where x[] is a float array of length 2*nx containing single-precision floating-point complex pairs of data. This routine requires the index array provided by the program below. This index should be generated at compile time, not by the DSP. TI retains all rights, title and interest in this code and only authorizes the use of the bit-reversal code and related table generation code with TMS320 family DSPs manufactured by TI.

```
/* ------------------------------------------------------------------------ */
/* This routine calculates the index for bit reversal of */
/* an array of length nx. The length of the index table is */
/* 2^(2*ceil(k/2)) where nx = 2^k. */
/* */
/* In other words, the length of the index table is: */
/* - for even power of radix: sqrt(nx) */
/* - for odd power of radix: sqrt(2*nx) */
/* ------------------------------------------------------------------------ */
void bitrev_index(short *index, int nx)
{
    int i, j, k, radix = 2;
    short nbits, nbot, ntop, ndiff, n2, raddiv2;
    nbits = 0;
    i = nx;
    while (i > 1)
    {
        i = i >> 1;
        nbits++;
    }
    raddiv2 = radix >> 1;
    nbot = nbits >> raddiv2;
    nbot = nbot << raddiv2 - 1;
    ndiff = nbits & raddiv2;
    ntop = nbot + ndiff;
    n2 = 1 << ntop;
    index[0] = 0;
    for ( i = 1, j = n2/radix + 1; i < n2 - 1; i++)
    {
        index[i] = j - 1;
        for (k = n2/radix; k*(radix-1) < j; k /= radix)
            j -= k*(radix-1);
        j += k;
    }
    index[n2 - 1] = n2 - 1;
}
```

Algorithm

This is the C equivalent for the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_sp_bitrev_cplx(double* x, short* index, int nx)
{
   int    i;
   short  i0, i1, i2, i3;
   short  j0, j1, j2, j3;
```

```
double xi0, xi1, xi2, xi3;
double xj0, xj1, xj2, xj3;
short  t;
int    a, b, ia, ib, ibs;
int    mask;
int    nbits, nbot, ntop, ndiff, n2, halfn;
nbits = 0;
i = nx;
while (i > 1)
{
   i = i >> 1;
   nbits++;
}
nbot    = nbits >> 1;
ndiff   = nbits & 1;
ntop    = nbot + ndiff;
n2       = 1 << ntop;
mask    = n2 - 1;
halfn   = nx >> 1;
for (i0 = 0; i0 < halfn; i0 += 2)
{
    b       = i0 & mask;
    a       = i0 >> nbot;
    if (!b) ia = index[a];
    ib      = index[b];
    ibs     = ib << nbot;
    j0      = ibs + ia;
    t       = i0 < j0;
    xi0     = x[i0];
    xj0     = x[j0];
    if (t)
    {
       x[i0] = xj0;
       x[j0] = xi0;
    }
    i1      = i0 + 1;
```

```
            j1      = j0 + halfn;
            xi1     = x[i1];
            xj1     = x[j1];
            x[i1] = xj1;
            x[j1] = xi1;
            i3      = i1 + halfn;
            j3      = j1 + 1;
            xi3     = x[i3];
            xj3     = x[j3];
            if (t)
            {
               x[i3] = xj3;
               x[j3] = xi3;
            }
        }
}
```

Special Requirements

- The value of nx must be a power of 2.
- The table from bitrev_index is already created.
- The x array is actually an array of 2*nx floats. It is assumed to be double-word aligned.

Implementation Notes

- LDDW is used to load in one complex number at a time (both the real and the imaginary parts).
- There are 12 stores in 10 cycles but all of them are to locations already loaded. No use of the write buffer is made.
- If nx ≤ 4K one can use the char (8-bit) data type for the index variable. This would require changing the LDH when loading index values in the assembly routine to LDB. This would further reduce the size of the index table by half its size.
- ***Endianness:*** Little endian configuration used.
- ***Interruptibility:*** This code is interrupt-tolerant, but not interruptible.

Benchmarks

Cycles	(5/2)nx + 26 e.g., nx = 256, cycles = 666
Code size (in bytes)	608

DSPF_sp_cfftr4_dif *Single-precision floating-point decimation in frequency radix-4 FFT with complex input*

Function void DSPF_sp_cfftr4_dif (float* x, float* w, short n)

Arguments

x	Pointer to an array holding the input and output floating-point array which contains n complex points.
w	Pointer to an array holding the coefficient floating-point array which contains 3*n/4 complex numbers.
n	Number of complex points in x, a power of 4 such that n <= 16K.

Description This routine implements the DIF (decimation in frequency) complex radix 4 FFT with digit-reversed output and normal order input. The number of points, n, must be a power of 4 {4, 16, 64, 256, 1024, ...}. This routine is an in-place routine in the sense that the output is written over the input. It is not an in-place routine in the sense that the input is in normal order and the output is in digit-reversed order.

There must be n complex points (2*n values), and 3*n/4 complex coefficients (3*n/2 values).

Each real and imaginary input value is interleaved in the x array {rx0, ix0, rx1, ix2, ...} and the complex numbers are in normal order. Each real and imaginary output value is interleaved in the x array and the complex numbers are in digit-reversed order {rx0, ix0, ...}. The real and imaginary values of the coefficients are interleaved in the w array {rw0, −iw0, rw1, −iw1, ...} and the complex numbers are in normal order.

Note that the imaginary coefficients are negated {cos(d*0), sin(d*0), cos(d*1), sin(d*1), ...} rather than {cos(d*0), −sin(d*0), cos(d*1), −sin(d*1), ...} where d = 2*PI/n. The value of w(n,k) is usually written w(n,k) = e^−j(2*PI*k/n) = cos(2*PI*k/n) − sin(2*PI*k/n). The routine can be used to implement an inverse FFT by performing the complex conjugate on the input complex numbers (negating the imaginary value), and dividing the result by n. Another method to use the FFT to perform an inverse FFT, is to swap the real and imaginary values of the input and the result, and divide the result by n. In either case, the input is still in normal order and the output is still in digit-reversed order. Note that you can not make the radix 4 FFT into an inverse FFT by using the complex conjugate of the coefficients as you can do with the complex radix 2 FFT.

If you label the input locations from 0 to (n−1) (normal order), the digit-reversed locations can be calculated by reversing the order of the bit pairs of the labels.

For example, for a 1024 point FFT, the digit-reversed location for
617d = 1001101001b = 10 01 10 10 01 is
422d = 0110100110b = 01 10 10 01 10 and vice versa.

The twiddle factor array w can be generated by the gen_twiddle function provided in support\fft\tw_r4fft.c. The .exe file for this function, bin\tw_r4fft.exe, can be used to dump the twiddle factor array into a file.

The function bit_rev in support\fft\bit_rev.c can be used to bit reverse the output array in order to convert it to normal order.

Algorithm

This is the C equivalent for the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_sp_cfftr4_dif(float* x, float* w, short n)
{
    short n1, n2, ie, ia1, ia2, ia3, i0, i1, i2, i3, j, k;
    float r1, r2, r3, r4, s1, s2, s3, s4, co1, co2, co3, si1,
si2, si3;
    n2 = n;
    ie = 1;
    for(k=n; k>1; k>>=2)
    {
       n1 = n2;
       n2 >>= 2;
       ia1 = 0;
       for(j=0; j<n2; j++)
       {
           ia2 = ia1 + ia1;
           ia3 = ia1 + ia2;
           co1 = w[ia1*2];
           si1 = w[ia1*2 + 1];
           co2 = w[ia2*2];
           si2 = w[ia2*2 + 1];
           co3 = w[ia3*2];
           si3 = w[ia3*2 + 1];
           ia1 += ie;
           for(i0=j; i0<n; i0+=n1)
           {
               i1 = i0 + n2;
```

```
                    i2 = i1 + n2;
                    i3 = i2 + n2;
                    r1 = x[i0*2]   + x[i2*2];
                    r3 = x[i0*2]   - x[i2*2];
                    s1 = x[i0*2+1] + x[i2*2+1];
                    s3 = x[i0*2+1] - x[i2*2+1];
                    r2 = x[i1*2]   + x[i3*2];
                    r4 = x[i1*2]   - x[i3*2];
                    s2 = x[i1*2+1] + x[i3*2+1];
                    s4 = x[i1*2+1] - x[i3*2+1];
                    x[i0*2]   = r1 + r2;
                    r2        = r1 - r2;
                    r1        = r3 - s4;
                    r3        = r3 + s4;
                    x[i0*2+1] = s1 + s2;
                    s2        = s1 - s2;
                    s1        = s3 + r4;
                    s3        = s3 - r4;
                    x[i1*2]   = co1*r3 + si1*s3;
                    x[i1*2+1] = co1*s3 - si1*r3;
                    x[i2*2]   = co2*r2 + si2*s2;
                    x[i2*2+1] = co2*s2 - si2*r2;
                    x[i3*2]   = co3*r1 + si3*s1;
                    x[i3*2+1] = co3*s1 - si3*r1;
               }
            }
            ie <<= 2;
        }
    }
```

Special Requirements There are no special alignment requirements.

Implementation Notes

- The two inner loops are executed as one loop with conditional instructions. The variable wcntr is used to determine when the load pointers and coefficient offsets need to be reset.
- The first 8 cycles of the inner loop prolog are conditionally scheduled in parallel with the outer loop. This increases the code size by 12 words, but improves the cycle time.

- A load counter, lcntr, is used so that extraneous loads are not performed.
- If more registers were available, the inner loop could probably be as small as 11 cycles (22 ADDSP/SUBSP instructions). The inner loop was extended to 14 cycles to allow more variables to share registers and thus only need 32 registers.
- The store variable, scntr, is used to determine when the store pointer needs to be reset.
- The variable n2b is used as the outer-loop counter. We are finished when n2b = 0.
- LDDW instructions are not used so that the real and imaginary values can be loaded to separate register files and so that the load and store pointers can use the same offset, n2.
- The outer loop resets the inner loop count to n by multiplying ie by n2b, which is equivalent to ie multiplied by n2, which is always n. The product is always the same since the outer loop shifts n2 to the right by 2 and shifts ie to the left by 2.
- The twiddle factor array w can be generated by the tw_r4fft function provided in dsplib\support\fft\tw_r4fft.c. The exe file for this function, dsplib\bin\tw_r4fft.exe, can be used dump the twiddle factor array into a file.
- The function bit_rev in dsplib\support\fft can be used to bit reverse the output array to convert it into normal order.
- ***Endianness:*** This code is endian neutral.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	(14*n/4 + 23)*log4(n) + 20 e.g., if n = 256, cycles = 3696.
Code size (in bytes)	1184

DSPF_sp_cfftr2_dit *Single-precision floating-point radix-2 FFT with complex input*

Function void DSPF_sp_cfftr2_dit (float * x, float * w, short n)

Arguments

x	Pointer to complex data input.
w	Pointer to complex twiddle factor in bit-reverse order.
n	Length of FFT in complex samples, power of 2 such that n ≥ 32 and n<=32K.

Description This routine performs the decimation-in-time (DIT) radix-2 FFT of the input array x. x has N complex floating-point numbers arranged as successive real and imaginary number pairs. Input array x contains N complex points (N*2 elements). The coefficients for the FFT are passed to the function in array w which contains N/2 complex numbers (N elements) as successive real and imaginary number pairs. The FFT coefficients w are in N/2 bit-reversed order The elements of input array x are in normal order The assembly routine performs 4 output samples (2 real and 2 imaginary) for a pass through inner loop.

```
How to Use

void main(void)
     {
      gen_w_r2(w, N); // Generate coefficient table
      bit_rev(w, N>>1); // Bit-reverse coefficient table
      DSPF_sp_cfftr2_dit(x, w, N);
                      // input in normal order, output in
                      // order bit-reversed
                      // coefficient table in bit-reversed
                      // order
     }
```

Note that (bit-reversed) coefficients for higher order FFT (1024 point) can be used unchanged as coefficients for a lower order FFT (512, 256, 128 ... ,2) The routine can be used to implement inverse FFT by any one of the following methods:

1) Inputs (x) are replaced by their complex-conjugate values.
   Output values are divided by N.
2) FFT coefficients (w) are replaced by their complex conjugates.
   Output values are divided by N.
3) Swap real and imaginary values of input.
4) Swap real and imaginary values of output.

Algorithm This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_sp_cfftr2_dit(float* x, float* w, short n)
{
   short n2, ie, ia, i, j, k, m;
   float rtemp, itemp, c, s;
   n2 = n;
   ie = 1;
   for(k=n; k > 1; k >>= 1)
   {
      n2 >>= 1;
      ia = 0;
      for(j=0; j < ie; j++)
      {
         c = w[2*j];
         s = w[2*j+1];
         for(i=0; i < n2; i++)
         {
            m = ia + n2;
            rtemp     = c * x[2*m]   + s * x[2*m+1];
            itemp     = c * x[2*m+1] - s * x[2*m];
            x[2*m]    = x[2*ia]   - rtemp;
            x[2*m+1]  = x[2*ia+1] - itemp;
            x[2*ia]   = x[2*ia]   + rtemp;
            x[2*ia+1] = x[2*ia+1] + itemp;
            ia++;
         }
         ia += n2;
      }
      ie <<= 1;
   }
}
```

The following C code is used to generate the coefficient table (non-bit reversed).

```
#include <math.h>

/* generate real and imaginary twiddle
   table of size n/2 complex numbers */

gen_w_r2(float* w, int n)
```

```
{
   int i;
   float pi = 4.0*atan(1.0);
   float e = pi*2.0/n;
    for(i=0; i < ( n>>1 ); i++)
    {
       w[2*i]   = cos(i*e);
       w[2*i+1] = sin(i*e);
    }
}
```

The following C code is used to bit reverse the coefficients.

```
bit_rev(float* x, int n)
{
  int i, j, k;
  float rtemp, itemp;
  j = 0;
  for(i=1; i < (n-1); i++)
  {
     k = n >> 1;
     while(k <= j)
     {
        j -= k;
        k >>= 1;
     }
     j += k;
     if(i < j)
     {
        rtemp    = x[j*2];
        x[j*2]   = x[i*2];
        x[i*2]   = rtemp;
        itemp    = x[j*2+1];
        x[j*2+1] = x[i*2+1];
        x[i*2+1] = itemp;
     }
```

```
    }
}
```

Special Requirements

- The value of n is an integral power of 2 such that n ≥32 and n <=32K.
- The FFT Coefficients w are in bit-reversed order
- The elements of input array x are in normal order
- The imaginary coefficients of w are negated as {cos(d*0), sin(d*0), cos(d*1), sin(d*1) ...} as opposed to the normal sequence of {cos(d*0), −sin(d*0), cos(d*1), −sin(d*1) ...} where d = 2*PI/n.
- The x and w arrays are double-word aligned.

Implementation Notes

- The two inner loops are combined into one inner loop whose loop count is n/2.
- The prolog has been completely merged with the epilog. But this gives rise to a problem which has not been overcome. The problem is that the minimum trip count is 32. The safe trip count is at least 16 bound by the size of the epilog. In addition because of merging the prolog and the epilog a data dependency via memory is caused which forces n to be at least 32.
- The bit-reversed twiddle factor array w can be generated by using the tw_r2fft function provided in the dsplib\support\fft directory or by running tw_r2fft.exe provided in dsplib\bin. The twiddle factor array can also be generated by using gen_w_r2 and bit_rev algorithms as described above.
- The function bit_rev in dsplib\support\fft can be used to bit reverse the output array to convert it into normal order.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	(2 * n * log(base−2) n) + 42 For n = 64, Cycles = 810
Code size (in bytes)	1248

DSPF_sp_fftSPxSP *Single-precision floating-point mixed radix forwards FFT with complex input*

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

Arguments

N	Length of fft in complex samples, power of 2 such that N ≥ 8 and N ≤ 8192.
ptr_x	Pointer to complex data input.
ptr_w	Pointer to complex twiddle factor (see below).
ptr_y	Pointer to complex output data.
brev	Pointer to bit reverse table containing 64 entries.
n_min	Smallest fft butterfly used in computation used for decomposing fft into subffts, see notes.
offset	Index in complex samples of sub-fft from start of main fft.
n_max	Size of main fft in complex samples.

Description The benchmark performs a mixed radix forwards fft using a special sequence of coefficients generated in the following way:

```
/* generate vector of twiddle factors for optimized
algorithm */
void tw_gen(float * w, int N)
{
  int j, k;
  double x_t, y_t, theta1, theta2, theta3;
  const double PI = 3.141592654;
  for (j=1, k=0; j <= N>>2; j = j<<2)
  {
      for (i=0; i < N>>2; i+=j)
      {
          theta1 = 2*PI*i/N;
          x_t = cos(theta1);
          y_t = sin(theta1);
          w[k] = (float)x_t;
          w[k+1] = (float)y_t;
          theta2 = 4*PI*i/N;
          x_t = cos(theta2);
          y_t = sin(theta2);
          w[k+2] = (float)x_t;
          w[k+3] = (float)y_t;
          theta3 = 6*PI*i/N;
          x_t = cos(theta3);
          y_t = sin(theta3);
          w[k+4] = (float)x_t;
```

```
            w[k+5] = (float)y_t;
            k+=6;
        }
    }
}
```

This redundant set of twiddle factors is size 2*N float samples. The function is accurate to about 130dB of signal to noise ratio to the DFT function below:

```
void dft(int N, float x[], float y[])
{
   int k,i, index;
   const float PI = 3.14159654;
   float * p_x;
   float arg, fx_0, fx_1, fy_0, fy_1, co, si;
   for(k = 0; k<N; k++)
   {
     p_x = x;
     fy_0 = 0;
     fy_1 = 0;
     for(i=0; i<N; i++)
     {
       fx_0 = p_x[0];
       fx_1 = p_x[1];
       p_x += 2;
       index = (i*k) % N;
       arg = 2*PI*index/N;
       co = cos(arg);
       si = -sin(arg);
       fy_0 += ((fx_0 * co) - (fx_1 * si));
       fy_1 += ((fx_1 * co) + (fx_0 * si));
     }
     y[2*k] = fy_0;
     y[2*k+1] = fy_1;
   }
}
```

The function takes the table and input data and calculates the fft producing the frequency domain data in the Y array. As the fft allows every input point to effect every output point in a cache based system such as the c6711, this causes cache thrashing. This is mitigated by allowing the main fft of size N to be divided into several steps, allowing as much data reuse as possible. For example the following function:

```
DSPF_sp_fftSPxSP(1024, &x[0],&w[0],y,brev,4, 0,1024);
```

is equivalent to:

```
DSPF_sp_fftSPxSP(1024,&x[2*0], &w[0] , y,brev,256, 0,1024;
DSPF_sp_fftSPxSP(256, &x[2*0], &w[2*768],y,brev,4, 0,1024;
DSPF_sp_fftSPxSP(256, &x[2*256],&w[2*768],y,brev,4, 256,1024;
DSPF_sp_fftSPxSP(256, &x[2*512],&w[2*768],y,brev,4, 512,1024;
DSPF_sp_fftSPxSP(256, &x[2*768],&w[2*768],y,brev,4, 768,1024;
```

Notice how the first fft function is called on the entire 1K data set it covers the first pass of the fft until the butterfly size is 256. The following 4 ffts do 256 pt ffts 25% of the size. These continue down to the end when the butterfly is of size 4. They use an index to the main twiddle factor array of 0.75*2*N. This is because the twiddle factor array is composed of successively decimated versions of the main array. N not equal to a power of 4 can be used, i.e. 512. In this case to decompose the fft the following would be needed :

```
sp_fftSPxSP_asm(512, &x_asm[0],&w[0],y_asm,brev,2, 0,512);
```

is equivalent to:

```
sp_fftSPxSP_asm(512, &x_asm[2*0], &w[0] , y_asm,brev,128,
0,512)
sp_fftSPxSP_asm(128, &x_asm[2*0], &w[2*384],y_asm,brev,4,
0,512)
sp_fftSPxSP_asm(128, &x_asm[2*128],&w[2*384],y_asm,brev,4,
128,512)
sp_fftSPxSP_asm(128, &x_asm[2*256],&w[2*384],y_asm,brev,4,
256,512)
sp_fftSPxSP_asm(128, &x_asm[2*384],&w[2*384],y_asm,brev,4,
384,512)
```

The twiddle factor array is composed of log4(N) sets of twiddle factors, (3/4)*N, (3/16)*N, (3/64)*N, etc. The index into this array for each stage of the fft is calculated by summing these indices up appropriately. For multiple ffts they can share the same table by calling the small ffts from further down in the twiddle factor array. In the same way as the decomposition works for more data reuse. Thus, the above decomposition can be summarized for a general N, radix "rad" as follows:

```
sp_fftSPxSP_asm(N, &x[0], &w[0], y,brev,N/4,0, N)
sp_fftSPxSP_asm(N/4,&x[0], &w[2*3*N/4],y,brev,rad,0, N)
sp_fftSPxSP_asm(N/4,&x[2*N/4], &w[2*3*N/4],y,brev,rad,N/4, N)
sp_fftSPxSP_asm(N/4,&x[2*N/2], &w[2*3*N/4],y,brev,rad,N/2, N)
sp_fftSPxSP_asm(N/4,&x[2*3*N/4],&w[2*3*N/4],y,brev,rad,3*N/4,
N)
```

As discussed previously, N can be either a power of 4 or 2. If N is a power of 4, then rad = 4, and if N is a power of 2 and not a power of 4, then rad = 2. "rad" is used to control how many stages of decomposition are performed. It is also used to determine whether a radix-4 or radix-2 decomposition should be performed at the last stage. Hence when "rad" is set to "N/4" the first stage of the transform alone is performed and the code exits. To complete the FFT, four other calls are required to perform N/4 size FFTs. In fact, the ordering of these 4 FFTs amongst themselves does not matter and hence from a cache perspective, it helps to go through the remaining 4 FFTs in exactly the opposite order to the first. This is illustrated as follows:

```
sp_fftSPxSP_asm(N, &x[0], &w[0], y,brev,N/4,0, N)
sp_fftSPxSP_asm(N/4,&x[2*3*N/4],&w[2*3*N/4],y,brev,rad,3*N/4,
N)
sp_fftSPxSP_asm(N/4,&x[2*N/2], &w[2*3*N/4],y,brev,rad,N/2, N)
sp_fftSPxSP_asm(N/4,&x[2*N/4], &w[2*3*N/4],y,brev,rad,N/4, N)
sp_fftSPxSP_asm(N/4,&x[0], &w[2*3*N/4],y,brev,rad,0, N)
```

In addition this function can be used to minimize call overhead, by completing the FFT with one function call invocation as shown below:

```
sp_fftSPxSP_asm(N, &x[0], &w[0], y, brev, rad, 0, N)
```

Algorithm This is the C equivalent of the assembly code without restrictions: Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_sp_fftSPxSP(int N, float *ptr_x, float *ptr_w,
         float *ptr_y,

   unsigned char *brev, int n_min, int offset, int n_max)

{

        int  i, j, k, l1, l2, h2, predj;

        int  tw_offset, stride, fft_jmp;

        float x0, x1, x2, x3,x4,x5,x6,x7;

        float xt0, yt0, xt1, yt1, xt2, yt2, yt3;

        float yt4, yt5, yt6, yt7;

        float si1,si2,si3,co1,co2,co3;

        float xh0,xh1,xh20,xh21,xl0,xl1,xl20,xl21;

        float x_0, x_1, x_l1, x_l1p1, x_h2 , x_h2p1, x_l2,
x_l2p1;

        float xl0_0, xl1_0, xl0_1, xl1_1;

        float xh0_0, xh1_0, xh0_1, xh1_1;

        float *x,*w;

        int   k0, k1, j0, j1, l0, radix;

        float * y0, * ptr_x0, * ptr_x2;

        radix = n_min;

        stride = N; /* N is the number of complex samples */

        tw_offset = 0;

        while (stride > radix)

        {

            j = 0;

            fft_jmp = stride + (stride>>1);

            h2 = stride>>1;

            l1 = stride;
```

```
l2 = stride + (stride>>1);
x = ptr_x;
w = ptr_w + tw_offset;
for (i = 0; i < N; i += 4)
{
    co1 = w[j];
    si1 = w[j+1];
    co2 = w[j+2];
    si2 = w[j+3];
    co3 = w[j+4];
    si3 = w[j+5];
    x_0    = x[0];
    x_1    = x[1];
    x_h2   = x[h2];
    x_h2p1 = x[h2+1];
    x_l1   = x[l1];
    x_l1p1 = x[l1+1];
    x_l2   = x[l2];
    x_l2p1 = x[l2+1];
    xh0  = x_0    + x_l1;
    xh1  = x_1    + x_l1p1;
    xl0  = x_0    – x_l1;
    xl1  = x_1    – x_l1p1;
    xh20 = x_h2   + x_l2;
    xh21 = x_h2p1 + x_l2p1;
    xl20 = x_h2   – x_l2;
    xl21 = x_h2p1 – x_l2p1;
    ptr_x0 = x;
    ptr_x0[0] = xh0 + xh20;
    ptr_x0[1] = xh1 + xh21;
    ptr_x2 = ptr_x0;
    x += 2;
    j += 6;
    predj = (j – fft_jmp);
    if (!predj) x += fft_jmp;
    if (!predj) j = 0;
```

```
        xt0 = xh0 - xh20;
        yt0 = xh1 - xh21;
        xt1 = xl0 + xl21;
        yt2 = xl1 + xl20;
        xt2 = xl0 - xl21;
        yt1 = xl1 - xl20;
        ptr_x2[l1  ] = xt1 * co1 + yt1 * si1;
        ptr_x2[l1+1] = yt1 * co1 - xt1 * si1;
        ptr_x2[h2  ] = xt0 * co2 + yt0 * si2;
        ptr_x2[h2+1] = yt0 * co2 - xt0 * si2;
        ptr_x2[l2  ] = xt2 * co3 + yt2 * si3;
        ptr_x2[l2+1] = yt2 * co3 - xt2 * si3;
    }
    tw_offset += fft_jmp;
    stride = stride>>2;
}/* end while */
j = offset>>2;
ptr_x0 = ptr_x;
y0 = ptr_y;
/*l0 = _norm(n_max) - 17;    get size of fft */
l0=0;
for(k=30;k>=0;k--)
    if( (n_max & (1 << k)) == 0 )
      l0++;
    else
       break;
l0=l0-17;
if (radix <= 4) for (i = 0; i < N; i += 4)
{
        /* reversal computation */
        j0 = (j     ) & 0x3F;
        j1 = (j >> 6);
        k0 = brev[j0];
        k1 = brev[j1];
        k = (k0 << 6) +  k1;
        k = k >> l0;
```

```
j++;        /* multiple of 4 index */
x0   = ptr_x0[0];  x1 = ptr_x0[1];
x2   = ptr_x0[2];  x3 = ptr_x0[3];
x4   = ptr_x0[4];  x5 = ptr_x0[5];
x6   = ptr_x0[6];  x7 = ptr_x0[7];
ptr_x0 += 8;
xh0_0  = x0 + x4;
xh1_0  = x1 + x5;
xh0_1  = x2 + x6;
xh1_1  = x3 + x7;
if (radix == 2) {
  xh0_0 = x0;
  xh1_0 = x1;
  xh0_1 = x2;
  xh1_1 = x3;
}
yt0  = xh0_0 + xh0_1;
yt1  = xh1_0 + xh1_1;
yt4  = xh0_0 – xh0_1;
yt5  = xh1_0 – xh1_1;
xl0_0  = x0 – x4;
xl1_0  = x1 – x5;
xl0_1  = x2 – x6;
xl1_1  = x3 – x7;
if (radix == 2) {
  xl0_0 = x4;
  xl1_0 = x5;
  xl1_1 = x6;
  xl0_1 = x7;
}
yt2  = xl0_0 + xl1_1;
yt3  = xl1_0 – xl0_1;
yt6  = xl0_0 – xl1_1;
yt7  = xl1_0 + xl0_1;
if (radix == 2) {
  yt7  = xl1_0 – xl0_1;
```

```
                yt3  = xl1_0 + xl0_1;
              }
              y0[k] = yt0; y0[k+1] = yt1;
              k += n_max>>1;
              y0[k] = yt2; y0[k+1] = yt3;
              k += n_max>>1;
              y0[k] = yt4; y0[k+1] = yt5;
              k += n_max>>1;
              y0[k] = yt6; y0[k+1] = yt7;
        }
}
```

Special Requirements

- The value of N must be a power of 2 and N ≥ 8 N ≤ 8192 points.
- Complex time data x and twiddle factors w are aligned on double-word boundaries. Real values are stored in even word positions and imaginary values in odd positions.
- All data is in single-precision floating-point format. The complex frequency data will be returned in linear order.

Implementation Notes

- A special sequence of coeffs. used as generated above produces the fft. This collapses the inner 2 loops in the traditional Burrus and Parks implementation Fortran code.
- The revised FFT uses a redundant sequence of twiddle factors to allow a linear access through the data. This linear access enables data and instruction level parallelism.
- The data produced by the DSPF_sp_fftSPxSP fft is in normal form, the whole data array is written into a new output buffer.
- The DSPF_sp_fftSPxSP butterfly is bit reversed, i.e. the inner 2 points of the butterfly are crossed over, this has the effect of making the data come out in bit reversed rather than DSPF_sp_fftSPxSP digit-reversed order. This simplifies the last pass of the loop. ia simple table is used to do the bit reversal out of place.

```
unsigned char brev[64] = {
0x0, 0x20, 0x10, 0x30, 0x8, 0x28, 0x18, 0x38,
0x4, 0x24, 0x14, 0x34, 0xc, 0x2c, 0x1c, 0x3c,
0x2, 0x22, 0x12, 0x32, 0xa, 0x2a, 0x1a, 0x3a,
0x6, 0x26, 0x16, 0x36, 0xe, 0x2e, 0x1e, 0x3e,
0x1, 0x21, 0x11, 0x31, 0x9, 0x29, 0x19, 0x39,
0x5, 0x25, 0x15, 0x35, 0xd, 0x2d, 0x1d, 0x3d,
0x3, 0x23, 0x13, 0x33, 0xb, 0x2b, 0x1b, 0x3b,
0x7, 0x27, 0x17, 0x37, 0xf, 0x2f, 0x1f, 0x3f
};
```

- The special sequence of twiddle factors w can be generated using the tw_fftSPxSP_C67 function provided in the dsplib\support\fft\tw_fftSPxSP_C67.c file or by running tw_fftSPxSP_C67.exe in dsplib\bin.
- The brev table required for this function is provided in the file dsplib\support\fft\brev_table.h.
- ***Endianness:*** Configuration is little endian.
- ***Interruptibility:*** An interruptible window of 1 cycle is available between the two outer loops.

Benchmarks

Cycles	cycles = 3 * ceil(log4(N)−1) * N + 21 * ceil(log4(N)−1) + 2*N + 44 e.g., N = 1024, cycles = 14464 e.g., N = 512, cycles = 7296 e.g., N = 256, cycles = 2923 e.g., N = 128, cycles = 1515 e.g., N = 64, cycles = 598
Code size (in bytes)	1440

DSPF_sp_ifftSPxSP *Single-precision floating-point mixed radix inverse FFT with complex input*

Function

void DSPF_sp_ifftSPxSP (int n, float * ptr_x, float * ptr_w, float * ptr_y, unsigned char * brev, int n_min, int offset, int n_max)

Arguments

n	Length of ifft in complex samples, power of 2 such that n ≥ 8 and n ≤ 8192.
ptr_x	Pointer to complex data input (normal order).
ptr_w	Pointer to complex twiddle factor (see below).
ptr_y	Pointer to complex output data (normal order).
brev	Pointer to bit reverse table containing 64 entries.
n_min	Smallest ifft butterfly used in computation used for decomposing ifft into subiffts, see notes.
offset	Index in complex samples of sub-ifft from start of main ifft.
n_max	Size of main ifft in complex samples.

Description

The benchmark performs a mixed radix forwards ifft using a special sequence of coefficients generated in the following way:

```
/*generate vector of twiddle factors for optimized algorithm*/
 void tw_gen(float * w, int N)
 {
   int j, k;
   double x_t, y_t, theta1, theta2, theta3;
   const double PI = 3.141592654;
   for (j=1, k=0; j <= N>>2; j = j<<2)
   {
       for (i=0; i < N>>2; i+=j)
       {
            theta1 = 2*PI*i/N;
            x_t = cos(theta1);
            y_t = sin(theta1);
            w[k] = (float)x_t;
            w[k+1] = (float)y_t;
            theta2 = 4*PI*i/N;
            x_t = cos(theta2);
            y_t = sin(theta2);
            w[k+2] = (float)x_t;
            w[k+3] = (float)y_t;
            theta3 = 6*PI*i/N;
            x_t = cos(theta3);
            y_t = sin(theta3);
            w[k+4] = (float)x_t;
            w[k+5] = (float)y_t;
            k+=6;
       }
   }
 }
```

This redundant set of twiddle factors is size 2*N float samples. The function is accurate to about 130dB of signal to noise ratio to the IDFT function below:

```
void idft(int n, float x[], float y[])
{
  int k,i, index;
  const float PI = 3.14159654;
  float * p_x;
  float arg, fx_0, fx_1, fy_0, fy_1, co, si;
  for(k = 0; k<n; k++)
  {
    p_x = x;
    fy_0 = 0;
    fy_1 = 0;
    for(i=0; i<n; i++)
    {
      fx_0 = p_x[0];
      fx_1 = p_x[1];
      p_x += 2;
      index = (i*k) % n;
      arg = 2*PI*index/n;
      co = cos(arg);
      si = sin(arg);
```

```
      fy_0 += ((fx_0 * co) - (fx_1 * si));
      fy_1 += ((fx_1 * co) + (fx_0 * si));
    }
    y[2*k] = fy_0/n;
    y[2*k+1] = fy_1/n;
  }
          }
```

The function takes the table and input data and calculates the ifft producing the frequency domain data in the Y array. the output is scaled by a scaling factor of 1/N. As the ifft allows every input point to effect every output point in a cache based system such as the c6711, this causes cache thrashing. This is mitigated by allowing the main ifft of size N to be divided into several steps, allowing as much data reuse as possible. For example the following function:

```
sp_ifftSPxSP_asm(1024, &x[0],&w[0],y,brev,4, 0,1024)
```

is equivalent to:

```
sp_ifftSPxSP(1024,&x[2*0],&w[0],y,brev,256,0,1024)
sp_ifftSPxSP(256,&x[2*0],&w[2*768],y,brev,4,0,1024)
sp_ifftSPxSP(256,&x[2*256],&w[2*768],y,brev,4,256,1024)
sp_ifftSPxSP(256,&x[2*512],&w[2*768],y,brev,4,512,1024)
sp_ifftSPxSP(256,&x[2*768],&w[2*768],y,brev,4,768,1024)
```

Notice how the first ifft function is called on the entire 1K data set it covers the first pass of the ifft until the butterfly size is 256. The following 4 iffts do 256 pt iffts 25% of the size. These continue down to the end when the butterfly is of size 4. They use an index to the main twiddle factor array of 0.75*2*N. This is because the twiddle factor array is composed of successively decimated versions of the main array. N not equal to a power of 4 can be used, i.e. 512. In this case to decompose the ifft the following would be needed :

```
sp_ifftSPxSP_asm(512, &x[0],&w[0],y,brev,2, 0,512)
```

is equivalent to:

```
sp_ifftSPxSP(512, &x[2*0], &w[0] , y,brev,128, 0,512)
sp_ifftSPxSP(128, &x[2*0], &w[2*384],y,brev,4, 0,512)
sp_ifftSPxSP(128, &x[2*128],&w[2*384],y,brev,4, 128,512)
sp_ifftSPxSP(128, &x[2*256],&w[2*384],y,brev,4, 256,512)
sp_ifftSPxSP(128, &x[2*384],&w[2*384],y,brev,4, 384,512)
```

The twiddle factor array is composed of log4(N) sets of twiddle factors, (3/4)*N, (3/16)*N, (3/64)*N, etc. The index into this array for each stage of the ifft is calculated by summing these indices up appropriately. For multiple iffts they can share the same table by calling the small iffts from further down in the twiddle factor array. In the same way as the decomposition works for more data reuse. Thus, the above decomposition can be summarized for a general N radix “rad” as follows:

```
sp_ifftSPxSP(N, &x[0], &w[0], y,brev,N/4,0, N)
sp_ifftSPxSP(N/4,&x[0], &w[2*3*N/4],y,brev,rad,0, N)
sp_ifftSPxSP(N/4,&x[2*N/4], &w[2*3*N/4],y,brev,rad,N/4, N)
sp_ifftSPxSP(N/4,&x[2*N/2], &w[2*3*N/4],y,brev,rad,N/2, N)
sp_ifftSPxSP(N/4,&x[2*3*N/4],&w[2*3*N/4],y,brev,rad,3*N/4,N)
```

As discussed previously, N can be either a power of 4 or 2. If N is a power of 4, then rad = 4, and if N is a power of 2 and not a power of 4, then rad = 2. “rad” is used to control how many stages of decomposition are performed. It is also used to determine whether a radix-4 or radix-2 decomposition should be performed at the last stage. Hence when “rad” is set to “N/4” the first stage of the transform alone is performed and the code exits. To complete the FFT, four other calls are required to perform N/4 size FFTs. In fact, the ordering of these 4 FFTs amongst themselves does not matter and hence from a cache perspective, it helps to go through the remaining 4 FFTs in exactly the opposite order to the first. This is illustrated as follows:

```
sp_ifftSPxSP(N, &x[0], &w[0], y,brev,N/4,0, N)
sp_ifftSPxSP(N/4,&x[2*3*N/4],&w[2*3*N/4],y,brev,rad,3*N/4,N)
sp_ifftSPxSP(N/4,&x[2*N/2], &w[2*3*N/4],y,brev,rad,N/2, N)
sp_ifftSPxSP(N/4,&x[2*N/4], &w[2*3*N/4],y,brev,rad,N/4, N)
sp_ifftSPxSP(N/4,&x[0], &w[2*3*N/4],y,brev,rad,0, N)
```

In addition this function can be used to minimize call overhead, by completing the FFT with one function call invocation as shown below:

```
sp_ifftSPxSP_asm(N, &x[0], &w[0], y, brev, rad, 0,N)
```

Algorithm

This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_sp_ifftSPxSP(int n, float ptr_x[], float ptr_w[],
            float ptr_y[], unsigned char brev[], int n_min,
            int offset, int n_max)
{
    int  i, j, k, l1, l2, h2, predj;
    int  tw_offset, stride, fft_jmp;
    float x0, x1, x2, x3,x4,x5,x6,x7;
    float xt0, yt0, xt1, yt1, xt2, yt2, yt3;
    float yt4, yt5, yt6, yt7;
    float si1,si2,si3,co1,co2,co3;
    float xh0,xh1,xh20,xh21,xl0,xl1,xl20,xl21;
    float x_0, x_1, x_l1, x_l1p1, x_h2 , x_h2p1, x_l2, x_l2p1;
    float xl0_0, xl1_0, xl0_1, xl1_1;
    float xh0_0, xh1_0, xh0_1, xh1_1;
    float *x,*w;
```

```
int   k0, k1, j0, j1, l0, radix;
float * y0, * ptr_x0, * ptr_x2;
radix = n_min;
stride = n; /* n is the number of complex samples */
tw_offset = 0;
while (stride > radix)
{
    j = 0;
    fft_jmp = stride + (stride>>1);
    h2 = stride>>1;
    l1 = stride;
    l2 = stride + (stride>>1);
    x = ptr_x;
    w = ptr_w + tw_offset;
    for (i = 0; i < n; i += 4)
    {
       co1 = w[j];
       si1 = w[j+1];
       co2 = w[j+2];
       si2 = w[j+3];
       co3 = w[j+4];
       si3 = w[j+5];
       x_0    = x[0];
       x_1    = x[1];
       x_h2   = x[h2];
       x_h2p1 = x[h2+1];
       x_l1   = x[l1];
       x_l1p1 = x[l1+1];
       x_l2   = x[l2];
       x_l2p1 = x[l2+1];
       xh0  = x_0    + x_l1;
       xh1  = x_1    + x_l1p1;
       xl0  = x_0    - x_l1;
       xl1  = x_1    - x_l1p1;
       xh20 = x_h2   + x_l2;
       xh21 = x_h2p1 + x_l2p1;
```

```
          xl20 = x_h2   - x_l2;
          xl21 = x_h2p1 - x_l2p1;
          ptr_x0 = x;
          ptr_x0[0] = xh0 + xh20;
          ptr_x0[1] = xh1 + xh21;
          ptr_x2 = ptr_x0;
          x += 2;
          j += 6;
          predj = (j - fft_jmp);
          if (!predj) x += fft_jmp;
          if (!predj) j = 0;
          xt0 = xh0 - xh20;
          yt0 = xh1 - xh21;
          xt1 = xl0 - xl21;
          yt2 = xl1 - xl20;
          xt2 = xl0 + xl21;
          yt1 = xl1 + xl20;
          ptr_x2[l1  ] = xt1 * co1 - yt1 * si1;
          ptr_x2[l1+1] = yt1 * co1 + xt1 * si1;
          ptr_x2[h2  ] = xt0 * co2 - yt0 * si2;
          ptr_x2[h2+1] = yt0 * co2 + xt0 * si2;
          ptr_x2[l2  ] = xt2 * co3 - yt2 * si3;
          ptr_x2[l2+1] = yt2 * co3 + xt2 * si3;
       }
           tw_offset += fft_jmp;
           stride = stride>>2;
     }/* end while */
     j = offset>>2;
     ptr_x0 = ptr_x;
     y0 = ptr_y;
     /*l0 = _norm(n_max) - 17;    get size of fft */
     l0=0;
     for(k=30;k>=0;k--)
         if( (n_max & (1 << k)) == 0 )
             l0++;
         else
```

```
            break;
        l0=l0-17;
        if (radix <= 4) for (i = 0; i < n; i += 4)
        {
            /* reversal computation */
            j0 = (j      ) & 0x3F;
            j1 = (j >> 6);
            k0 = brev[j0];
            k1 = brev[j1];
            k = (k0 << 6) +  k1;
            k = k >> l0;
            j++;        /* multiple of 4 index */
            x0   = ptr_x0[0];  x1 = ptr_x0[1];
            x2   = ptr_x0[2];  x3 = ptr_x0[3];
            x4   = ptr_x0[4];  x5 = ptr_x0[5];
            x6   = ptr_x0[6];  x7 = ptr_x0[7];
            ptr_x0 += 8;
            xh0_0  = x0 + x4;
            xh1_0  = x1 + x5;
            xh0_1  = x2 + x6;
            xh1_1  = x3 + x7;
            if (radix == 2)
    {
                xh0_0 = x0;
                xh1_0 = x1;
                xh0_1 = x2;
                xh1_1 = x3;
            }
            yt0  = xh0_0 + xh0_1;
            yt1  = xh1_0 + xh1_1;
            yt4  = xh0_0 - xh0_1;
            yt5  = xh1_0 - xh1_1;
            xl0_0  = x0 - x4;
            xl1_0  = x1 - x5;
            xl0_1  = x2 - x6;
            xl1_1  = x7 - x3;
```

```
            if (radix == 2)
        {
                xl0_0 = x4;
                xl1_0 = x5;
                xl1_1 = x6;
                xl0_1 = x7;
            }
            yt2  = xl0_0 + xl1_1;
            yt3  = xl1_0 + xl0_1;
            yt6  = xl0_0 - xl1_1;
            yt7  = xl1_0 - xl0_1;
            y0[k] = yt0/n_max; y0[k+1] = yt1/n_max;
            k += n_max>>1;
            y0[k] = yt2/n_max; y0[k+1] = yt3/n_max;
            k += n_max>>1;
            y0[k] = yt4/n_max; y0[k+1] = yt5/n_max;
            k += n_max>>1;
            y0[k] = yt6/n_max; y0[k+1] = yt7/n_max;
      }
}
```

Special Requirements

- The value of N must be a power of 2 and $N \geq 8$, $N \leq 8192$ points.
- Complex time data x and twiddle factors w are aligned on double-word boundaries. Real values are stored in even word positions and imaginary values in odd positions.
- All data is in single-precision floating-point format. The complex frequency data will be returned in linear order.
- The x array must be padded with 16 words at the end.

Implementation Notes

- A special sequence of coeffs. used as generated above produces the ifft. This collapses the inner 2 loops in the traditional Burrus and Parks implementation Fortran code.
- The revised FFT uses a redundant sequence of twiddle factors to allow a linear access through the data. This linear access enables data and instruction level parallelism.

- The data produced by the DSPF_sp_ifftSPxSP ifft is in normal form, the whole data array is written into a new output buffer.
- The DSPF_sp_ifftSPxSP butterfly is bit reversed, i.e., the inner 2 points of the butterfly are crossed over, this has the effect of making the data come out in bit reversed rather than DSPF_sp_ifftSPxSP digit reversed order. This simplifies the last pass of the loop. ia simple table is used to do the bit reversal out of place.

```
unsigned char brev[64] = {
0x0, 0x20, 0x10, 0x30, 0x8, 0x28, 0x18, 0x38,
0x4, 0x24, 0x14, 0x34, 0xc, 0x2c, 0x1c, 0x3c,
0x2, 0x22, 0x12, 0x32, 0xa, 0x2a, 0x1a, 0x3a,
0x6, 0x26, 0x16, 0x36, 0xe, 0x2e, 0x1e, 0x3e,
0x1, 0x21, 0x11, 0x31, 0x9, 0x29, 0x19, 0x39,
0x5, 0x25, 0x15, 0x35, 0xd, 0x2d, 0x1d, 0x3d,
0x3, 0x23, 0x13, 0x33, 0xb, 0x2b, 0x1b, 0x3b,
0x7, 0x27, 0x17, 0x37, 0xf, 0x2f, 0x1f, 0x3f
};
```

- The special sequence of twiddle factors w can be generated using the tw_fftSPxSP_C67 function provided in the dsplib\support\fft\tw_fftSPxSP_C67.c file or by running tw_fftSPxSP_C67.exe in dsplib\bin.
- The brev table required for this function is provided in the file dsplib\support\fft\brev_table.h.
- ***Endianness:*** Configuration is little endian.
- ***Interruptibility:*** This code is *intended* to be interrupt-tolerant but not interruptible. However, a bug in the assembly code for Rev 2.0 and earlier of the library causes this function to *not* be interrupt tolerant. Therefore, in order to safely use this function you must disable interrupts prior to the call and then restore interrupts after.

Benchmarks

Cycles	cycles = 3 * ceil(log4(N)−1) * N + 21*ceil(log4(N)−1) + 2*N + 44 e.g., N = 1024, cycles = 14464 e.g., N = 512, cycles = 7296 e.g., N = 256, cycles = 2923 e.g., N = 128, cycles = 1515 e.g., N = 64, cycles = 598
Code size (in bytes)	1472

DSPF_sp_icfftr2_dif *Single-precision inverse, complex, radix-2, decimation-in-frequency FFT*

Function void DSPF_sp_icfftr2_dif (float* x, float* w, short n)

Arguments

x — Input and output sequences (dim−n) (input/output) x has n complex numbers (2*n SP values). The real and imaginary values are interleaved in memory. The input is in bit-reversed order and output is in normal order.

w — FFT coefficients (dim−n/2) (input) w has n/2 complex numbers (n SP values). FFT coefficients must be in bit-reversed order. The real and imaginary values are interleaved in memory.

n — FFT size (input).

Description This routine is used to compute the inverse, complex, radix-2, decimation-in-frequency Fast Fourier Transform of a single-precision complex sequence of size n, and a power of 2. The routine requires bit-reversed input and bit-reversed coefficients (twiddle factors) and produces results that are in normal order.

Final scaling by 1/N is not done in this function.

```
How To Use
void main(void)
{
   gen_w_r2(w, N); // Generate coefficient table
   bit_rev(w, N>>1); // Bit-reverse coefficient table
   DSPF_sp_cfftr2_dit(x, w, N);
                       // radix-2 DIT forward FFT
                       // input in normal order, output in
                       // order bit-reversed
                       // coefficient table in bit-reversed
                       // order
   DSPF_sp_icfftr2_dif(x, w, N);
                       // Inverse radix 2 FFT
                       // input in bit-reversed order,
                       // order output in normal
                       // coefficient table in bit-reversed
                       // order
   divide(x, N); // scale inverse FFT output
                       // result is the same as original
                       // input
}
```

Algorithm

This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_sp_icfftr2_dif(float* x, float* w, short n)
{
    short n2, ie, ia, i, j, k, m;
    float rtemp, itemp, c, s;
    n2 = 1;
    ie = n;
    for(k=n; k > 1; k >>= 1)
    {
        ie >>= 1;
        ia = 0;
        for(j=0; j < ie; j++)
        {
            c = w[2*j];
            s = w[2*j+1];
            for(i=0; i < n2; i++)
            {
                m = ia + n2;
                rtemp     = x[2*ia]   - x[2*m];
                x[2*ia]   = x[2*ia]   + x[2*m];
                itemp     = x[2*ia+1] - x[2*m+1];
                x[2*ia+1] = x[2*ia+1] + x[2*m+1];
                x[2*m]    = c*rtemp   - s*itemp;
                x[2*m+1]  = c*itemp   + s*rtemp;
                ia++;
            }
           ia += n2;
        }
        n2 <<= 1;
    }
}
```

The following C code is used to generate the coefficient table (non-bit reversed):

```
#include <math.h>
/* generate real and imaginary twiddle
   table of size n/2 complex numbers */
gen_w_r2(float* w, int n)
{
   int i;
   float pi = 4.0*atan(1.0);
   float e = pi*2.0/n;
    for(i=0; i < ( n>>1 ); i++)
    {
       w[2*i]   = cos(i*e);
       w[2*i+1] = sin(i*e);
    }
}
```

The following C code is used to bit-reverse the coefficients:

```
bit_rev(float* x, int n)
{
  int i, j, k;
  float rtemp, itemp;
  j = 0;
  for(i=1; i < (n-1); i++)
  {
     k = n >> 1;
     while(k <= j)
     {
        j -= k;
        k >>= 1;
     }
     j += k;
     if(i < j)
     {
        rtemp    = x[j*2];
        x[j*2]   = x[i*2];
```

```
            x[i*2]   = rtemp;
            itemp    = x[j*2+1];
            x[j*2+1] = x[i*2+1];
            x[i*2+1] = itemp;
         }
      }
   }
```

The following C code is used to perform the final scaling of the IFFT:

```
/* divide each element of x by n */
divide(float* x, int n)
{
   int i;
   float inv = 1.0 / n;
   for(i=0; i < n; i++)
   {
     x[2*i]   = inv * x[2*i];
     x[2*i+1] = inv * x[2*i+1];
   }
}
```

Special Requirements

- Both input x and coefficient w should be aligned on double-word boundary.
- The x value should be padded with 4 words.
- The value of n should be greater than 8.

Implementation Notes

- Loading input x as well as coefficient w in double word.
- MPY was used to perform an MV. EX. mpy x, 1, y <=> mv x, y
- Because the data loads are 1 iteration ahead of the coefficient loads, counter i was copied so that the actual count could live longer for the coefficient loads.
- Two inner loops are collapsed into one loop.
- Prolog and epilog are done in parallel with the outermost loop.
- Since the twiddle table is in bit-reversed order, it turns out that the same twiddle table will also work for smaller IFFTs. This means that if you need to do both 512 and 1024 point IFFTs in the same application, you only need to have the 1024 point twiddle table. The 512 point FFT will use the first half of the 1024 point twiddle table.

- The bit-reversed twiddle factor array w can be generated by using the gen_twiddle function provided in the support\fft directory or by running tw_r2fft.exe provided in bin\. The twiddle factor array can also be generated using the gen_w_r2 and bit_rev algorithms, as described above.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	2*n*log2(n) + 37 e.g., IF n = 64, cycles = 805 e.g., IF n = 128, cycles = 1829
Code size (in bytes)	1600

4.1.4 Filtering and Convolution

DSPF_sp_fir_cplx *Single-precision complex finite impulse response filter*

Function

void DSPF_sp_fir_cplx (const float * restrict x, const float * restrict h, float * restrict r, int nh, int nr)

Arguments

x[2*(nr+nh−1)]	Pointer to complex input array. The input data pointer x must point to the (nh)th complex element; i.e., element 2*(nh−1).
h[2*nh]	Pointer to complex coefficient array (in normal order).
r[2*nr]	Pointer to complex output array.
nh	Number of complex coefficients in vector h.
nr	Number of complex output samples to calculate.

Description

This function implements the FIR filter for complex input data. The filter has nr output samples and nh coefficients. Each array consists of an even and odd term with even terms representing the real part and the odd terms the imaginary part of the element. The coefficients are expected in normal order.

Algorithm

This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_sp_fir_cplx(const float * x, const float * h,
                  float * restrict r, int nh, int nr)
{
     int i,j;
     float imag, real;
     for (i = 0; i < 2*nr; i += 2)
     {
          imag = 0;
          real = 0;
          for (j = 0; j < 2*nh; j += 2)
          {
               real += h[j] * x[i−j] − h[j+1] * x[i+1−j];
               imag += h[j] * x[i+1−j] + h[j+1] * x[i−j];
          }
          r[i] = real;
          r[i+1] = imag;
     }
}
```

Special Requirements

- The value of nr is a multiple of 2 and greater than or equal to 2.
- The value of nh is greater than or equal to 5.
- The x and h arrays are double-word aligned.
- The x array points to 2*(nh−1)th input element.

Implementation Notes

- The outer loop is unrolled twice.
- Outer loop instructions are executed in parallel with inner loop.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	2 * nh * nr + 33 For nh=24 and nr=64, cycles=3105 For nx=32 and nr=64, cycles=4129
Code size (in bytes)	640

DSPF_sp_fir_gen *Single-precision generic FIR filter*

Function void DSPF_sp_fir_gen (const float *x, const float *h, float * restrict r, int nh, int nr)

Arguments

x	Pointer to array holding the input floating-point array.
h	Pointer to array holding the coefficient floating-point array.
r	Pointer to output array
nh	Number of coefficients.
nr	Number of output values.

Description This routine implements a block FIR filter. There are nh filter coefficients, nr output samples, and nh+nr−1 input samples. The coefficients need to be placed in the h array in reverse order {h(nh−1), ... , h(1), h(0)} and the array x starts at x(−nh+1) and ends at x(nr−1). The routine calculates y(0) through y(nr−1) using the following formula:

r(n) = h(0)*x(n) + h(1)*x(n−1) + ... + h(nh−1)*x(n−nh+1)
where n = {0, 1, ... , nr−1}.

Algorithm This is the C equivalent for the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_sp_fir_gen(const float *x, const float *h,
            float * restrict r,
            int nh, int nr)
{
   int i, j;
   float sum;
   for(i=0; i < nr; i++)
   {
      sum = 0;
      for(j=0; j < nh; j++)
      {
          sum += x[i+j] * h[i];
      }
      r[j] = sum;
   }
}
```

Special Requirements

- The x and h arrays are double−word aligned.
- Little endianness is assumed for LDDW instructions.
- The number of coefficients must be greater than or equal to 4.
- The number of outputs must be greater than or equal to 4

Implementation Notes

- LDDW instructions are used to load two SP floating-point values simultaneously for the x and h arrays.
- The outer loop is unrolled four times.
- The inner loop is unrolled two times and software pipelined.
- The variables prod1, prod3, prod5, and prod7 share A9.
  The variables prod0, prod2, prod4, and prod6 share B6.
  The variables sum1, sum3, sum5, and sum7 share A7.
  The variables sum0, sum2, sum4, and sum6 share B7.
  This multiple assignment is possible since the variables are always read just once on the first cycle that they are available.
- The first eight cycles of the inner loop prolog are conditionally scheduled in parallel with the outer loop. This increases the code size by 14 words, but improves the cycle time.
- A load counter is used so that an epilog is not needed. No extraneous loads are performed.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is *intended* to be interrupt-tolerant but not interruptible. However, a bug in the assembly code for Rev 2.0 and earlier of the library causes this function to *not* be interrupt tolerant. Therefore, in order to safely use this function you must disable interrupts prior to the call and then restore interrupts after.

Benchmarks

Cycles	(4*floor((nh−1)/2)+14)*(ceil(nr/4)) + 8 e.g., nh=10, nr=100, cycles=758 cycles
Code size (in bytes)	640

DSPF_sp_fir_r2 *Single-precision complex finite impulse response filter*

Function void DSPF_sp_fir_r2 (const float * restrict x, const float * restrict h, float * restrict r, int nh, int nr)

Arguments

x[nr+nh−1]	Pointer to input array of size nr+nh−1.
h[nh]	Pointer to coefficient array of size nh (in reverse order).
r[nr]	Pointer to output array of size nr.
nh	Number of coefficients.
nr	Number of output samples.

Description Computes a real FIR filter (direct-form) using coefficients stored in vector h[]. The real data input is stored in vector x[]. The filter output result is stored in vector r[]. The filter calculates nr output samples using nh coefficients. The coefficients are expected to be in reverse order.

Algorithm This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_sp_fir_r2(const float * x, const float * h,
             float *restrict r, int nh, int nr)
{
    int i, j;
    float sum;
    for (j = 0; j < nr; j++)
    {
        sum = 0;
        for (i = 0; i < nh; i++)
            sum += x[i + j] * h[i];
    r[j] = sum;
    }
}
```

Special Requirements

- The value of nr is a multiple of 2 and greater than or equal to 2.
- The value of nh is a multiple of 2 and greater than or equal to 8.
- The x and h arrays are double-word aligned.
- Coefficients in array h are expected to be in reverse order.
- The x and h arrays should be padded with 4 words at the end.

Implementation Notes

- The outer loop is unrolled four times and inner loop is unrolled twice.
- Outer loop instructions are executed in parallel with inner loop.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is *intended* to be interrupt-tolerant but not interruptible. However, a bug in the assembly code for Rev 2.0 and earlier of the library causes this function to *not* be interrupt tolerant. Therefore, in order to safely use this function you must disable interrupts prior to the call and then restore interrupts after.

Benchmarks

Cycles	(nh * nr)/2 + 34, if nr multiple of 4 (nh * nr)/2 + 45, if nr not multiple of 4 For nh=24 and nr=64, cycles=802 For nh=30 and nr=50, cycles=795
Code size (in bytes)	960

DSPF_sp_fircirc *Single-precision circular FIR algorithm*

Function

void DSPF_sp_fircirc (float *x, float *h, float *r, int index, int csize, int nh, int nr)

Arguments

x[]	Input array (circular buffer of 2^(csize+1) bytes). Must be aligned at 2^(csize+1) byte boundary.
h[nh]	Filter coefficients array. Must be double-word aligned.
r[nr]	Output array
index	Offset by which to start reading from the input array. Must be a multiple of 2.
csize	Size of circular buffer x[] is 2^(csize+1) bytes. Must be 2 ≤ csize ≤ 31.
nh	Number of filter coefficients. Must be a multiple of 2 and ≥ 4.
nr	Size of output array. Must be a multiple of 4.

Description

This routine implements a circularly addressed FIR filter. nh is the number of filter coefficients. nr is the number of the output samples.

Algorithm

This is the C equivalent for the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_sp_fircirc  (float x[], float h[], float r[],
            int index, int csize, int nh, int nr)
{
    int i, j;
    //Circular Buffer block size = ((2^(csize + 1)) / 4)
    //floating point numbers
    int mod = (1 << (csize - 1));
    float r0;
    for (i = 0; i < nr; i++)
    {
        r0 = 0;
        for (j = 0; j < nh; j++)
        {
            //Operation "% mod" is equivalent to "& (mod -1)"
            //r0 += x[(i + j + index) % mod] * h[j];
            r0 += x[(i + j + index) & (mod - 1)] * h[j];
        }
        r[i] = r0;
    }
}
```

Special Requirements

- ❑ The circular input buffer x[] must be aligned at a 2^(csize+1) byte boundary. csize must lie in the range 2 ≤ csize ≤ 31.
- ❑ The number of coefficients (nh) must be a multiple of 2 and greater than or equal to 4.
- ❑ The number of outputs (nr) must be a multiple of 4 and greater than or equal to 4.
- ❑ The index (offset to start reading input array) must be a multiple of 2 and less than or equal to (2^(csize−1) − 6).
- ❑ The coefficient array is assured to be in reverse order; that is, h(nh−1) to h(0) hold coefficients h0, h1, h2, etc.

Implementation Notes

- ❑ LDDW instructions are used to load two SP floating-point values simultaneously for the x and h arrays.

- The outer loop is unrolled four times.
- The inner loop is unrolled two times.
- The variables prod1, prod3, prod5 and prod7 share A9.
  The variables prod0, prod2, prod4 and prod6 share B6.
  The variables sum1, sum3, sum5 and sum7 share A7.
  The variables sum0, sum2, sum4 and sum6 share B8.
  This multiple assignment is possible since the variables are always read just once on the first cycle that they are available.
- A load counter is used so that an epilog is not needed. No extraneous loads are performed.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	(2*nh + 10) nr/4 + 18 For nh = 30 & nr=100, cycles = 1768
Code size (in bytes)	512

DSPF_sp_biquad *Single-precision 2nd order IIR (biquad) filter*

Function void DSPF_sp_biquad (float *x, float *b, float *a, float *delay, float *r, int nx)

Arguments

x	Pointer to input samples.
b	Pointer to nr coefs b0, b1, b2.
a	Pointer to dr coefs a1, a2.
delay	Pointer to filter delays.
r	Pointer to output samples.
nx	Number of input/output samples.

Description This routine implements a DF 2 transposed structure of the biquad filter. The transfer function of a biquad can be written as:

$$H(Z) = \frac{b(0) + b(1)z \wedge (-1) + b(2)z \wedge (-2)}{1 + a(1)z \wedge (-1) + a(2)z \wedge (-2)}$$

Algorithm

```
void DSPF_sp_biquad(float *x, float *b, float *a,
         float *delay, int nx)
{
    int i;
    float a1, a2, b0, b1, b2, d0, d1, x_i;
    a1 = a[0];
    a2 = a[1];
    b0 = b[0];
    b1 = b[1];
    b2 = b[2];
    d0 = delay[0];
    d1 = delay[1];
    for (i = 0; i < nx; i++)
    {
        x_i = x[i];
        r[i] = b0 * x_i + d0;
        d0 = b1 * x_i - a1 * r[i] + d1;
        d1 = b2 * x_i - a2 * r[i];
    }
    delay[0] = d0;
    delay[1] = d1;
}
```

Special Requirements

- The coefficient pointers are double-word aligned.
- The value of nx should be a multiple of 3.

Implementation Notes

- Unrolling the loop three times implies that the order of the filter has been increased by 2. This is because the output at time instant n is dependent on the outputs at instants n−3 and n−4. This is mathematically equivalent to multiplying the transfer function's numerator and denominator by (1 + k1 * z^−1) (1 + k2 * z^−1), where k1 is a1 and k2 is (a2 − a1 * a1) / a1. Hence, two new poles are introduced: one at −a1 and the other at −(a2 − a1 * a1) / a1.Thus, use of this function requires that modulii of a1 and (a2 − a1 * a1) / a1 be less than 1.

- Register sharing has been used to optimize on the use of registers.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	4 * nx + 76 For nx = 60, cycles = 316 For nx = 90, cycles = 436
Code size (in bytes)	1312

DSPF_sp_iir

Single-precision IIR filter (used in the VSELP vocoder)

Function

void DSPF_sp_iir (float* restrict r1, const float* x, float* restrict r2, const float* h2, const float* h1, int nr)

Arguments

r1[nr+4]	Delay element values (i/p and o/p).
x[nr + 4]	Pointer to the input array.
r2[nr]	Pointer to the output array.
h1[5]	Moving average filter coefficients.
h2[5]	Auto-regressive filter coefficients.
nr	Number of output samples.

Description

The IIR performs an auto-regressive moving-average (ARMA) filter with 4 auto-regressive filter coefficients and 5 moving-average filter coefficients for nr output samples. The output vector is stored in two locations. This routine is used as a high pass filter in the VSELP vocoder. The 4 values in the r1 vector store the initial values of the delays.

Algorithm

This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_sp_iir (float* restrict r1,
      const float*    x,
      float* restrict r2,
      const float*    h2,
      const float*    h1,
      int nr
      )
{
```

```
    int i, j;
    float sum;
    for (i = 0; i < nr; i++)
    {
       sum = h2[0] * x[4+i];
       for (j = 1; j <= 4; j++)
          sum += h2[j] * x[4+i-j] - h1[j] * r1[4+i-j];
       r1[4+i] = sum;
       r2[i] = r1[4+i];
    }
}
```

Special Requirements

- The value of nr must be a multiple of 2.
- Extraneous loads are allowed in the program.
- Due to unrolling modulus(h1[1]) < 1 must be true.

Implementation Notes

- The inner loop is completely unrolled so that two loops become one loop.
- The outer loop is unrolled twice to break the dependency bound of 8 cycles.
- The values of the r1 array are not loaded each time to calculate the value of the sum variable. Instead, the 4 values of the r1 array required are maintained in registers so that memory operations are significantly reduced.
- Unrolling by 2 implies calculation of constants before the start of the loop. Due to shortage of registers these constants are stored in the stack and later retrieved each time they are required.
- The stack must be placed in L2 to reduce overhead due to external memory access stalls.
- ***Endianness:*** The code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	6 * nr + 59 e.g., for nr = 64, cycles = 443
Code size (in bytes)	1152

DSPF_sp_iirlat *Single-precision all-pole IIR lattice filter*

Function void DSPF_sp_iirlat (float *x, int nx, const float * restrict k, int nk, float * restrict b, float * r)

Arguments

x[nx]	Input vector.
nx	Length of input vector.
k[nk]	Reflection coefficients.
nk	Number of reflection coefficients/lattice stages. Must be a multiple of 2 and ≥ 6.
b[nk+1]	Delay line elements from previous call. Should be initialized to all zeros prior to the first call.
r[nx]	Output vector

Description This routine implements a real all-pole IIR filter in lattice structure (AR lattice). The filter consists of nk lattice stages. Each stage requires one reflection coefficient k and one delay element b. The routine takes an input vector x[] and returns the filter output in r[]. Prior to the first call of the routine the delay elements in b[] should be set to zero. The input data may have to be pre-scaled to avoid overflow or achieve better SNR. The reflections coefficients lie in the range −1.0 < k < 1.0. The order of the coefficients is such that k[nk−1] corresponds to the first lattice stage after the input and k[0] corresponds to the last stage.

Algorithm

```
void DSPF_sp_iirlat(float * x, int nx,
            const float * restrict k, int nk,
            float * restrict b, float * r)
{
   float rt;     // output       //
   int i, j;
   for (j = 0; j < nx; j++)
   {
      rt = x[j];
      for (i = nk − 1; i >= 0; i−−)
      {
          rt = rt − b[i] * k[i];
          b[i + 1] = b[i] + rt * k[i];
      }
```

```
        b[0] = rt;
        r[j] = rt;
    }
}
```

Special Requirements

- The value of nk is a multiple of 2 and ≥ 6.
- Extraneous loads are allowed (80 bytes) before the start of array.
- The arrays k and b are double-word aligned.

Implementation Notes

- The loop has been unrolled by four times.
- Register sharing has been used to optimize on the use of registers.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	(6*floor((nk+1)/4) + 29)* nx + 25 For nk = 10, nx = 100 cycles = 4125
Code size (in bytes)	1024

DSPF_sp_convol *Single-precision convolution*

Function void DSPF_sp_convol (float *x, float *h, float *r, int nh, int nr)

Arguments

x	Pointer to real input vector of size = nr+nh−1 a typically contains input data (x) padded with consecutive nh − 1 zeros at the beginning and end.
h	pointer to real input vector of size nh in forward order. h typically contains the filter coefs.
r	Pointer to real output vector of size nr.
nh	Number of elements in vector b. Note: nh ≤ nr nh is typically noted as m in convol formulas. nh must be a multiple of 2.
nr	Number of elements in vector r. nr must be a multiple of 4.

Description

This function calculates the full-length convolution of real vectors x and h using time-domain techniques. The result is placed in real vector r. It is assumed that input vector x is padded with nh−1 no of zeros in the beginning and end. It is assumed that the length of the input vector h, nh, is a multiple of 2 and the length of the output vector r, nr, is a multiple of 4. nh is greater than or equal to 4 and nr is greater than or equal to nh. The routine computes 4 output samples at a time.

Algorithm

This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_sp_convol(float *x, float *h, float *r, short nh,
         short nr)
{
    short   ocntr, icntr;
    float   acc ;
    for (ocntr = nr ; ocntr > 0 ; ocntr--)
    {
        acc = 0 ;
        for (icntr = nh ; icntr > 0 ; icntr--)
        {
           acc += x[nr-ocntr+nh-icntr]*h[(icntr-1)];
        }
        r[nr-ocntr] = acc;
    }
}
```

Special Requirements

- The value of nh is a multiple of 2 and greater than or equal to 4.
- The value of nr is a multiple of 4.
- The x and h arrays are assumed to be aligned on a double-word boundary.

Implementation Notes

- The inner loop is unrolled twice and the outer loop is unrolled four times.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	(nh/2)*nr + (nr/2)*5 + 9 For nh=24 and nr=64, cycles=937 For nh=20 and nr=32, cycles=409
Code size (in bytes)	480

4.1.5 Math

DSPF_sp_dotp_sqr *Single-precision dot product and sum of square*

Function float DSPF_sp_dotp_sqr (float G, const float * x, const float * y, float * restrict r, int nx)

Arguments

G	Sum of y-squared initial value.
x[nx]	Pointer to first input array.
y[nx]	Pointer to second input array.
r	Pointer to output for accumulation of x[]*y[].
nx	Length of input vectors.

Description This routine computes the dot product of x[] and y[] arrays,adding it to the value in the location pointed to by r. Additionally, it computes the sum of the squares of the terms in the y array, adding it to the argument G. The final value of G is given as the return value of the function.

Algorithm This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
float DSPF_sp_dotp_sqr(float G, const float * x,
          const float * y, float *restrict r, int nx)
{
   int i;
   for (i = 0; i < nx; i++)
   {
         *r += x[i] * y[i];      /* Compute Dot Product */
         G += y[i] * y[i];       /* Compute Square  */
   }
   return G;
}
```

Special Requirements There are no special alignment requirements.

Implementation Notes

- Multiple assignment was used to reduce loop carry path.
- ***Endianness:*** This code is endian neutral.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	nx + 23 For nx=64, cycles=87. For nx=30, cycles=53
Code size (in bytes)	288

DSPF_sp_dotprod *Dot product of 2 single-precision float vectors*

Function float DSPF_sp_dotprod (const float *x, const float *y, const int nx)

Arguments

x	Pointer to array holding the first floating-point vector.
y	Pointer to array holding the second floating-point vector.
nx	Number of values in the x and y vectors.

Description This routine calculates the dot product of 2 single-precision float vectors.

Algorithm This is the C equivalent for the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
float DSPF_sp_dotprod(const float *x, const float *y,
          const int nx)
{
   int i;
   float sum = 0;
   for (i=0; i < nx; i++)
   {
      sum += x[i] * y[i];
   }
   return sum;
}
```

Special Requirements

- The x and y arrays must be double-word aligned.
- A memory pad of 4 bytes is required at the end of each array if the number of inputs is odd.
- The value of nx must be > 0.

Implementation Notes

- LDDW instructions are used to load two SP floating-point values at a time for the x and y arrays.
- The loop is unrolled once and software pipelined. However, by conditionally adding to the dot product odd numbered array sizes are also permitted.
- Since the ADDSP and MPYSP instructions take 4 cycles, A8, B8, A0, and B0 multiplex different variables to save on register usage. This multiple assignment is possible since the variables are always read just once on the first cycle that they are available.
- The loop is primed to reduce the prolog by 4 cycles (14 words) with no increase in cycle time.
- The load counter is used as the loop counter which requires a 3-cycle (6 word) epilog to finish the calculations. This does not increase the cycle time.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is *intended* to be interrupt-tolerant but not interruptible. However, a bug in the assembly code for Rev 2.0 and earlier of the library causes this function to *not* be interrupt tolerant. Therefore, in order to safely use this function you must disable interrupts prior to the call and then restore interrupts after.

Benchmarks

Cycles	nx/2 + 25 e.g., for nx = 512, cycles = 281
Code size (in bytes)	256

DSPF_sp_dotp_cplx *Complex single-precision floating-point dot product*

Function

void DSPF_sp_dotp_cplx (const float *x, const float *y, int n, float *restrict re, float * restrict im)

Arguments

x	Pointer to array holding the first floating-point vector.
y	Pointer to array holding the second floating-point vector.

n	Number of values in the x and y vectors.
re	Pointer to the location storing the real part of the result.
im	Pointer to the location storing the imaginary part of the result.

Description This routine calculates the dot product of 2 single-precision complex float vectors. The even numbered locations hold the real parts of the complex numbers while the odd numbered locations contain the imaginary portions.

Algorithm This is the C equivalent for the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_sp_dotp_cplx(const float* x, const float* y, int n,
         float* restrict re, float* restrict im)
{
    float real=0, imag=0;
    int i=0;
    for(i=0; i<n; i++)
    {
        real+=(x[2*i]*y[2*i] – x[2*i+1]*y[2*i+1]);
        imag+=(x[2*i]*y[2*i+1] + x[2*i+1]*y[2*i]);
    }
    *re=real;
    *im=imag;
}
```

Special Requirements

- Since single assignment of registers is not used, interrupts should be disabled before this function is called.
- Loop counter must be even and > 0.
- The x and y arrays must be double-word aligned.

Implementation Notes

- LDDW instructions are used to load two SP floating-point values at a time for the x and y arrays.
- A load counter avoids all extraneous loads.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	2*N + 22 e.g., for N = 512, cycles = 1046
Code size (in bytes)	352

DSPF_sp_maxval *Maximum element of single-precision vector*

Function float DSPF_sp_maxval (const float* x, int nx)

Arguments

x	Pointer to input array.
nx	Number of inputs in the input array.

Description This routine finds out the maximum number in the input array. This code returns the maximum value in the array.

Algorithm This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
float DSPF_sp_maxval(const float* x, int nx)
{
   int i,index;
   float max;
   *((int *)&max) = 0xff800000;
   for (i = 0; i < nx; i++)
   if (x[i] > max)
   {
      max = x[i];
      index = i;
   }
   return max;
}
```

Special Requirements

- The value of nx should be a multiple of 2 and ≥ 2.
- The x array should be double-word aligned.

Implementation Notes

- The loop is unrolled six times.

- Six maximums are maintained in each iteration.
- One of the maximums is calculated using SUBSP in place of CMPGTSP.
- NAN (not a number in single-precision format) in the input are disregarded.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is *intended* to be interrupt-tolerant but not interruptible. However, a bug in the assembly code for Rev 2.0 and earlier of the library causes this function to *not* be interrupt tolerant. Therefore, in order to safely use this function you must disable interrupts prior to the call and then restore interrupts after.

Benchmarks

Cycles	3*ceil(nx/6) + 35 For nx=60, cycles=65 For nx=34, cycles=53
Code size (in bytes)	448

DSPF_sp_maxidx *Index of maximum element of single-precision vector*

Function int DSPF_sp_maxidx (const float* x, int nx)

Arguments

x	Pointer to input array.
nx	Number of inputs in the input array.

Description This routine finds out the index of maximum number in the input array. This function returns the index of the greatest value.

Algorithm

```
int DSPF_sp_maxidx(const float* x, int nx)
{
   int index, i;
   float max;
   *((int *)&max) = 0xff800000;
   for (i = 0; i < nx; i++)
```

```
    if (x[i] > max)
    {
       max = x[i];
       index = i;
    }
    return index;
}
```

Special Requirements

- The value of nx is a multiple of 3.
- The value is nx ≥ 3, and nx ≤ 2^16−1.

Implementation Notes

- The loop is unrolled three times.
- Three maximums are maintained in each iteration.
- MPY instructions are used for move.
- ***Endianness:*** This code is endian neutral.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	2*nx/3 + 13 For nx=60, cycles=53 For nx=30, cycles=33
Code size (in bytes)	256

DSPF_sp_minval *Minimum element of single-precision vector*

Function float DSPF_sp_minval (const float* x, int nx)

Arguments

x	Pointer to input array.
nx	Number of inputs in the input array.

Description This routine finds out and returns the minimum number in the input array.

Algorithm

```
float DSPF_sp_minval(const float* x, int nx)
{
   int i,index;
   float min;
   *((int *)&min) = 0x7f800000;
   for (i = 0; i < nx; i++)
   if (x[i] < min)
   {
      min = x[i];
      index = i;
   }
   return min;
}
```

Special Requirements

- The value of nx should be a multiple of 2 and ≥ 2.
- The x array should be double-word aligned.
- NAN (not a number in single-precision format) in the input are disregarded.

Implementation Notes

- The loop is unrolled six times.
- Six minimums are maintained in each iteration. One of the minimums is calculated using SUBSP in place of CMPGTSP
- NAN (not a number in single-precision format) in the input are disregarded.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is *intended* to be interrupt-tolerant but not interruptible. However, a bug in the assembly code for Rev 2.0 and earlier of the library causes this function to *not* be interrupt tolerant. Therefore, in order to safely use this function you must disable interrupts prior to the call and then restore interrupts after.

Benchmarks

Cycles	3*ceil(nx/6) + 35 For nx=60 cycles=65 For nx=34 cycles=53
Code size (in bytes)	448

DSPF_sp_vecrecip *Single-precision vector reciprocal*

Function void DSPF_sp_vecrecip (const float *x, float * restrict r, int n)

Arguments

x	Pointer to input array.
r	Pointer to output array.
n	Number of elements in array.

Description The sp_vecrecip module calculates the reciprocal of each element in the array x and returns the output in array r. It uses 2 iterations of the Newton-Raphson method to improve the accuracy of the output generated by the RCPSP instruction of the C67x. Each iteration doubles the accuracy. The initial output generated by RCPSP is 8 bits. So after the first iteration it is 16 bits and after the second it is the full 23 bits. The formula used is:

r[n+1] = r[n](2 − v*r[n])
where v = the number whose reciprocal is to be found.
x[0], the seed value for the algorithm, is given by RCPSP.

Algorithm This is the C equivalent of the assembly code without restrictions.

```
void DSPF_sp_vecrecip(const float* x, float* restrict r,
            int n)
{
   int i;
   for(i = 0; i < n; i++)
      r[i] = 1 / x[i];
}
```

Special Requirements There are no alignment requirements.

Implementation Notes

- The inner loop is unrolled four times to allow calculation of four reciprocals in the kernel. However, the stores are executed conditionally to allow n to be any number > 0.
- Register sharing is used to make optimal use of available registers.
- No extraneous loads occur except for the case when n ≤ 4 where a pad of 16 bytes is required.
- ***Endianness:*** This code is little endian.

- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	8*floor((n−1)/4) + 53 e.g., for n = 100, cycles = 245
Code size (in bytes)	512

DSPF_sp_vecsum_sq *Single-precision sum of squares*

Function float DSPF_sp_vecsum_sq (const float *x, int n)

Arguments

x	Pointer to first input array.
n	Number of elements in arrays.

Description This routine performs a sum of squares of the elements of the array x and returns the sum.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
float DSPF_sp_vecsum_sq(const float *x,int n)
{
    int i;
    float sum=0;
    for(i = 0;  i < n; i++ )
        sum += x[i]*x[i];
    return sum;
}
```

Special Requirements

- The x array must be double-word aligned.
- Since loads of 8 floats beyond the array occur, a pad must be provided.

Implementation Notes

- The inner loop is unrolled twice. Hence, two registers are used to hold the sum of squares. ADDSPs are staggered.

- ***Endianness:*** This code is endian neutral.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	floor((n−1)/2) + 26 e.g., for n = 200, cycles = 125
Code size (in bytes)	384

DSPF_sp_w_vec *Single-precision weighted sum of vectors*

Function void DSPF_sp_w_vec (const float* x, const float * y, float m, float * restrict r, int nr)

Arguments

x	Pointer to first input array.
y	Pointer to second input array.
m	Weight factor.
r	Output array pointer.
nr	Number of elements in arrays.

Description This routine is used to obtain the weighted vector sum. Both the inputs and output are single-precision floating-point numbers.

Algorithm This is the C equivalent of the assembly code without restrictions.

```
void DSPF_sp_w_vec( const float * x,const float * y, float m
            float * restrict r,int nr)
{
   int i;
   for (i = 0; i < nr; i++)
       r[i] = (m * x[i]) + y[i];
}
```

Special Requirements

- The x and y arrays must be double-word aligned.
- The value of nr must be > 0.

Implementation Notes

- The inner loop is unrolled twice.
- No extraneous loads occur except for odd values of n.
- Write buffer fulls occur unless the array r is in cache.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	2*floor((n−1)/2) + 19 e.g., for n = 200, cycles = 219
Code size (in bytes)	384

DSPF_sp_vecmul *Single-precision vector multiplication*

Function void DSPF_sp_vecmul (const float *x, const float *y, float * restrict r, int n)

Arguments

x	Pointer to first input array.
y	Pointer to second input array.
r	Pointer to output array.
n	Number of elements in arrays.

Description This routine performs an element by element floating-point multiply of the vectors x[] and y[] and returns the values in r[].

Algorithm This is the C equivalent of the assembly code without restrictions.

```
void DSPF_sp_vecmul(const float * x, const float * y,
 float * restrict r, int n)
 {
   int i;
   for(i = 0; i < n; i++)
       r[i] = x[i] * y[i];
 }
```

Special Requirements The x and y arrays must be double-word aligned.

Implementation Notes

- The inner loop is unrolled twice to allow calculation of 2 outputs in the kernel. However the stores are executed conditionally to allow n to be any number > 0.
- No extraneous loads occur except for the case when n is odd where a pad of 4 bytes is required.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	2*floor((n−1)/2) + 18 e.g., for n = 200, cycles = 216
Code size (in bytes)	192

4.1.6 Matrix

DSPF_sp_mat_mul *Single-precision matrix multiplication*

Function void DSPF_sp_mat_mul (float *x, int r1, int c1, float *y, int c2, float *r)

Arguments

x	Pointer to r1 by c1 input matrix.
r1	Number of rows in x.
c1	Number of columns in x. Also number of rows in y.
y	Pointer to c1 by c2 input matrix.
c2	Number of columns in y.
r	Pointer to r1 by c2 output matrix.

Description This function computes the expression r = x * y for the matrices x and y. The column dimension of x must match the row dimension of y. The resulting matrix has the same number of rows as x and the same number of columns as y. The values stored in the matrices are assumed to be single-precision floating-point values. This code is suitable for dense matrices. No optimizations are made for sparse matrices.

Algorithm

```
void DSPF_sp_mat_mul(float *x, int r1, int c1, float *y, int
c2, float *r)
{
   int i, j, k;
   float sum;
 //  Multiply each row in x by each column in y.
 //  The product of row m in x and column n in y is placed
 //  in position (m,n) in the result.
   for (i = 0; i < r1; i++)
       for (j = 0; j < c2; j++)
        {
           sum = 0;
            for (k = 0; k < c1; k++)
                sum += x[k + i*c1] * y[j + k*c2];
            r[j + i*c2] = sum;
        }
}
```

Special Requirements

- The x, y, and r data are stored in distinct arrays. That is, in-place processing is not allowed.
- All r1, c1, c2 are assumed to be > 1
- Five floats are always loaded extra from the locations:

  y[c1' * c2'], y[c1' * c2' + 1],
  x[r1 * c1'], x[r1' * c1'] and x[2 * c1]
  where
  r1' = r1 + (r1&1)
  c2' = c2 + (c2&1)
  c1' = c1 + 1 + 2*(c1&1)
- If (r1&1) means r1 is odd, one extra row of x[] matrix is loaded
- If (c2&1) means c2 is odd, one extra col of y[] matrix is loaded

Implementation Notes

- All three loops are unrolled two times
- All the prolog stages of the innermost loop (k loop) are collapsed

- ❑ ***Endianness:*** This code is endian neutral.
- ❑ ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	(0.5 * r1' * c1 * c2') + (6 * c2' * r1') + (4 * r1') + 22 where r1' = r1 + (r1&1) c2' = c2 + (c2&1) For r1 = 12, c1 = 14 and c2 = 18, cycles = 2890
Code size (in bytes)	992

DSPF_sp_mat_trans *Single-precision matrix transpose*

Function void DSPF_sp_mat_trans (const float *restrict x, int rows, int cols, float *restrict r)

Arguments

x	Input matrix containing rows*cols floating-point numbers.
rows	Number of rows in matrix x. Also number of columns in matrix r.
cols	Number of columns in matrix x. Also number of rows in matrix r.
r	Output matrix containing cols*rows floating-point numbers.

Description This function transposes the input matrix x[] and writes the result to matrix r[].

Algorithm This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_sp_mat_trans(const float *restrict x, int rows,
            int cols, float *restrict r)
{
   int i,j;
   for(i=0; i<cols; i++)
      for(j=0; j<rows; j++)
         r[i * rows + j] = x[i + cols * j];
}
```

Special Requirements The number of rows and columns is > 0.

Implementation Notes

- The loop is unrolled twice.
- ***Endianness:*** This code is endian neutral.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	2 * rows * cols + 7 For rows=10 and cols=20, cycles=407 For rows=15 and cols=20, cycles=607
Code size (in bytes)	128

DSPF_sp_mat_mul_cplx *Complex matrix multiplication*

Function void DSPF_sp_mat_mul_cplx (const float* x, int r1, int c1, const float* y, int c2, float* restrict r)

Arguments

x[2*r1*c1]	Input matrix containing r1*c1 complex floating-point numbers having r1 rows and c1 columns of complex numbers.
r1	Number of rows in matrix x.
c1	Number of columns in matrix x. Also number of rows in matrix y.
y[2*c1*c2]	Input matrix containing c1*c2 complex floating-point numbers having c1 rows and c2 columns of complex numbers.
c2	Number of columns in matrix y.
r[2*r1*c2]	Output matrix of c1*c2 complex floating-point numbers having c1 rows and c2 columns of complex numbers. Complex numbers are stored consecutively with real values are stored in even word positions and imaginary values in odd positions.

Description This function computes the expression “r = x * y” for the matrices x and y. The columnar dimension of x must match the row dimension of y. The resulting matrix has the same number of rows as x and the same number of columns as y. Each element of Matrices are assumed to be complex numbers with real values are stored in even word positions and imaginary values in odd positions.

Algorithm

This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_sp_mat_mul_cplx(const float* x, int r1, int c1,
  const float* y, int c2, float* restrict r)
{
    float real, imag;
    int i, j, k;
    for(i = 0; i < r1; i++)
    {
      for(j = 0; j < c2; j++)
      {
        real=0;
        imag=0;
        for(k = 0; k < c1; k++)
        {
        real += (x[i*2*c1 + 2*k]*y[k*2*c2 + 2*j]
        -x[i*2*c1 + 2*k + 1] * y[k*2*c2 + 2*j + 1]);
        imag+=(x[i*2*c1 + 2*k] * y[k*2*c2 + 2*j + 1]
           + x[i*2*c1 + 2*k + 1] * y[k*2*c2 + 2*j]);
        }
        r[i*2*c2 + 2*j] = real;
        r[i*2*c2 + 2*j + 1] = imag;
      }
    }
}
```

Special Requirements

- Use values $c1 \geq 4$, and $r1, r2 \geq 1$
- The x array should be padded with 6 words.
- The x and y arrays should be double-word aligned.

Implementation Notes

- Innermost loop is unrolled twice.
- Two inner loops are collapsed into one loop.
- Outermost loop is executed in parallel with inner loops.

- ❑ Real values are stored in even word positions and imaginary values in odd positions.
- ❑ ***Endianness:*** This code is little endian.
- ❑ ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	2*r1*c1*c2' + 33 WHERE c2'=2*ceil(c2/2) When r1=3, c1=4, c2=4, cycles = 129 When r1=4, c1=4, c2=5, cycles = 225
Code size (in bytes)	800

4.1.7 Miscellaneous

DSPF_sp_blk_move *Single-precision block move*

Function void DSPF_sp_blk_move (const float * x, float *restrict r, int nx)

Arguments

x[nx]	Pointer to source data to be moved.
r[nx]	Pointer to destination array.
nx	Number of floats to move.

Description This routine moves nx floats from one memory location pointed to by x to another pointed to by r.

Algorithm This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_sp_blk_move(const float * x, float *restrict r, int
nx)
{
    int i;
    for (i = 0 ; i < nx; i++)
        r[i] = x[i];
}
```

Special Requirements

- The value of nx is greater than 0.
- If nx is odd then x and r should be padded with 1 word.
- The x and r arrays are double-word aligned.

Implementation Notes

- The loop is unrolled twice.
- Cache touching is used to remove the Write Buffer Full problem.
- ***Endianness:*** This implementation is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	2*ceil(nx/2)+7 For nx=64, cycles=71 For nx=25, cycles=33
Code size (in bytes)	128

DSPF_blk_eswap16 *Endian swap a block of 16-bit values*

Function void DSPF_blk_eswap16 (void *restrict x, void *restrict r, int nx)

Arguments

x[nx]	Pointer to source data.
r[nx]	Pointer to destination array.
nx	Number of shorts (16-bit values) to swap.

Description The data in the x[] array is endian swapped, meaning that the byte-order of the bytes within each half-word of the r[] array is reversed. This is meant to facilitate moving big-endian data to a little-endian system or vice versa. When the r pointer is non-NULL, the endian swap occurs out-of-place, similar to a block move. When the r pointer is NULL, the endian swap occurs in place, allowing the swap to occur without using any additional storage.

Algorithm This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_blk_eswap16(void *restrict x, void *restrict r,
                      int nx)
```

```
{
    int i;
    char *_src, *_dst;
    if (r)
    {
        _src = (char *)x;
        _dst = (char *)r;
    }
    else
    {
        _src = (char *)x;
        _dst = (char *)x;
    }
    for (i = 0; i < nx; i++)
    {
        char t0, t1;
        t0 = _src[i*2 + 1];
        t1 = _src[i*2 + 0];
        _dst[i*2 + 0] = t0;
        _dst[i*2 + 1] = t1;
    }
       }
```

Special Requirements

- The value of nx is greater than 0 and multiple of 8.
- The value of nx is padded with 2 words.
- The x and r arrays should be word aligned.
- Input array x and output array r do not overlap, except in the special case r==NULL so that the operation occurs in place.

Implementation Notes

- The loop is unrolled eight times.
- ***Endianness:*** This implementation is endian neutral.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	0.625 * nx + 12 For nx=64, cycles=52 For nx=32, cycles=32
Code size (in bytes)	256

DSPF_blk_eswap32 *Endian swap a block of 32-bit values*

Function void DSPF_blk_eswap32 (void *restrict x, void *restrict r, int nx)

Arguments

x[nx]	Pointer to source data.
r[nx]	Pointer to destination array.
nx	Number of floats (32-bit values) to swap.

Description The data in the x[] array is endian swapped, meaning that the byte-order of the bytes within each word of the r[] array is reversed. This is meant to facilitate moving big-endian data to a little-endian system or vice versa. When the r pointer is non-NULL, the endian swap occurs out-of-place, similar to a block move. When the r pointer is NULL, the endian swap occurs in place, allowing the swap to occur without using any additional storage.

Algorithm This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_blk_eswap32(void *restrict x, void *restrict r,
             int nx)
{
    int i;
    char *_src, *_dst;
    if (r)
    {
        _src = (char *)x;
        _dst = (char *)r;
    }
    else
    {
```

```
        _src = (char *)x;
        _dst = (char *)x;
    }
    for (i = 0; i < nx; i++)
    {
        char t0, t1, t2, t3;
        t0 = _src[i*4 + 3];
        t1 = _src[i*4 + 2];
        t2 = _src[i*4 + 1];
        t3 = _src[i*4 + 0];
        _dst[i*4 + 0] = t0;
        _dst[i*4 + 1] = t1;
        _dst[i*4 + 2] = t2;
        _dst[i*4 + 3] = t3;
    }
}
```

Special Requirements

- The value of nx is greater than 0 and multiple of 2.
- The x and r arrays should be word aligned.
- Input array x and Output array r do not overlap, except in the special case “r==NULL” so that the operation occurs in place.

Implementation Notes

- The loop is unrolled twice.
- Multiply instructions are used for shifting left and right.
- ***Endianness:*** This implementation is endian neutral.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	1.5 * nx + 14 For nx=64 cycles=110 For nx=32 cycles=62
Code size (in bytes)	224

DSPF_blk_eswap64 *Endian swap a block of 64-bit values*

Function void DSPF_blk_eswap64 (void *restrict x, void *restrict r, int nx)

Arguments

x[nx]	Pointer to source data.
r[nx]	Pointer to destination array.
nx	Number of doubles (64-bit values) to swap.

Description The data in the x[] array is endian swapped, meaning that the byte-order of the bytes within each double word of the r[] array is reversed. This is meant to facilitate moving big-endian data to a little-endian system or vice versa. When the r pointer is non-NULL, the endian swap occurs out-of-place, similar to a block move. When the r pointer is NULL, the endian swap occurs in place, allowing the swap to occur without using any additional storage.

Algorithm This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_blk_eswap64(void *restrict x, void *restrict r,
              int nx)
{
  int i;
  char *_src, *_dst;
  if (r)
  {
      _src = (char *)x;
      _dst = (char *)r;
  }
  else
  {
      _src = (char *)x;
      _dst = (char *)x;
  }
  for (i = 0; i < nx; i++)
  {
      char t0, t1, t2, t3, t4, t5, t6, t7;
      t0 = _src[i*8 + 7];
      t1 = _src[i*8 + 6];
```

```
        t2 = _src[i*8 + 5];
        t3 = _src[i*8 + 4];
        t4 = _src[i*8 + 3];
        t5 = _src[i*8 + 2];
        t6 = _src[i*8 + 1];
        t7 = _src[i*8 + 0];
        _dst[i*8 + 0] = t0;
        _dst[i*8 + 1] = t1;
        _dst[i*8 + 2] = t2;
        _dst[i*8 + 3] = t3;
        _dst[i*8 + 4] = t4;
        _dst[i*8 + 5] = t5;
        _dst[i*8 + 6] = t6;
        _dst[i*8 + 7] = t7;
    }
}
```

Special Requirements

- The value of nx is greater than 0.
- The x and r arrays should be word aligned.
- Input array x and Output array r do not overlap, except in the special case “r==NULL” so that the operation occurs in place.

Implementation Notes

- Multiply instructions are used for shifting left and right.
- ***Endianness:*** This implementation is endian neutral.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	3 * nx + 14 For nx=64, cycles=206 For nx=32, cycles=110
Code size (in bytes)	224

DSPF_fltoq15 *IEEE single-precision floating-point-to-Q15 format*

Function void DSPF_fltoq15 (const float* restrict x, short* restrict r, int nx)

Arguments

x[nx]	Input array containing values of type float.
r[nx]	Output array contains Q15 equivalents of x[nx].
nx	Number of elements in both arrays.

Description Convert the IEEE floating-point numbers stored in vector x[] into Q.15 format numbers stored in vector r[]. Results will be rounded towards negative infinity. All values that exceed the size limit will be saturated to 0x7fff if value is positive and 0x8000 if value is negative.

Algorithm This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_fltoq15
(
    const float* restrict x,
    short*       restrict r,
    int         nx
)
{
    int i, a;
    for(i = 0; i < nx; i++)
    {
        a = floor(32768 * x[i]);
        // saturate to 16-bit //
        if (a>32767)  a =  32767;
        if (a<-32768) a = -32768;
        r[i] = (short) a;
    }
}
```

Special Requirements

- ❑ No special alignment requirements.
- ❑ The value of nx must be > 0.

Implementation Notes

- SSHL has been used to saturate the output of the instruction SPINT.
- There are no write buffer fulls because one STH occurs per cycle.
- ***Endianness:*** This implementation is endian neutral.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	nx + 17 e.g., nx = 512, cycles = 529
Code size (in bytes)	384

DSPF_sp_minerr *VSELP vocoder code book search algorithm*

Function float DSPF_sp_minerr (const float* GSP0_TABLE, const float* errCoefs, int *restrict max_index)

Arguments

GSP0_TABLE[256*9]	GSP0 terms array.
errCoefs[9]	Array of error coefficients. Must be double-word aligned.
max_index	Index to GSP0_TABLE[max_index], the first element of the 9-element vector that resulted in the maximum dot product.

Description Performs a dot product on 256 pairs of 9 element vectors and searches for the pair of vectors which produces the maximum dot product result. This is a large part of the VSELP vocoder codebook search. The function stores the index to the first element of the 9-element vector that resulted in the maximum dot product in the memory location Pointed by max_index. The maximum dot product value is returned by the function.

Algorithm This is the C equivalent for the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
float DSPF_sp_minerr(const float* GSP0_TABLE,
             const float* errCoefs, int *restrict max_index)
{
```

```
    float val, maxVal = -50;
    int i, j;
    for (i = 0; i < GSP0_NUM; i++)
    {
       for (val = 0, j = 0; j < GSP0_TERMS; j++)
          val += GSP0_TABLE[i*GSP0_TERMS+j] * errCoefs[j];
       if (val > maxVal)
       {
          maxVal = val;
          *max_index = i*GSP0_TERMS;
       }
    }
    return (maxVal);
}
```

Special Requirements errCoefs must be double-word aligned.

Implementation Notes

- The inner loop is totally unrolled.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	1188
Code size (in bytes)	736

DSPF_q15tofl *Q15 format to single-precision IEEE floating-point format*

Function void DSPF_q15tofl (const short *x, float * restrict r, int nx)

Arguments

x	Input array containing shorts in Q15 format.
r	Output array containing equivalent floats.
nx	Number of values in the x vector.

Description

This routine converts data in the Q15 format into IEEE single-precision floating point.

Algorithm

This is the C equivalent for the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_q15tofl(const short *x, float * restrict r, int nx)
{
    int i;
    for (i = 0; i < nx; i++)
        r[i] = (float)x[i] / 0x8000;
}
```

Special Requirements

- The x array must be double-word aligned.
- The value of nx must be > 0.
- Extraneous loads are allowed in the program.

Implementation Notes

- LDDW instructions are used to load four short values at a time.
- The loop is unrolled once and software pipelined. However, by conditionally storing odd numbered array sizes are also permitted.
- To avoid write buffer fulls on the 671x the output array is brought into cache inside the kernel. Thus, the store happens to addresses already in L1D. Thus, no use of the write buffer is made.
- No write buffer fulls occur because of cache touching.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	3*floor((nx−1)/4) + 20 e.g., for nx = 512, cycles = 401
Code size (in bytes)	448

4.2 Double-Precision Functions

4.2.1 Adaptive Filtering

DSPF_dp_lms *Double-precision floating-point LMS algorithm*

Function

double DSPF_dp_lms (double *x, double *h, double *desired, double *r, double adapt rate, double error, int nh, int nr)

Arguments

x	Pointer to input samples.
h	Pointer to the coefficient array.
desired	Pointer to the desired output array.
r	Pointer to filtered output array.
adapt rate	Adaptation rate.
error	Initial error.
nh	Number of coefficients.
nr	Number of output samples.

Description

The dp_lms implements an LMS adaptive filter. Given an actual input signal and a desired input signal, the filter produces an output signal, the final coefficient values and returns the final output error signal.

Algorithm

This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
double DSPF_dp_lms(double *x, double *h, double *y,
   int nh, double *d, double ar, int nr, double error)
{
    int i,j;
    double sum;
    for (i = 0; i < nr; i++)
    {
     for (j = 0; j < nh; j++)
     {
        h[j] = h[j] + (ar*error*x[i+j-1]);
     }
```

```
      sum = 0.0f;
       for (j = 0; j < nh; j++)
       {
         sum += h[j] * x[i+j];
       }
      y[i] = sum;
      error = d[i] - sum;
    }
    return error;
  }
```

Special Requirements

- ❑ The inner-loop counter must be a multiple of 2 and ≥ 2.
- ❑ Little endianness is assumed.
- ❑ Extraneous loads are allowed in the program.
- ❑ The coefficient array is assumed to be in reverse order, i.e., h(nh−1) to h(0) hold coeffs. h0, h1, h2, etc.

Implementation Notes

- ❑ The inner loop is unrolled Two times to allow update of two coefficients in the kernel.
- ❑ The error term needs to be computed in the outer loop before a new iteration of the inner loop can start. As a result the prolog cannot be placed in parallel with epilog (after the loop kernel).
- ❑ Register sharing is used to make optimal use of available registers.
- ❑ ***Endianness:*** This code is little endian.
- ❑ ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	(4*nh + 47) nr + 27 e.g., for nh = 24 and nr = 36
Code size (in bytes)	640

4.2.2 Correlation

DSPF_dp_autocor *Double-precision autocorrelation*

Function void DSPF_dp_autocor (double * restrict r, const double* restrict x, int nx, int nr)

Arguments

r	Pointer to output array of autocorrelation of length nr.
x	Pointer to input array of length nx+nr. Input data must be padded with nr consecutive zeros at the beginning.
nx	Length of autocorrelation vector.
nr	Length of lags.

Description This routine performs the autocorrelation of the input array x. It is assumed that the length of the input array, x, is a multiple of 2 and the length of the output array, r, is a multiple of 4. The assembly routine computes 4 output samples at a time. It is assumed that input vector x is padded with nr no of zeros in the beginning.

Algorithm This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_dp_autocor(double * restrict r, const double
    * restrict x, int nx, int nr)
    {
        int i,k;
        double sum;
        for (i = 0; i < nr; i++)
        {
            sum = 0;
            for (k = nr; k < nx+nr; k++)
               sum += x[k] * x[k−i];
            r[i] = sum ;
        }
    }
```

Special Requirements

❑ The value nx is a multiple of 2 and greater than or equal to 4.

- The value of nr is a multiple of 4 and greater than or equal to 4.
- The value of nx is greater than or equal to nr

Implementation Notes

- The inner loop is unrolled twice and the outer loop is unrolled four times.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	2*nx*nr + 5/2*nr + 32 For nx=32 and nr=64, cycles=4258 For nx=24 and nr=32, cycles=1648
Code size (in bytes)	576

4.2.3 FFT

DSPF_dp_bitrev_cplx *Bit reversal for double-precision complex numbers*

Function void DSPF_dp_bitrev_cplx (double *x, short *index, int nx)

Arguments

x	Complex input array to be bit reversed. Contains 2*nx doubles.
index	Array of size ~sqrt(nx) created by the routine bitrev_index to allow the fast implementation of the bit reversal.
nx	Number of elements in array x[]. Must be power of 2.

Description This routine performs the bit reversal of the input array x[], where x[] is a double array of length 2*nx containing double-precision floating-point complex pairs of data. This routine requires the index array provided by the program below. This index should be generated at compile time not by the DSP. TI retains all rights, title and interest in this code and only authorizes the use of the bit-reversal code and related table-generation code with TMS320 family DSPs manufactured by TI.

```
/* ----------------------------------------------------------- */
/* This routine calculates the index for bit reversal of */
```

```
/* an array of length nx. The length of the index table is */
/* 2^(2*ceil(k/2)) where nx = 2^k. */
/* */
/* In other words, the length of the index table is: */
/* - for even power of radix: sqrt(nx) */
/* - for odd power of radix: sqrt(2*nx) */
/* ----------------------------------------------------------- */
void bitrev_index(short *index, int nx)
{
    int i, j, k, radix = 2;
    short nbits, nbot, ntop, ndiff, n2, raddiv2;
    nbits = 0;
    i = nx;
    while (i > 1)
    {
        i = i >> 1;
        nbits++;
    }
    raddiv2 = radix >> 1;
    nbot = nbits >> raddiv2;
    nbot = nbot << raddiv2 - 1;
    ndiff = nbits & raddiv2;
    ntop = nbot + ndiff;
    n2 = 1 << ntop;
    index[0] = 0;
    for ( i = 1, j = n2/radix + 1; i < n2 - 1; i++)
    {
        index[i] = j - 1;
        for (k = n2/radix; k*(radix-1) < j; k /= radix)
            j -= k*(radix-1);
        j += k;
    }
    index[n2 - 1] = n2 - 1;
}
```

Algorithm This is the C equivalent for the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
            void dp_bitrev_cplx(double* x, short* index,
int nx)
            {
               int i;
               short i0, i1, i2;
               short j0, j1, j2;
               double xi0r, xi0i, xi1r, xi1i, xi2r, xi2i;
               double xj0r, xj0i, xj1r, xj1i, xj2r, xj2i;
      short t;
               int a, b, ia, ib, ibs;
               int mask;
               int nbits, nbot, ntop, ndiff, n2, halfn;
               nbits = 0;
               i = nx;
               while (i > 1)
               {
                  i = i >> 1;
                  nbits++;
               }
               nbot = nbits >> 1;
               ndiff = nbits & 1;
               ntop = nbot + ndiff;
               n2 = 1 << ntop;
               mask = n2 - 1;
               halfn = nx >> 1;
               for (i0 = 0; i0 < halfn; i0 += 2)
               {
                   b = i0 & mask;
                   a = i0 >> nbot;
                   if (!b) ia = index[a];
                   ib = index[b];
                   ibs = ib << nbot;
                   j0 = ibs + ia;
                   t = i0 < j0;
                   xi0r = x[2*i0];
```

```
            xi0i = x[2*i0+1];
        xj0r = x[2*j0];
            xj0i = x[2*j0+1];
            if (t)
            {
                x[2*i0] = xj0r;
                x[2*i0+1] = xj0i;
x[2*j0] = xi0r;
         x[2*j0+1] = xi0i;
        }
            i1 = i0 + 1;
            j1 = j0 + halfn;
            xi1r = x[2*i1];
            xi1i = x[2*i1+1];
        xj1r = x[2*j1];
        xj1i = x[2*j1+1];
            x[2*i1] = xj1r;
            x[2*i1+1] = xj1i;
        x[2*j1] = xi1r;
            x[2*j1+1] = xi1i;
        i2 = i1 + halfn;
            j2 = j1 + 1;
        xi2r = x[2*i2];
        xi2i = x[2*i2+1];
            xj2r = x[2*j2];
            xj2i = x[2*j2+1];
            if (t)
            {
            x[2*i2] = xj2r;
            x[2*i2+1] = xj2i;
        x[2*j2] = xi2r;
            x[2*j2+1] = xi2i;
            }
          }
        }
```

Special Requirements

- The value of nx must be a power of 2.
- The table from bitrev_index is already created.
- The x array is actually an array of 2*nx doubles.

Implementation Notes

- The index table can be generated using the bitrev_index function provided in the dsplib\support\fft directory.
- If nx ≤ 4K one can use the char (8-bit) data type for the “index” variable. This would require changing the LDH when loading index values in the assembly routine to LDB. This would further reduce the size of the Index Table by half its size.
- ***Endianness:*** Little endian configuration used.
- ***Interruptibility:*** This code is interrupt-tolerant, but not interruptible.

Benchmarks

Cycles	5*nx + 33 e.g., nx = 128, cycles = 673
Code size (in bytes)	736

DSPF_dp_cfftr4_dif *Double-precision floating-point decimation in frequency radix-4 FFT with complex input*

Function void DSPF_dp_cfftr4_dif (double* x, double* w, short n)

Arguments

x	Pointer to an array holding the input and output floating-point array which contains n complex points.
w	Pointer to an array holding the coefficient floating-point array which contains 3*n/4 complex numbers.
n	Number of complex points in x.

Description This routine implements the DIF (decimation in frequency) complex radix 4 FFT with digit-reversed output and normal order input. The number of points, n, must be a power of 4 {4, 16, 64, 256, 1024, ...}. This routine is an in-place routine in the sense that the output is written over the input. It is not an in-place routine in the sense that the input is in normal order and the output is in digit-reversed order.

There must be n complex points (2*n values), and 3*n/4 complex coefficients (3*n/2 values). Each real and imaginary input value is interleaved in the x array {rx0, ix0, rx1, ix2, ...} and the complex numbers are in normal order. Each real and imaginary output value is interleaved in the x array and the complex numbers are in digit-reversed order {rx0, ix0, ...}. The real and imaginary values of the coefficients are interleaved in the w array {rw0, -iw0, rw1, -iw1, ...} and the complex numbers are in normal order.

Note that the imaginary coefficients are negated.

{cos(d*0), sin(d*0), cos(d*1), sin(d*1), ...} rather than
{cos(d*0), −sin(d*0), cos(d*1), −sin(d*1), ...}

where d = 2*PI/n. The value of w(n,k) is usually written w(n,k) = e^−j(2*PI*k/n) = cos(2*PI*k/n) − sin(2*PI*k/n).

The routine can be used to implement an inverse FFT by performing the complex conjugate on the input complex numbers (negating the imaginary value), and dividing the result by n.

Another method to use the FFT to perform an inverse FFT, is to swap the real and imaginary values of the input and the result and divide the result by n. In either case, the input is still in normal order and the output is still in digit-reversed order.

Note that you can not make the radix 4 FFT into an inverse FFT by using the complex conjugate of the coefficients as you can do with the complex radix 2 FFT.

If you label the input locations from 0 to (n-1) (normal order), the digit-reversed locations can be calculated by reversing the order of the bit pairs of the labels. For example, for a 1024 point FFT, the digit reversed location for
617d = 1001101001b = 10 01 10 10 01 is
422d = 0110100110b = 01 10 10 01 10 and vice versa.

Algorithm

This is the C equivalent for the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_dp_cfftr4_dif(double* x, double* w, short n)
{
    short n1, n2, ie, ia1, ia2, ia3, i0, i1, i2, i3, j, k;
    double r1, r2, r3, r4, s1, s2, s3, s4, co1, co2, co3;
    double si1, si2, si3;
n2 = n;
    ie = 1;
```

```
for(k=n; k>1; k>>=2)
{
   n1 = n2;
   n2 >>= 2;
   ia1 = 0;
   for(j=0; j<n2; j++)
   {
       ia2 = ia1 + ia1;
       ia3 = ia1 + ia2;
       co1 = w[ia1*2];
       si1 = w[ia1*2 + 1];
       co2 = w[ia2*2];
       si2 = w[ia2*2 + 1];
       co3 = w[ia3*2];
       si3 = w[ia3*2 + 1];
       ia1 += ie;
       for(i0=j; i0<n; i0+=n1)
       {
           i1 = i0 + n2;
           i2 = i1 + n2;
           i3 = i2 + n2;
           r1 = x[i0*2]   + x[i2*2];
           r3 = x[i0*2]   - x[i2*2];
           s1 = x[i0*2+1] + x[i2*2+1];
           s3 = x[i0*2+1] - x[i2*2+1];
           r2 = x[i1*2]   + x[i3*2];
           r4 = x[i1*2]   - x[i3*2];
           s2 = x[i1*2+1] + x[i3*2+1];
           s4 = x[i1*2+1] - x[i3*2+1];
           x[i0*2]   = r1 + r2;
           r2        = r1 - r2;
           r1        = r3 - s4;
           r3        = r3 + s4;
           x[i0*2+1] = s1 + s2;
           s2        = s1 - s2;
```

```
                    s1          = s3 + r4;
                    s3          = s3 - r4;
                    x[i1*2]     = co1*r3 + si1*s3;
                    x[i1*2+1] = co1*s3 - si1*r3;
                    x[i2*2]     = co2*r2 + si2*s2;
                    x[i2*2+1] = co2*s2 - si2*r2;
                    x[i3*2]     = co3*r1 + si3*s1;
                    x[i3*2+1] = co3*s1 - si3*r1;
               }
           }
           ie <<= 2;
       }
}
```

Special Requirements There are no special alignment requirements.

Implementation Notes

- All the three loops are executed as one loop with conditional instructions.
- The outer-loop counter is used as load counter to prevent extraneous loads
- If more registers were available, the inner loop could probably be as small as 28 cycles The loop was extended to 56 cycles to allow more variables to share registers.
- The pointer for X and W are maintained on both register sides to avoid crosspath Conflicts.
- Variable that is used as inner-loop counter.
- The variable, K, is used as the outer-loop counter. We are finished when n2b = 0.
- The twiddle factor array w can be generated by the tw_r4fft function provided in dsplib\support\fft\tw_r4fft.c. The exe file for this function, dsplib\bin\tw_r4fft.exe, can be used dump the twiddle factor array into a file.
- The function bit_rev in dsplib\support\fft can be used to bit-reverse the output array to convert it into normal order.
- ***Endianness:*** This code is little endian.

- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	14*n*log4(n) + 46 e.g., if n = 256, cycles = 14382.
Code size (in bytes)	1344

DSPF_dp_cfftr2 *Double-precision cache optimized radix-2 forward FFT with complex input*

Function void DSPF_dp_cfftr2 (int n, double * x, double * w, int n_min)

Arguments

x	Input and output sequences (dim-n) (input/output). x has n complex numbers (2*n DP values). The real and imaginary values are interleaved in memory. The input is in normal order and output is in bit-reversed order.
w	FFT coefficients (dim-n) (input). w has n complex numbers (n DP values). FFT coefficients are in a special sequence so that FFT can be called on smaller input sets multiple times to avoid cache thrashing. The real and imaginary values are interleaved in memory.
n	FFT size which is a power of 2 and > 4 (input).

Description This routine is used to compute the complex, radix-2, fast fourier transform of a double-precision complex sequence of size n, and a power of 2 in a cache-friendly way. The routine requires normal order input and normal order coefficients (twiddle factors) in a special sequence and produces results that are in bit-reversed order.

The input can be broken into smaller parts and called multiple times to avoid cache thrashing.

```
How to use
void main(void)
{
```

```
gen_w_r2(w, N); // Generate coefficient table
              // in normal order
              // Function is given in C-CODE section
dp_cfftr2(N, x, w, 1); // input in normal order, output
              // in order bit-reversed
bit_rev(x, N) // Bit reverse the output if
                  // normal order output is needed
                 // Function is given in C-CODE section
}
```

Main fft of size N can be divided into several steps (where number of steps is a power of 2), allowing as much data reuse as possible.

For example the following function:

```
dp_cfftr2(N, x, w, 1);
```

is equivalent to:

```
dp_cfftr2(N, x, w, N/4);
dp_cfftr2(N/4, &x[2 * 0 * (N/4)], &w[N + N/2], 1);
dp_cfftr2(N/4, &x[2 * 1 * (N/4)], &w[N + N/2], 1);
dp_cfftr2(N/4, &x[2 * 2 * (N/4)], &w[N + N/2], 1);
dp_cfftr2(N/4, &x[2 * 3 * (N/4)], &w[N + N/2], 1);
```

Notice how the first fft function is called on the entire data set. It covers the first pass of the fft until the butterfly size is N/4. The following 4 ffts do N/4 point ffts, 25% of the original size. These continue down to the end when the butterfly is of size 2. We use an index of 2* 3/4 *N to the main twiddle factor array for the last 4 calls. This is because the twiddle factor array is composed of successively decimated versions of the main array. The twiddle factor array is composed of log2(N) sets of twiddle factors of size N, N/2, N/4, N/8 etc. The index into this array for each stage of the fft can be calculated by summing these indices up appropriately. For example, if we are dividing the input into 2 parts then index into this array should be N, if we are dividing into 4 parts then index into this array should be N+N/2, if we are dividing into 8 parts index should be N+N/2+N/4. For multiple ffts they can share the same table by calling the small ffts from further down in the twiddle factor array, in the same way as the decomposition works for more data reuse. The functions for creating this special sequence of twiddle factors and bit-reversal are provided in the C CODE section. In general if divide the input into NO_OF_DIV parts we can call the function as follows:

```
// Divide the input into NO_OF_DIV parts
dp_cfftr2(N, x, w, N/NO_OF_DIV);
```

```
// Find out the index into twiddle factor array
for(w_index=0,j = NO_OF_DIV; j > 1 ; j >>= 1)
{
  w_index += j;
}
w_index = N * w_index / NO_OF_DIV;
// Call the Function a subset of inputs
for(i=0; i<NO_OF_DIV; i++)
{
   dp_cfftr2(N/NO_OF_DIV, &x[2*i*(N/NO_OF_DIV)], &w[w_in-
dex], 1);
}
```

Algorithm

This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_dp_cfftr2(int n, double * x, double * w, int
n_min)
{
     int n2, ie, ia, i, j, k, m;
     double rtemp, itemp, c, s;
     n2 = n;
     ie = 1;
     for(k = n; k > n_min; k >>= 1)
      {
        n2 >>= 1;
        ia = 0;
        for(j=0; j < ie; j++)
        {
  for(i=0; i < n2; i++)
            {
               c = w[2*i];
               s = w[2*i+1];
               m = ia + n2;
               rtemp     = x[2*ia]   - x[2*m];
               x[2*ia]   = x[2*ia]   + x[2*m];
               itemp     = x[2*ia+1] - x[2*m+1];
```

```
                x[2*ia+1] = x[2*ia+1] + x[2*m+1];
                x[2*m]    = c*rtemp   - s*itemp;
                x[2*m+1]  = c*itemp   + s*rtemp;
                ia++;
               }
            ia += n2;
           }
          ie <<= 1;
          w = w + k;
       }
}
```

The following C code is used to generate the coefficient table.

```
#include <math.h>
/* generate real and imaginary twiddle
   table of size n complex numbers (or 2*n numbers) */
void gen_w_r2(double* w, int n)
{
     int i, j=1;
     double pi = 4.0*atan(1.0);
     double e = pi*2.0/n;
     for(j=1; j < n; j <<= 1)
     {
     for(i=0; i < ( n>>1 ); i += j)
        {
        *w++   = cos(i*e);
        *w++   = -sin(i*e);
        }
     }
}
```

The following C code is used to bit-reverse the output.

```
bit_rev(double* x, int n)
{
  int i, j, k;
  double rtemp, itemp;
```

```
  j = 0;
  for(i=1; i < (n-1); i++)
  {
    k = n >> 1;
     while(k <= j)
     {
        j -= k;
        k >>= 1;
     }
     j += k;
     if(i < j)
     {
        rtemp    = x[j*2];
        x[j*2]   = x[i*2];
        x[i*2]   = rtemp;
        itemp    = x[j*2+1];
        x[j*2+1] = x[i*2+1];
        x[i*2+1] = itemp;
     }
  }
}
```

Special Requirements

- Both input x and coefficient w should be aligned on double-word boundary.
- The value of n should be greater than 4 and a power of 2.

Implementation Notes

- Outer loop instructions are executed in parallel with the inner loop epilog.
- The special sequence of twiddle factor array w can be generated using the gen_w_r2 function provided in the previous section.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	4 * n * lg(n) + 16 * lg(n) + 34 e.g., IF n = 64, cycles = 1666 e.g., IF n = 32, cycles = 754
Code size (in bytes)	1408

DSPF_dp_icfftr2

Double-precision cache optimized radix-2 inverse FFT with complex input

Function void DSPF_dp_icfftr2 (int n, double * x, double * w, int n_min)

Arguments

x	Input and output sequences (dim-n) (input/output). x has n complex numbers (2*n DP values). The real and imaginary values are interleaved in memory. The input is in normal order and output is in bit-reversed order.
w	FFT coefficients (dim-n) (input). w has n complex numbers (n DP values). FFT coefficients are in a special sequence so that FFT can be called on smaller input sets multiple times to avoid cache thrashing. The real and imaginary values are interleaved in memory.
n	FFT size which is a power of 2 and > 4 (input).

Description This routine is used to compute the inverse complex radix-2, fast fourier transform of a double-precision complex sequence of size n, and a power of 2 in a cache-friendly way. The routine requires normal order input and normal order coefficients (twiddle factors) in a special sequence and produces results that are in bit-reversed order.

The input can be broken into smaller parts and called multiple times to avoid cache thrashing.

```
How to use
void main(void)
{
   gen_w_r2(w, N); // Generate coefficient table
                 // in normal order
                 // Function is given in C-CODE section
  dp_icfftr2(N, x, w, 1); // input in normal order, output
                 // in order bit-reversed
  bit_rev(x, N) // Bit reverse the output if
                    // normal order output is needed
                    // Function is given in C-CODE section
```

```
divide(x, N); // scale inverse FFT output
// result is the same as original
// input
}
```

Main Inverse fft of size N can be divided into several steps (where number of steps is a power of 2), allowing as much data reuse as possible.

For example the following function

```
dp_icfftr2(N, x, w, 1);
```

is equivalent to:

```
dp_icfftr2(N, x, w, N/4);
dp_icfftr2(N/4, &x[2 * 0 * (N/4)], &w[N + N/2], 1);
dp_icfftr2(N/4, &x[2 * 1 * (N/4)], &w[N + N/2], 1);
dp_icfftr2(N/4, &x[2 * 2 * (N/4)], &w[N + N/2], 1);
dp_icfftr2(N/4, &x[2 * 3 * (N/4)], &w[N + N/2], 1);
```

Notice how the first icfft function is called on the entire data set. It covers the first pass of the fft until the butterfly size is N/4. The following 4 ffts do N/4 point ffts, 25% of the original size. These continue down to the end when the butterfly is of size 2. We use an index of 2* 3/4 *N to the main twiddle factor array for the last 4 calls. This is because the twiddle factor array is composed of successively decimated versions of the main array. The twiddle factor array is composed of log2(N) sets of twiddle factors of size N, N/2, N/4, N/8 etc. The index into this array for each stage of the fft can be calculated by summing these indices up appropriately. For example, if we are dividing the input into 2 parts then index into this array should be N, if we are dividing into 4 parts then index into this array should be N+N/2, if we are dividing into 8 parts index should be N+N/2+N/4. For multiple iffts they can share the same table by calling the small iffts from further down in the twiddle factor array, in the same way as the decomposition works for more data reuse. The functions for creating this special sequence of twiddle factors and bit-reversal are provided in the C CODE section. In general if divide the input into NO_OF_DIV parts we can call the function as follows:

```
// Divide the input into NO_OF_DIV parts
dp_icfftr2(N, x, w, N/NO_OF_DIV);
// Find out the index into twiddle factor array
for(w_index=0,j = NO_OF_DIV; j > 1 ; j >>= 1)
{
   w_index += j;
}
w_index = N * w_index / NO_OF_DIV;
// Call the Function a subset of inputs
for(i=0; i<NO_OF_DIV; i++)
{
```

```
    dp_icfftr2(N/NO_OF_DIV, &x[2*i*(N/NO_OF_DIV)],
    &w[w_index], 1);
}
```

Algorithm

This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_dp_icfftr2(int n, double * x, double * w, int
n_min)
{
     int n2, ie, ia, i, j, k, m;
     double rtemp, itemp, c, s;
     n2 = n;
     ie = 1;
     for(k = n; k > n_min; k >>= 1)
      {
        n2 >>= 1;
        ia = 0;
        for(j=0; j < ie; j++)
        {
  for(i=0; i < n2; i++)
          {
            c = w[2*i];
            s = w[2*i+1];
            m = ia + n2;
            rtemp     = x[2*ia]   - x[2*m];
            x[2*ia]   = x[2*ia]   + x[2*m];
            itemp     = x[2*ia+1] - x[2*m+1];
            x[2*ia+1] = x[2*ia+1] + x[2*m+1];
            x[2*m]    = c*rtemp   + s*itemp;
            x[2*m+1]  = c*itemp   - s*rtemp;
            ia++;
           }
        ia += n2;
       }
      ie <<= 1;
      w = w + k;
   }
```

```
}
```

The following C code is used to generate the coefficient table.

```
#include <math.h>
/* generate real and imaginary twiddle
   table of size n complex numbers (or 2*n numbers) */
void gen_w_r2(double* w, int n)
{
     int i, j=1;
     double pi = 4.0*atan(1.0);
     double e = pi*2.0/n;
     for(j=1; j < n; j <<= 1)
     {
     for(i=0; i < ( n>>1 ); i += j)
        {
        *w++   = cos(i*e);
        *w++   = -sin(i*e);
        }
     }
}
```

The following C code is used to bit-reverse the output.

```
bit_rev(double* x, int n)
{
  int i, j, k;
  float rtemp, itemp;
  j = 0;
  for(i=1; i < (n-1); i++)
  {
    k = n >> 1;
     while(k <= j)
     {
        j -= k;
        k >>= 1;
     }
     j += k;
```

```
     if(i < j)
     {
        rtemp    = x[j*2];
        x[j*2]   = x[i*2];
        x[i*2]   = rtemp;
        itemp    = x[j*2+1];
        x[j*2+1] = x[i*2+1];
        x[i*2+1] = itemp;
     }
  }
}
```

The following C code is used to perform the final scaling of the IFFT:

```
/* divide each element of x by n */

divide(double* x, int n)
{
         int i;
  double inv = 1.0 / n;
    for(i=0; i < n; i++)
     {
        x[2*i] = inv * x[2*i];
               x[2*i+1] = inv * x[2*i+1];
             }
        }
```

Special Requirements

- Both input x and coefficient w should be aligned on double-word boundary.
- The value of n should be greater than 4 and a power of 2.

Implementation Notes

- Outer loop instructions are executed in parallel with the inner loop epilog.
- The special sequence of twiddle factor array w can be generated using the gen_w_r2 function provided in the previous section.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	4 * n * lg(n) + 16 * lg(n) + 34 e.g., IF n = 64, cycles = 1666 e.g., IF n = 32, cycles = 754
Code size (in bytes)	1408

4.2.4 Filtering and Convolution

DSPF_dp_fir_cplx *Double-precision complex finite impulse response filter*

Function void DSPF_dp_fir_cplx (const double * restrict x, const double * restrict h, double * restrict r, int nh, int nr)

Arguments

x[2*(nr+nh-1)]	Pointer to complex input array. The input data pointer x must point to the (nh)th complex element, i.e., element 2*(nh-1).
h[2*nh]	Pointer to complex coefficient array (in normal order).
r[2*nr]	Pointer to complex output array.
nh	Number of complex coefficients in vector h.
nr	Number of complex output samples to calculate.

Description This function implements the FIR filter for complex input data.

The filter has nr output samples and nh coefficients. Each array consists of an even and odd term with even terms representing the real part and the odd terms the imaginary part of the element. The coefficients are expected in normal order.

Algorithm This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_dp_fir_cplx(const double * x, const double * h,
                  double * restrict r, int nh, int nr)
{
     int i,j;
     double imag, real;
```

```
        for (i = 0; i < 2*nr; i += 2)
        {
            imag = 0;
            real = 0;
            for (j = 0; j < 2*nh; j += 2)
            {
                real += h[j] * x[i-j] - h[j+1] * x[i+1-j];
                imag += h[j] * x[i+1-j] + h[j+1] * x[i-j];
            }
            r[i] = real;
            r[i+1] = imag;
        }
}
```

Special Requirements

- nr is a multiple of 2 and greater than or equal to 2.
- nh is greater than or equal to 4.
- x points to 2*(nh-1)th input element.

Implementation Notes

- The outer loop is unrolled twice.
- Outer loop instructions are executed in parallel with inner loop.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	8*nh*nr + 5*nr + 30 For nh=24 and nr=48, cycles=9486 For nh=16 and nr=36, cycles=4818
Code size (in bytes)	608

DSPF_dp_fir_gen *Double-precision generic FIR filter*

Function void DSPF_dp_fir_gen (const double *x, const double *h, double * restrict r, int nh, int nr)

Arguments

x	Pointer to array holding the input floating-point array.
h	Pointer to array holding the coefficient floating-point array.
r	Pointer to output array.
nh	Number of coefficients.
nr	Number of output values.

Description This routine implements a block FIR filter. There are “nh” filter coefficients, “nr” output samples, and “nh+nr-1” input samples. The coefficients need to be placed in the “h” array in reverse order {h(nh-1), ... , h(1), h(0)} and the array “x” starts at x(-nh+1) and ends at x(nr-1). The routine calculates y(0) through y(nr-1) using the following formula:

r(n) = h(0)*x(n) + h(1)*x(n-1) + ... + h(nh-1)*x(n-nh+1)

where n = {0, 1, ... , nr-1}.

Algorithm This is the C equivalent for the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_dp_fir_gen(const double *x, const double *h,
            double * restrict r, int nh, int nr)
{
   int i, j;
   double sum;
   for(i=0; i < nr; i++)
   {
      sum = 0;
      for(j=0; j < nh; j++)
      {
          sum += x[i+j] * h[j];
      }
      r[i] = sum;
   }
}
```

Special Requirements

- Little endianness is assumed for LDDW instructions.
- The number of coefficients must be greater than or equal to 4.
- The number of outputs must be greater than or equal to 4

Implementation Notes

- The outer loop is unrolled 4 times.
- The inner loop is unrolled 2 times and software pipelined.
- Register sharing is used to make optimum utilization of available registers
- Outer loop instructions and Prolog for next stage are scheduled in parallel with last iteration of kernel
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	(16*floor((nh+1)/2)+10)*(ceil(nr/4)) + 32 for nh=26, nr=42, cycles=2430 cycles.
Code size (in bytes)	672

DSPF_dp_fir_r2 *Double-precision complex finite impulse response filter*

Function void DSPF_dp_fir_r2 (const double * restrict x, const double * restrict h, double * restrict r, int nh, int nr)

Arguments

x[nr+nh-1]	Pointer to Input array of size nr+nh-1.
h[nh]	Pointer to coefficient array of size nh (in reverse order).
r[nr]	Pointer to output array of size nr.
nh	Number of coefficients.
nr	Number of output samples.

Description Computes a real FIR filter (direct-form) using coefficients stored in vector h[]. The real data input is stored in vector x[]. The filter output result is stored in vector r[]. The filter calculates nr output samples using nh coefficients. The coefficients are expected to be in reverse order.

Algorithm

This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_dp_fir_r2(const double * x, const double * h,
                   double *restrict r, int nh, int nr)
{
    int i, j;
    double sum;
    for (i = 0; i < nr; i++)
    {
        sum = 0;
 for (j = 0; j < nh; j++)
           sum += x[i + j] * h[j];
    r[i] = sum;
    }
}
```

Special Requirements

- nr is a multiple of 2 and greater than or equal to 2.
- nh is a multiple of 2 and greater than or equal to 8.
- Coefficients in array h are expected to be in reverse order.
- x and h should be padded with 4 words at the end.

Implementation Notes

- The outer loop is unrolled four times and inner loop is unrolled twice.
- Register sharing is used to make optimum utilization of available registers.
- Outer loop instructions are executed in parallel with inner loop.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	(8*nh + 10)*ceil(nr/4) + 32 For nh=24 and nr=62, cycles=3264
Code size (in bytes)	672

DSPF_dp_fircirc — *Double-precision circular FIR algorithm*

Function void DSPF_dp_fircirc (double *x, double *h, double *r, int index, int csize, int nh, int nr)

Arguments

x[]	Input array (circular buffer of 2^(csize+1) bytes). Must be aligned at 2^(csize+1) byte boundary.
h[nh]	Filter coefficients array. Must be double-word aligned.
r[nr]	Output array.
index	Offset by which to start reading from the input array. Must be a multiple of 2.
csize	Size of circular buffer x[] is 2^(csize+1) bytes. Must be 2 ≤ csize ≤ 31.
nh	Number of filter coefficients. Must be a multiple of 2 and ≥ 4.
nr	Size of output array. Must be a multiple of 4.

Description This routine implements a circularly addressed FIR filter. The variable nh is the number of filter coefficients. The variable nr is the number of output samples.

Algorithm This is the C equivalent for the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_dp_fircirc  (double x[], double h[], double r[],
              int index, int csize, int nh, int nr)
{
    int i, j;
    //Circular Buffer block size = ((2^(csize + 1)) / 8)
    //floating point numbers
    int mod = (1 << (csize - 2));
    double r0;
    for (i = 0; i < nr; i++)
    {
        r0 = 0;
        for (j = 0; j < nh; j++)
        {
         //Operation "% mod" is equivalent to "& (mod -1)"
```

```
            //r0 += x[(i + j + index) % mod] * h[j];
         r0 += x[(i + j + index) & (mod - 1)] * h[j];
            }
        r[i] = r0;
    }
}
```

Special Requirements

- The circular input buffer x[] must be aligned at a 2^(csize+1) byte boundary. csize must lie in the range 2 <= csize <= 31.
- The number of coefficients (nh) must be a multiple of 2 and greater than or equal to 4.
- The number of outputs (nr) must be a multiple of 4 and greater than or equal to 4.
- The index (offset to start reading input array) must be a multiple of 2 and less than or equal to (2^(csize-2) - 6)
- The coefficient array is assumed to be in reverse order; that is, h(nh-1) to h(0) hold coeffs. h0, h1, h2 etc.

Implementation Notes

- The outer loop is unrolled 4 times.
- The inner loop is unrolled 2 times.
- Register sharing is due to make optimal utilization of the available registers.
- Outer loop instructions and prolog for next stage are scheduled in the last cycle of Kernel.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	(2*nh + 2) nr + 38 For nh = 36 & nr=64, cycles = 4774
Code size (in bytes)	640

DSPF_dp_biquad *Double-precision second order IIR (biquad) filter*

Function void DSPF_dp_biquad (double *x, double *b, double *a, double *delay, double *r, int nx)

Arguments

x	Pointer to input samples.
b	Pointer to nr coefs b0, b1, b2.
a	Pointer to dr coefs a1, a2.
delay	Pointer to filter delays.
r	Pointer to output samples.
nx	Number of input/output samples.

Description This routine implements a DF 2 transposed structure of the biquad filter. The transfer function of a biquad can be written as:

$$H(Z) = \frac{b(0) + b(1)z \wedge (-1) + b(2)z \wedge (-2)}{1 + a(1)z \wedge (-1) + a(2)z \wedge (-2)}$$

Algorithm

```
void DSPF_dp_biquad(double *x, double *b, double *a,
                         double *delay, double *r, int nx)
{
    int i;
    double a1, a2, b0, b1, b2, d0, d1, x_i;
    a1 = a[0];
    a2 = a[1];
    b0 = b[0];
    b1 = b[1];
    b2 = b[2];
    d0 = delay[0];
    d1 = delay[1];
    for (i = 0; i < nx; i++)
    {
        x_i = x[i];
        r[i] = b0 * x_i + d0;
        d0 = b1 * x_i - a1 * r[i] + d1;
```

```
        d1 = b2 * x_i - a2 * r[i];
    }
    delay[0] = d0;
    delay[1] = d1;
}
```

Special Requirements The value of nx is ≥ 4.

Implementation Notes

- Register sharing has been used to optimize on the use of registers.
- x[i] is loaded on both sides to avoid crosspath conflict
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	16 * nx + 49 For nx = 64, cycles = 1073 For nx = 48, cycles = 817
Code size (in bytes)	576

DSPF_dp_iir *Double-precision IIR filter (used in the VSELP vocoder)*

Function void DSPF_dp_iir (double* restrict r1, const double* x, double* restrict r2, const double* h2, const double* h1, int nr)

Arguments

r1[nr+4]	Delay element values (i/p and o/p).
x[nr]	Pointer to the input array.
r2[nr+4]	Pointer to the output array.
h2[5]	Auto-regressive filter coefficients.
h1[5]	Moving average filter coefficients.
nr	Number of output samples.

Description The IIR performs an auto-regressive moving-average (ARMA) filter with 4 auto-regressive filter coefficients and 5 moving-average filter coefficients for nr output samples. The output vector is stored in two locations. This routine is used as a high pass filter in the VSELP vocoder. The 4 values in the r1 vector store the initial values of the delays.

Algorithm

This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_dp_iir (double* restrict r1,
             const double*    x,
             double* restrict r2,
             const double*    h2,
             const double*    h1,
             int nr
             )
{
   int i, j;
   double sum;
   for (i = 0; i < nr; i++)
   {
      sum = h2[0] * x[4+i];
      for (j = 1; j <= 4; j++)
         sum += h2[j] * x[4+i-j] - h1[j] * r1[4+i-j];
      r1[4+i] = sum;
      r2[i] = r1[4+i];
   }
}
```

Special Requirements

- The value of nr must be > 0.
- Extraneous loads are allowed in the program.

Implementation Notes

- The inner loop is completely unrolled so that two loops become one loop.
- Register sharing is used to make optimum utilization of available registers.
- ***Endianness:*** The code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	24*nr + 48 e.g., for nr = 32, cycles = 816
Code size (in bytes)	608

DSPF_dp_iirlat *Double-precision all-pole IIR lattice filter*

Function

void DSPF_dp_iirlat (double *x, int nx, const double * restrict k, int nk, double * restrict b, double * r)

Arguments

x[nx]	Input vector.
nx	Length of input vector.
k[nk]	Reflection coefficients.
nk	Number of reflection coefficients/lattice stages. Must be a multiple of 2 and ≥ 6.
b[nk+1]	Delay line elements from previous call. Should be initialized to all zeros prior to the first call.
r[nx]	Output vector.

Description

This routine implements a real all-pole IIR filter in lattice structure (AR lattice). The filter consists of nk lattice stages. Each stage requires one reflection coefficient k and one delay element b. The routine takes an input vector x[] and returns the filter output in r[]. Prior to the first call of the routine the delay elements in b[] should be set to zero. The input data may have to be pre-scaled to avoid overflow or achieve better SNR. The reflections coefficients lie in the range −1.0 < k < 1.0. The order of the coefficients is such that k[nk-1] corresponds to the first lattice stage after the input and k[0] corresponds to the last stage.

Algorithm

```
void DSPF_dp_iirlat(double * x, int nx,
                const double * restrict k, int nk,
                double * restrict b, double * r)
{
    double rt;     // output       //
    int i, j;
    for (j = 0; j < nx; j++)
    {
       rt = x[j];
       for (i = nk - 1; i >= 0; i--)
       {
           rt = rt - b[i] * k[i];
           b[i + 1] = b[i] + rt * k[i];
```

```
        }
        b[0] = rt;
        r[j] = rt;
    }
}
```

Special Requirements

- nk is a multiple of 2 and ≥ 6.
- Extraneous loads are allowed (80 bytes) before the start of array.

Implementation Notes

- The loop has been unrolled by 4 times.
- Register sharing has been used to optimize on the use of registers.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	(24*Ceil(nk/4) + 19)* nx + 33 For nk = 14, nx = 64 cycles = 7393
Code size (in bytes)	832

DSPF_dp_convol *Double-precision convolution*

Function void DSPF_dp_convol (double *x, double *h, double *r, int nh, int nr)

Arguments

x	Pointer to real input vector of size = nr+nh-1 a typically contains input data (x) padded with consecutive nh − 1 zeros at the beginning and end.
h	Pointer to real input vector of size nh in forward order. h typically contains the filter coefs.
r	Pointer to real output vector of size nr.
nh	Number of elements in vector b. Note: nh ≤ nr nh is typically noted as m in convol formulas. nh must be a multiple of 2.
nr	Number of elements in vector r. nr must be a multiple of 4.

Description

This function calculates the full-length convolution of real vectors x and h using time-domain techniques. The result is placed in real vector r. It is assumed that input vector x is padded with nh-1 no of zeros in the beginning and end. It is assumed that the length of the input vector h, nh, is a multiple of 2 and the length of the output vector r, nr, is a multiple of 4. nh is greater than or equal to 4 and nr is greater than or equal to nh. The routine computes 4 output samples at a time.

Algorithm

This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_dp_convol(double *x, double *h, double *r,
                    short nh, short nr)
{
short   octr, ictr;
double  acc ;
for (octr = nr ; octr > 0 ; octr--)
{
      acc = 0 ;
      for (ictr = nh ; ictr > 0 ; ictr--)
      {
     acc += x[nr-octr+nh-ictr]*h[(ictr-1)];
      }
      r[nr-octr] = acc;
}
}
```

Special Requirements

- nh is a multiple of 2 and greater than or equal to 4
- nr is a multiple of 4

Implementation Notes

- The inner loop is unrolled twice and the outer loop is unrolled four times.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	2*(nh*nr) + 5/2*nr + 32 For nh=24 and nr=48, cycles=2456 For nh=20 and nr=32, cycles=1392
Code size (in bytes)	544

4.2.5 Math

DSPF_dp_dotp_sqr *Double-precision dot product and sum of square*

Function double DSPF_dp_dotp_sqr (double G, const double * x, const double * y, double * restrict r, int nx)

Arguments

x[nx]	Pointer to first input array.
y[nx]	Pointer to second input array.
r	Pointer to output for accumulation of x[]*y[].
nx	Length of input vectors.

Description This routine computes the dot product of x[] and y[] arrays, adding it to the value in the location pointed to by r. Additionally, it computes the sum of the squares of the terms in the y array,adding it to the argument G. The final value of G is given as the return value of the function.

Algorithm This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
double DSPF_dp_dotp_sqr(double G, const double * x, const
                  double * y, double *restrict r, int nx)
{
   int i;
   for (i = 0; i < nx; i++)
   {
         *r += x[i] * y[i];      /* Compute Dot Product */
         G += y[i] * y[i];       /* Compute Square  */
   }
   return G;
}
```

Special Requirements There are no special alignment requirements.

Implementation Notes

- Multiple assignment was used to reduce loop carry path.
- ***Endianness:*** This code is little endian .

- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	4*nx + 26 For nx=64, cycles=282. For nx=30, cycles=146
Code size (in bytes)	244

DSPF_dp_dotprod *Dot product of 2 double-precision float vectors*

Function double DSPF_dp_dotprod (const double *x, const double *y, const int nx)

Arguments

x	Pointer to array holding the first floating-point vector.
y	Pointer to array holding the second floating-point vector.
nx	Number of values in the x and y vectors.

Description This routine calculates the dot product of 2 double-precision float vectors.

Algorithm This is the C equivalent for the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
double DSPF_dp_dotprod(const double *x, const double *y,
                 const int nx)
{
   int i;
   double sum = 0;
   for (i=0; i < nx; i++)
   {
      sum += x[i] * y[i];
   }
   return sum;
}
```

Special Requirements

- A memory pad of 4 bytes is required at the end of each array if the number of inputs is odd.

- The value of nx must be > 0.

Implementation Notes

- The loop is unrolled once and software pipelined. However, by conditionally adding to the dot product odd numbered array sizes are also permitted.
- Multiple assignments are used to reduce loop carry path
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	4*ceil(nx/2) + 33 e.g., for nx = 256, cycles = 545
Code size (in bytes)	256

DSPF_dp_dotp_cplx *Complex double-precision floating-point dot product*

Function void DSPF_dp_dotp_cplx (const double *x, const double *y, int n, double *restrict re, double * restrict im)

Arguments

x	Pointer to array holding the first floating-point vector.
y	Pointer to array holding the second floating-point vector.
n	Number of values in the x and y vectors.
re	Pointer to the location storing the real part of the result.
im	Pointer to the location storing the imaginary part of the result.

Description This routine calculates the dot product of two double-precision complex float vectors. The even numbered locations hold the real parts of the complex numbers while the odd numbered locations contain the imaginary portions.

Algorithm This is the C equivalent for the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void dp_dotp_cplx(const double* x, const double* y, int n,
```

```
            double* restrict re, double* restrict im)
{
    double real=0, imag=0;
    int i=0;
    for(i=0; i<n; i++)
    {
       real+=(x[2*i]*y[2*i] - x[2*i+1]*y[2*i+1]);
       imag+=(x[2*i]*y[2*i+1] + x[2*i+1]*y[2*i]);
    }
    *re=real;
    *im=imag;
}
```

Special Requirements The value of nx must be > 0.

Implementation Notes

- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	8*N + 29 e.g., for N = 128, cycles = 1053
Code size (in bytes)	352

DSPF_dp_maxval *Maximum element of double-precision vector*

Function double DSPF_dp_maxval (const double* x, int nx)

Arguments

x	Pointer to input array.
nx	Number of Inputs in the input array.

Description This routine finds out the maximum number in the input array. This code returns the maximum value in the array.

Algorithm

This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
double DSPF_dp_maxval(const double* x, int nx)
{
   int i;
   double max;
   *((int *)&max) = 0x00000000;
   *((int *)&max+1) = 0xfff00000;
   for (i = 0; i < nx; i++)
   if (x[i] > max)
   {
     max = x[i];
    }
   return max;
}
```

Special Requirements

- The value of nx should be a multiple of 2 and ≥ 2.
- NAN (not a number in double-precision format) in the input is disregarded.

Implementation Notes

- The loop is unrolled six times.
- Six maximums are maintained in each iteration.
- NAN (not a number in -precision format) in the input are disregarded.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	7*ceil(nx/6) + 31 For nx=60, cycles=101 For nx=34, cycles=73
Code size (in bytes)	672

DSPF_dp_maxidx *Index of maximum element of double-precision vector*

Function int DSPF_dp_maxidx (const double* x, int nx)

Arguments

x	Pointer to input array.
nx	Number of Inputs in the input array.

Description This routine finds out the index of maximum number in the input array. This function returns the index of the greatest value.

Algorithm

```
    int DSPF_dp_maxidx(const double* x, int nx)
    {
       int index, i;
       double max;
*((int *)&max) = 0x00000000;
       *((int *)&max+1) = 0xfff00000;
       for (i = 0; i < nx; i++)
       if (x[i] > max)
      {
         max = x[i];
         index = i;
      }
      return index;
    }
```

Special Requirements

- The value of nx is a multiple of 3.
- The range is nx ≥ 3, and nx ≤ 2^16−1.

Implementation Notes

- The loop is unrolled three times.
- Three maximums are maintained in each iteration.
- MPY instructions are used for move.
- ***Endianness:*** This code is little endian.

- ❑ ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	4*nx/3 + 22 For nx=60, cycles=102 For nx=30, cycles=62
Code size (in bytes)	448

DSPF_dp_minval *Minimum element of double-precision vector*

Function double DSPF_dp_minval (const double* x, int nx)

Arguments

x	Pointer to input array.
nx	Number of Inputs in the input array.

Description This routine finds out and returns the minimum number in the input array.

Algorithm

```
double DSPF_dp_minval(const double* x, int nx)
{
   int i;
   float min;
   *((int *)&min) = 0x00000000;
*((int *)&min+1) = 0x7ff00000;
   for (i = 0; i < nx; i++)
   if (x[i] < min)
   {
     min = x[i];
    }
   return min;
}
```

Special Requirements

- ❑ The value of nx should be a multiple of 2 and ≥ 2.

- NAN (not a number in double-precision format) in the input are disregarded.

Implementation Notes

- The loop is unrolled six times.
- Six minimums are maintained in each iteration.
- NAN (not a number in double-precision format) in the input are disregarded.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	7*ceil(nx/6) + 31 For nx=60 cycles=101 For nx=34 cycles=73
Code size (in bytes)	640

DSPF_dp_vecrecip *Double-precision vector reciprocal*

Function void DSPF_dp_vecrecip (const double *x, double * restrict r, int n)

Arguments

x	Pointer to input array.
r	Pointer to output array.
n	Number of elements in array.

Description The dp_vecrecip module calculates the reciprocal of each element in the array x and returns the output in array r. It uses 3 iterations of the Newton-Raphson method to improve the accuracy of the output generated by the RCPDP instruction of the C67x. Each iteration doubles the accuracy. The initial output generated by RCPDP is 8 bits. So after the first iteration it is 16 bits and after the second it is the 23 bits and after third it is full 52 bits. The formula used is:

r[n+1] = r[n](2 − v*r[n])

where v = the number whose reciprocal is to be found.

x[0], the seed value for the algorithm, is given by RCPDP.

Algorithm This is the C equivalent of the assembly code without restrictions.

```
void DSPF_dp_vecrecip(const double* x, double* restrict r,
int n)
   {
 int i;
 for(i = 0; i < n; i++)
 r[i] = 1 / x[i];
   }
```

Special Requirements There are no alignment requirements.

Implementation Notes

- The inner loop is unrolled four times to allow calculation of four reciprocals in the kernel. However, the stores are executed conditionally to allow n to be any number > 0.
- Register sharing is used to make optimal use of available registers.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	78*ceil(n/4) + 24 e.g., for n = 54, cycles = 1116
Code size (in bytes)	448

DSPF_dp_vecsum_sq *Double-precision sum of squares*

Function double DSPF_dp_vecsum_sq (const double *x, int n)

Arguments

x	Pointer to input array.
n	Number of elements in array.

Description This routine performs a sum of squares of the elements of the array x and returns the sum.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
double DSPF_dp_vecsum_sq(const double *x,int n)
{
   int i;
   double sum=0;
   for(i = 0;  i < n; i++ )
        sum += x[i]*x[i];
   return sum;
}
```

Special Requirements Since loads of 4 doubles beyond the array occur, a pad must be provided.

Implementation Notes

- The inner loop is unrolled twice. Hence, two registers are used to hold the sum of squares. ADDDPs are staggered.
- ***Endianness:*** This code is endian neutral.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	4*Ceil(n/2) + 33 e.g., for n = 100, cycles = 233
Code size (in bytes)	288

DSPF_dp_w_vec *Double-precision weighted sum of vectors*

Function void DSPF_dp_w_vec (const double* x, const double* y, double m, double * restrict r, int nr)

Arguments

x	Pointer to first input array.
y	Pointer to second input array.
m	Weight factor.
r	Output array pointer.
nr	Number of elements in arrays.

Description This routine is used to obtain the weighted vector sum.

Both the inputs and output are double-precision floating-point numbers.

Algorithm This is the C equivalent of the assembly code without restrictions.

```
void DSPF_dp_w_vec( const double * x,const double * y,
                  double m, double * restrict r,int nr)
{
   int i;
   for (i = 0; i < nr; i++)
        r[i] = (m * x[i]) + y[i];
}
```

Special Requirements The value of nr must be > 0.

Implementation Notes

- The inner loop is unrolled twice.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	4*Ceil(n/2) + 32 e.g., for n = 100, cycles = 232
Code size (in bytes)	352

DSPF_dp_vecmul *Double-precision vector multiplication*

Function void DSPF_dp_vecmul (const double *x, const double *y, double * restrict r, int n)

Arguments

x	Pointer to first input array.
y	Pointer to second input array.
r	Pointer to output array.
n	Number of elements in arrays.

Description This routine performs an element by element double-precision floating-point multiplication of the vectors x[] and y[] and returns the values in r[].

Algorithm This is the C equivalent of the assembly code without restrictions.

```
void DSPF_dp_vecmul(const double * x, const double * y,
 double * restrict r, int n)
 {
   int i;
   for(i = 0; i < n; i++)
       r[i] = x[i] * y[i];
 }
```

Special Requirements The value of n > 0.

Implementation Notes

- The inner loop is unrolled twice to allow calculation of 2 outputs in the kernel. However the stores are executed conditionally to allow n to be any number > 0.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	4*Ceil(n/2) + 13 e.g., for n = 100, cycles = 213
Code size (in bytes)	256

4.2.6 Matrix

DSPF_dp_mat_mul *Double-precision matrix multiplication*

Function void DSPF_dp_mat_mul (double *x, int r1, int c1, double *y, int c2, double *r)

Arguments

x	Pointer to r1 by c1 input matrix.
r1	Number of rows in x.
c1	Number of columns in x. Also number of rows in y.
y	Pointer to c1 by c2 input matrix.
c2	Number of columns in y.
r	Pointer to r1 by c2 output matrix.

Description This function computes the expression r = x * y for the matrices x and y. The column dimension of x must match the row dimension of y. The resulting matrix has the same number of rows as x and the same number of columns as y.

The values stored in the matrices are assumed to be double-precision floating-point values.

This code is suitable for dense matrices. No optimizations are made for sparse matrices.

Algorithm

```
void DSPF_dp_mat_mul(double *x, int r1, int c1,
                double *y, int c2, double *r)
{
    int i, j, k;
    double sum;
 //  Multiply each row in x by each column in y.
 //  The product of row m in x and column n in y is placed
 //  in position (m,n) in the result.
    for (i = 0; i < r1; i++)
        for (j = 0; j < c2; j++)
        {
            sum = 0;
```

```
            for (k = 0; k < c1; k++)
                sum += x[k + i*c1] * y[j + k*c2];
            r[j + i*c2] = sum;
        }
}
```

Special Requirements

- The x, y, and r data are stored in distinct arrays. That is, in-place processing is not allowed.
- All r1, c1, c2 are assumed to be > 1.
- If r1 is odd, one extra row of x[] matrix is loaded.
- If c2 is odd, one extra col of y[] matrix is loaded.
- If c1 is odd, one extra col of x[] and one extra row of y[] array is loaded.

Implementation Notes

- All three loops are unrolled two times.
- All the prolog stages of the inner-most loop (k loop) are scheduled in parallel with outer loop.
- Extraneous loads are allowed in program.
- Outer-most loop Instructions are scheduled in parallel with inner loop instructions.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	(2 * r1' * c1 * c2') + 18*(c2'/2 * r1'/2) + 40 where r1' = r1 + (r1&1) c2' = c2 + (c2&1) For r1 = 12, c1 = 14 and c2 = 12, cycles = 4720
Code size (in bytes)	960

DSPF_dp_mat_trans *Double-precision matrix transpose*

Function

void DSPF_dp_mat_trans (const double *restrict x, int rows, int cols, double *restrict r)

Arguments

x	Input matrix containing rows*cols double-precision floating-point numbers.
rows	Number of rows in matrix x. Also number of columns in matrix r.
cols	Number of columns in matrix x. Also number of rows in matrix r.
r	Output matrix containing cols*rows double-precision floating-point numbers.

Description

This function transposes the input matrix x[] and writes the result to matrix r[].

Algorithm

This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void dp_mat_trans(const double *restrict x, int rows,
           int cols, double *restrict r)
{
   int i,j;
   for(i=0; i<cols; i++)
      for(j=0; j<rows; j++)
         r[i * rows + j] = x[i + cols * j];
}
```

Special Requirements The number of rows and columns is > 0.

Implementation Notes

- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	2 * rows * cols + 15 For rows=10 and cols=20, cycles=415 For rows=15 and cols=20, cycles=615
Code size (in bytes)	256

DSPF_dp_mat_mul_cplx *Complex matrix multiplication*

Function void DSPF_dp_mat_mul_cplx (const double* x, int r1, int c1, const double* y, int c2, double* restrict r)

Arguments

x[2*r1*c1]	Input matrix containing r1*c1 complex floating-point numbers having r1 rows and c1 columns of complex numbers.
r1	Number of rows in matrix x.
c1	Number of columns in matrix x. Also number of rows in matrix y.
y[2*c1*c2]	Input matrix containing c1*c2 complex floating-point numbers having c1 rows and c2 columns of complex numbers.
c2	Number of columns in matrix y.
r[2*r1*c2]	Output matrix of c1*c2 complex floating-point numbers having c1 rows and c2 columns of complex numbers. Complex numbers are stored consecutively with real values stored in even positions and imaginary values in odd positions.

Description This function computes the expression r = x * y for the matrices x and y. The columnar dimension of x must match the row dimension of y. The resulting matrix has the same number of rows as x and the same number of columns as y.

Each element of the matrix is assumed to be complex numbers with real values are stored in even positions and imaginary values in odd positions.

Algorithm This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSPF_dp_mat_mul_cplx(const double* x, int r1, int c1,
  const double* y, int c2, double* restrict r)
{
    double real, imag;
    int i, j, k;
    for(i = 0; i < r1; i++)
    {
      for(j = 0; j < c2; j++)
      {
```

```
            real=0;
            imag=0;
            for(k = 0; k < c1; k++)
            {
            real += (x[i*2*c1 + 2*k]*y[k*2*c2 + 2*j]
            -x[i*2*c1 + 2*k + 1] * y[k*2*c2 + 2*j + 1]);
            imag+=(x[i*2*c1 + 2*k] * y[k*2*c2 + 2*j + 1]
               + x[i*2*c1 + 2*k + 1] * y[k*2*c2 + 2*j]);
            }
            r[i*2*c2 + 2*j] = real;
            r[i*2*c2 + 2*j + 1] = imag;
          }
        }
}
```

Special Requirements

- The values r1, r2 ≥ 1, c1 should be a multiple of 2 and ≥ 2.
- The x array should be padded with 6 words

Implementation Notes

- Inner-most loop is unrolled twice.
- Outer-most loop is executed in parallel with inner loops.
- Real values are stored in even word positions and imaginary values in odd positions.
- ***Endianness:*** This code is little endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	8*r1*c1*c2'+ 18*(r1*c2)+40 where c2'=2*ceil(c2/2) When r1=3, c1=4, c2=4, cycles = 640 When r1=4, c1=4, c2=5, cycles = 1040
Code size (in bytes)	832

4.2.7 Miscellaneous

DSPF_dp_blk_move *Move a block of memory*

Function void DSPF_dp_blk_move (const double * x, double *restrict r, int nx)

Arguments

x[nx]	Pointer to source data to be moved.
r[nx]	Pointer to destination array.
nx	Number of floats to move.

Description This routine moves nx floats from one memory location pointed to by x to another pointed to by r.

Algorithm This is the C equivalent of the assembly code. Note that the assembly code is hand optimized and restrictions may apply.

```
void dp_blk_move(const double * x, double *restrict r,
              int nx)
{
   int i;
   for (i = 0 ; i < nx; i++)
       r[i] = x[i];
}
```

Special Requirements The value of nx is greater than 0.

Implementation Notes

- ***Endianness:*** This implementation is little-endian.
- ***Interruptibility:*** This code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	2*nx+ 8 For nx=64, cycles=136 For nx=25, cycles=58
Code size (in bytes)	96

Appendix A

Performance/Fractional Q Formats

This appendix describes performance considerations related to the C67x DSPLIB and provides information about the Q format used by DSPLIB functions.

Topic **Page**

A.1 Performance Considerations

Although DSPLIB can be used as a first estimation of processor performance for a specific function, you should be aware that the generic nature of DSPLIB might add extra cycles not required for customer specific usage.

Benchmark cycles presented assume best-case conditions, typically assuming all code and data are placed in internal data memory. Any extra cycles due to placement of code or data in external data memory or cache-associated effects (cache hits or misses) are not considered when computing the cycle counts.

You should also be aware that execution speed in a system is dependent on where the different sections of program and data are located in memory. You should account for such differences when trying to explain why a routine is taking more time than the reported DSPLIB benchmarks.

A.2 Fractional Q Formats

Unless specifically noted, DSPLIB functions use IEEE floating point format. But few of the functions make use of fixed-point Q0.15 format also. In a Q*m*.*n* format, there are *m* bits used to represent the two's complement integer portion of the number, and *n* bits used to represent the two's complement fractional portion. *m+n+1* bits are needed to store a general Q*m*.*n* number. The extra bit is needed to store the sign of the number in the most-significant bit position. The representable integer range is specified by $(-2^m, 2^m)$ and the finest fractional resolution is 2^{-n}.

For example, the most commonly used format is Q.15. Q.15 means that a 16-bit word is used to express a signed number between positive and negative one. The most-significant binary digit is interpreted as the sign bit in any Q format number. Thus, in Q.15 format, the decimal point is placed immediately to the right of the sign bit. The fractional portion to the right of the sign bit is stored in regular two's complement format.

A.2.1 Q.15 Format

Q.15 format places the sign bit at the leftmost binary digit, and the next 15 leftmost bits contain the two's complement fractional component. The approximate allowable range of numbers in Q.15 representation is (−1,1) and the finest fractional resolution is $2^{-15} = 3.05 \times 10^{-5}$.

Table A−1. Q.15 Bit Fields

Bit	15	14	13	12	11	10	9	…	0
Value	S	Q14	Q13	Q12	Q11	Q10	Q9	…	Q0

A.3 Overview of IEEE Standard Single- and Double-Precision Formats

Floating-point operands are classified as single precision (SP) and double precision (DP). Single-precision floating-point values are 32-bit values stored in a single register. Double-precision floating-point values are 64-bit values stored in a register pair. The register pair consists of consecutive even and odd registers from the same register file. The least significant 32 bits are loaded into the even register. The most significant 32 bits containing the sign bit and exponent are loaded into the next register (which is always the odd register). The register pair syntax places the odd register first, followed by a colon, then the even register (that is, A1:A0, B1:B0, A3:A2, B3:B2, etc.).

Instructions that use DP sources fall in two categories: instructions that read the upper and lower 32-bit words on separate cycles, and instructions that read both 32-bit words on the same cycle. All instructions that produce a double-precision result write the low 32-bit word one cycle before writing the high 32-bit word. If an instruction that writes a DP result is followed by an instruction that uses the result as its DP source and it reads the upper and lower words on separate cycles, then the second instruction can be scheduled on the same cycle that the high 32-bit word of the result is written. The lower result is written on the previous cycle. This is because the second instruction reads the low word of the DP source one cycle before the high word of the DP source.

IEEE floating-point numbers consist of normal numbers, denormalized numbers, NaNs (not a number), and infinity numbers. Denormalized numbers are nonzero numbers that are smaller than the smallest nonzero normal number. Infinity is a value that represents an infinite floating-point number. NaN values represent results for invalid operations, such as (+infinity + (–infinity)).

Normal single-precision values are always accurate to at least six decimal places, sometimes up to nine decimal places. Normal double-precision values are always accurate to at least 15 decimal places, sometimes up to 17 decimal places.

Table A−2 shows notations used in discussing floating-point numbers.

Table A−2. IEEE Floating-Point Notations

Symbol	**Meaning**
s	Sign bit
e	Exponent field
f	Fraction (mantissa) field
x	Can have value of 0 or 1 (don’t care)
NaN	Not-a-Number (SNaN or QNaN)
SnaN	Signal NaN
QnaN	Quiet NaN
NaN_out	QNaN with all bits in the f field= 1
Inf	Infinity
LFPN	Largest floating-point number
SFPN	Smallest floating-point number
LDFPN	Largest denormalized floating-point number
SDFPN	Smallest denormalized floating-point number
signed Inf	+infinity or –infinity
signed NaN_out	NaN_out with s = 0 or 1

Figure A−1 shows the fields of a single-precision floating-point number represented within a 32-bit register.

Figure A−1. Single-Precision Floating-Point Fields

31	30 23	22 0
s	e	f

Legend: s sign bit (0 positive, 1 negative)
e 8-bit exponent (0 < e < 255)
f 23-bit fraction
0 < f < 1*2–1 + 1*2–2 + ... + 1*2–23 or
0 < f < ((223)–1)/(223)

The floating-point fields represent floating-point numbers within two ranges: normalized (e is between 0 and 255) and denormalized (e is 0). The following formulas define how to translate the s, e, and f fields into a single-precision floating-point number.

Normal

$-1^s * 2^{(e-127)} * 1.f$ $0 < e < 255$

Denormalized (Subnormal)

$-1^s * 2^{-126} * 0.f$ $e = 0$; f nonzero

Table A−3 shows the s, e, and f values for special single-precision floatingpoint numbers.

Table A−3. Special Single-Precision Values

Symbol	**Sign (s)**	**Exponent (e)**	**Fraction (f)**
+0	0	0	0
–0	1	0	0
+Inf	0	255	0
–Inf	1	255	0
NaN	x	255	nonzero
QnaN	x	255	1xx..x
SnaN	x	255	0xx..x and nonzero

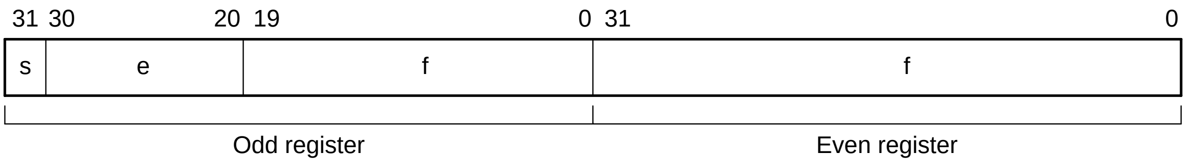
Table A−4 shows hex and decimal values for some single-precision floating-point numbers.

Table A−4. Hex and Decimal Representation for Selected Single-Precision Values

Symbol	**Hex Value**	**Decimal Value**
NaN_out	0x7FFF FFFF	QnaN
0	0x0000 0000	0.0
–0	0x8000 0000	–0.0
1	0x3F80 0000	1.0
2	0x4000 0000	2.0
LFPN	0x7F7F FFFF	3.40282347e+38
SFPN	0x0080 0000	1.17549435e–38
LDFPN	0x007F FFFF	1.17549421e–38
SDFPN	0x0000 0001	1.40129846e–45

Figure A−2 shows the fields of a double-precision floating-point number represented within a pair of 32-bit registers.

Figure A−2. Double-Precision Floating-Point Fields



Legend: s sign bit (0 positive, 1 negative)
e 11-bit exponent (0 < e < 2047)
f 52-bit fraction
0 < f < 1*2–1 + 1*2–2 + ... + 1*2–52 or
0 < f < ((252)–1)/(252)

The floating-point fields represent floating-point numbers within two ranges: normalized (e is between 0 and 2047) and denormalized (e is 0). The following formulas define how to translate the s, e, and f fields into a double-precision floating-point number.

Normal

–1s * 2(e–1023) * 1.f 0 < e < 2047

Denormalized (Subnormal)

–1s * 2–1022 * 0.f e = 0; f nonzero

Table A−5 shows the s, e, and f values for special double-precision floating-point numbers.

Table A−5. Special Double-Precision Values

Symbol	Sign (s)	Exponent (e)	Fraction (f)
+0	0	0	0
–0	1	0	0
+Inf	0	2047	0
–Inf	1	2047	0
NaN	x	2047	nonzero
QnaN	x	2047	1xx..x
SnaN	Á x	2047	0xx..x and nonzero

Table A−6 shows hex and decimal values for some double-precision floating-point numbers.

Table A−6. Hex and Decimal Representation for Selected Double-Precision Values

Symbol	Hex Value	Decimal Value
NaN_out	0x7FFF FFFF FFFF FFFF	QnaN
0	0x0000 0000 0000 0000	0.0
–0	0x8000 0000 0000 0000	–0.0
1	0x3FF0 0000 0000 0000	1.0
2	0x4000 0000 0000 0000	2.0
LFPN	0x7FEF FFFF FFFF FFFF	1.7976931348623157e+308
SFPN	0x0010 0000 0000 0000	2.2250738585072014e–308
LDFPN	0x000F FFFF FFFF FFFF	2.2250738585072009e–308
SDFPN	0x0000 0000 0000 0001	4.9406564584124654e–324

Appendix B

Software Updates and Customer Support

This appendix provides information about software updates, customer support and known issues.

Topic **Page**

B.1 DSPLIB Software Updates

C67x DSPLIB software updates may be periodically released incorporating product enhancements and fixes as they become available. You should read the README.TXT available in the root directory of every release.

B.2 DSPLIB Customer Support

If you have questions or want to report problems or suggestions regarding the C67x DSPLIB, contact Texas Instruments at dsph@ti.com.

B.3 Known Issues

These are the known issues as of Rev 2.0 of the C67x DSPLIB.

- There are some issues with how interrupts are handled in a few of the functions. So when calling the following functions you should disable interrupts before the call and restore interrupts after:
  - DSPF_sp_minval
  - DSPF_sp_maxval
  - DSPF_sp_ifftSPxSP
  - DSPF_sp_dotprod
  - DSPF_sp_fir_gen
  - DSPF_sp_fir_r2

  Failure to disable interrupts before calling these specific functions can cause corruption of the stack or sometimes a register which ultimately can lead to the application crashing.

- The DSPF_dp_vecrecip function gives incorrect results for every two alternate values of the input array.
- The DSPF_dp_biquad function results differ from the Matlab results.
- The DSPF_dp_mat_mul function in v2 of DSBLIB does not work for odd rows and columns.
- The DSPF_sp_blk_move function corrupts the next two returning lines of code.

Appendix C

Glossary

A

address: The location of program code or data stored; an individually accessible memory location.

A-law companding: See *compress and expand (compand)*.

API: See *application programming interface.*

application programming interface (API): Used for proprietary application programs to interact with communications software or to conform to protocols from another vendor's product.

assembler: A software program that creates a machine language program from a source file that contains assembly language instructions, directives, and macros. The assembler substitutes absolute operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses.

assert: To make a digital logic device pin active. If the pin is active low, then a low voltage on the pin asserts it. If the pin is active high, then a high voltage asserts it.

B

bit: A binary digit, either a 0 or 1.

big endian: An addressing protocol in which bytes are numbered from left to right within a word. More significant bytes in a word have lower numbered addresses. Endian ordering is specific to hardware and is determined at reset. See also *little endian*.

block: The three least significant bits of the program address. These correspond to the address within a fetch packet of the first instruction being addressed.

board support library (BSL): The BSL is a set of application programming interfaces (APIs) consisting of target side DSP code used to configure and control board level peripherals.

boot: The process of loading a program into program memory.

boot mode: The method of loading a program into program memory. The C6x DSP supports booting from external ROM or the host port interface (HPI).

BSL: See *board support library.*

byte: A sequence of eight adjacent bits operated upon as a unit.

C

cache: A fast storage buffer in the central processing unit of a computer.

cache controller: System component that coordinates program accesses between CPU program fetch mechanism, cache, and external memory.

central processing unit (CPU): The portion of the processor involved in arithmetic, shifting, and Boolean logic operations, as well as the generation of data- and program-memory addresses. The CPU includes the central arithmetic logic unit (CALU), the multiplier, and the auxiliary register arithmetic unit (ARAU).

chip support library (CSL): The CSL is a set of application programming interfaces (APIs) consisting of target side DSP code used to configure and control all on-chip peripherals.

clock cycle: A periodic or sequence of events based on the input from the external clock.

clock modes: Options used by the clock generator to change the internal CPU clock frequency to a fraction or multiple of the frequency of the input clock signal.

code: A set of instructions written to perform a task; a computer program or part of a program.

coder-decoder or compression/decompression (codec): A device that codes in one direction of transmission and decodes in another direction of transmission.

compiler: A computer program that translates programs in a high-level language into their assembly-language equivalents.

compress and expand (compand): A quantization scheme for audio signals in which the input signal is compressed and then, after processing, is reconstructed at the output by expansion. There are two distinct companding schemes: A-law (used in Europe) and μ-law (used in the United States).

control register: A register that contains bit fields that define the way a device operates.

control register file: A set of control registers.

CSL: See *chip support library*.

D

device ID: Configuration register that identifies each peripheral component interconnect (PCI).

digital signal processor (DSP): A semiconductor that turns analog signals—such as sound or light—into digital signals, which are discrete or discontinuous electrical impulses, so that they can be manipulated.

direct memory access (DMA): A mechanism whereby a device other than the host processor contends for and receives mastery of the memory bus so that data transfers can take place independent of the host.

DMA: See *direct memory access*.

DMA source: The module where the DMA data originates. DMA data is read from the DMA source.

DMA transfer: The process of transferring data from one part of memory to another. Each DMA transfer consists of a read bus cycle (source to DMA holding register) and a write bus cycle (DMA holding register to destination).

E

evaluation module (EVM): Board and software tools that allow the user to evaluate a specific device.

external interrupt: A hardware interrupt triggered by a specific value on a pin.

external memory interface (EMIF): Microprocessor hardware that is used to read to and write from off-chip memory.

F

fast Fourier transform (FFT): An efficient method of computing the discrete Fourier transform algorithm, which transforms functions between the time domain and the frequency domain.

fetch packet: A contiguous 8-word series of instructions fetched by the CPU and aligned on an 8-word boundary.

FFT: See *fast fourier transform.*

flag: A binary status indicator whose state indicates whether a particular condition has occurred or is in effect.

frame: An 8-word space in the cache RAMs. Each fetch packet in the cache resides in only one frame. A cache update loads a frame with the requested fetch packet. The cache contains 512 frames.

G

global interrupt enable bit (GIE): A bit in the control status register (CSR) that is used to enable or disable maskable interrupts.

H

HAL: *Hardware abstraction layer* of the CSL. The HAL underlies the service layer and provides it a set of macros and constants for manipulating the peripheral registers at the lowest level. It is a low-level symbolic interface into the hardware providing symbols that describe peripheral registers/bitfields and macros for manipulating them.

host: A device to which other devices (peripherals) are connected and that generally controls those devices.

host port interface (HPI): A parallel interface that the CPU uses to communicate with a host processor.

HPI: See *host port interface*; see also *HPI module*.

I

index: A relative offset in the program address that specifies which of the 512 frames in the cache into which the current access is mapped.

indirect addressing: An addressing mode in which an address points to another pointer rather than to the actual data; this mode is prohibited in RISC architecture.

instruction fetch packet: A group of up to eight instructions held in memory for execution by the CPU.

internal interrupt: A hardware interrupt caused by an on-chip peripheral.

interrupt: A signal sent by hardware or software to a processor requesting attention. An interrupt tells the processor to suspend its current operation, save the current task status, and perform a particular set of instructions. Interrupts communicate with the operating system and prioritize tasks to be performed.

interrupt service fetch packet (ISFP): A fetch packet used to service interrupts. If eight instructions are insufficient, the user must branch out of this block for additional interrupt service. If the delay slots of the branch do not reside within the ISFP, execution continues from execute packets in the next fetch packet (the next ISFP).

interrupt service routine (ISR): A module of code that is executed in response to a hardware or software interrupt.

interrupt service table (IST): A table containing a corresponding entry for each of the 16 physical interrupts. Each entry is a single-fetch packet and has a label associated with it.

Internal peripherals: Devices connected to and controlled by a host device. The C6x internal peripherals include the direct memory access (DMA) controller, multichannel buffered serial ports (McBSPs), host port interface (HPI), external memory-interface (EMIF), and runtime support timers.

IST: See *interrupt service table.*

L

least significant bit (LSB): The lowest-order bit in a word.

linker: A software tool that combines object files to form an object module, which can be loaded into memory and executed.

little endian: An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher-numbered addresses. Endian ordering is specific to hardware and is determined at reset. See also *big endian*.

M

maskable interrupt: A hardware interrupt that can be enabled or disabled through software.

memory map: A graphical representation of a computer system's memory, showing the locations of program space, data space, reserved space, and other memory-resident elements.

memory-mapped register: An on-chip register mapped to an address in memory. Some memory-mapped registers are mapped to data memory, and some are mapped to input/output memory.

most significant bit (MSB): The highest order bit in a word.

μ-law companding: See *compress and expand (compand)*.

multichannel buffered serial port (McBSP): An on-chip full-duplex circuit that provides direct serial communication through several channels to external serial devices.

multiplexer: A device for selecting one of several available signals.

N

nonmaskable interrupt (NMI): An interrupt that can be neither masked nor disabled.

O

object file: A file that has been assembled or linked and contains machine language object code.

off chip: A state of being external to a device.

on chip: A state of being internal to a device.

P

peripheral: A device connected to and usually controlled by a host device.

program cache: A fast memory cache for storing program instructions allowing for quick execution.

program memory: Memory accessed through the C6x's program fetch interface.

PWR: Power; see *PWR module*.

PWR module: PWR is an API module that is used to configure the power-down control registers, if applicable, and to invoke various power-down modes.

R

random-access memory (RAM): A type of memory device in which the individual locations can be accessed in any order.

register: A small area of high speed memory located within a processor or electronic device that is used for temporarily storing data or instructions. Each register is given a name, contains a few bytes of information, and is referenced by programs.

reduced-instruction-set computer (RISC): A computer whose instruction set and related decode mechanism are much simpler than those of microprogrammed complex instruction set computers. The result is a higher instruction throughput and a faster real-time interrupt service response from a smaller, cost-effective chip.

reset: A means of bringing the CPU to a known state by setting the registers and control bits to predetermined values and signaling execution to start at a specified address.

RTOS: *Real-time operating system.*

S

service layer: The top layer of the 2-layer chip support library architecture providing high-level APIs into the CSL and BSL. The service layer is where the actual APIs are defined and is the layer the user interfaces to.

synchronous-burst static random-access memory (SBSRAM): RAM whose contents does not have to be refreshed periodically. Transfer of data is at a fixed rate relative to the clock speed of the device, but the speed is increased.

synchronous dynamic random-access memory (SDRAM): RAM whose contents is refreshed periodically so the data is not lost. Transfer of data is at a fixed rate relative to the clock speed of the device.

syntax: The grammatical and structural rules of a language. All higher-level programming languages possess a formal syntax.

system software: The blanketing term used to denote collectively the chip support libraries and board support libraries.

T

tag: The 18 most significant bits of the program address. This value corresponds to the physical address of the fetch packet that is in that frame.

timer: A programmable peripheral used to generate pulses or to time events.

TIMER module: TIMER is an API module used for configuring the timer registers.

W

word: A multiple of eight bits that is operated upon as a unit. For the C6x, a word is 32 bits in length.

Index

I

K

L

M

N

O

P

Q

R

S