TMS320C64x+ DSP Little-Endian DSP Library Programmer's Reference

Literature Number: SPRUEB8B
March 2006
Revised March 2008

Texas Instruments

Preface

Read This First

About This Manual

This document describes the C64x+ digital signal processor little-endian (DSP) Library, or DSPLIB for short.

Notational Conventions

This document uses the following conventions:

- Hexadecimal numbers are shown with the suffix h. For example, the following number is 40 hexadecimal (decimal 64): 40h.
- Registers in this document are shown in figures and described in tables.

Related Documentation From Texas Instruments

The following books describe the C6000™ devices and related support tools. Copies of these documents are available on the Internet at www.ti.com. *Tip*: Enter the literature number in the search box provided at www.ti.com.

SPRU732 — *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide.* Describes the CPU architecture, pipeline, instruction set, and interrupts for the TMS320C64x and TMS320C64x+ digital signal processors (DSPs) of the TMS320C6000 DSP family. The C64x/C64x+ DSP generation comprises fixed-point devices in the C6000 DSP platform. The C64x+ DSP is an enhancement of the C64x DSP with added functionality and an expanded instruction set.

SPRAA84 — *TMS320C64x to TMS320C64+ CPU Migration Guide.* Describes migrating from the Texas Instruments TMS320C64x digital signal processor (DSP) to the TMS320C64x+ DSP. The objective of this document is to indicate differences between the two cores. Functionality in the devices that is identical is not included.

Trademarks

C6000, TMS320C64x+, TMS320C64x, C64x are trademarks of Texas Instruments.

Contents

Tables

Chapter 1

Introduction

This chapter provides a brief introduction to the TI C64x+ 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 C64x+ DSPLIB

The TI C64x+ DSPLIB is an optimized DSP Function Library for C programmers using devices that include the C64x+ megamodule. It includes many C-callable, 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 organized into the following seven different functional categories:

- Adaptive filtering
  - DSP_firlms2
- Correlation
  - DSP_autocor
- FFT
  - DSP_fft16x16
  - DSP_fft16x16_imre
  - DSP_fft16x16r
  - DSP_fft16x32
  - DSP_fft32x32
  - DSP_fft32x32s
  - DSP_ifft16x16
  - DSP_ifft16x16_imre
  - DSP_ifft16x32
  - DSP_ifft32x32
- Filtering and convolution
  - DSP_fir_cplx
  - DSP_fir_cplx_hM4X4
  - DSP_fir_gen
  - DSP_fir_gen_hM17_rA8X8
  - DSP_fir_r4
  - DSP_fir_r8
  - DSP_fir_r8_hM16_rM8A8X8
  - DSP_fir_sym
  - DSP_iir
  - DSP_iir_lat

- Math
  - DSP_dotp_sqr
  - DSP_dotprod
  - DSP_maxval
  - DSP_maxidx
  - DSP_minval
  - DSP_mul32
  - DSP_neg32
  - DSP_recip16
  - DSP_vecsumsq
  - DSP_w_vec
- Matrix
  - DSP_mat_mul
  - DSP_mat_trans
- Miscellaneous
  - DSP_bexp
  - DSP_blk_eswap16
  - DSP_blk_eswap32
  - DSP_blk_eswap64
  - DSP_blk_move
  - DSP_fltoq15
  - DSP_minerror
  - DSP_q15tofl

1.2 Features and Benefits

- Natural C Source Code
- Optimized C code with Intrinsics
- C−callable routines, fully compatible with the TI C6x compiler
- Fractional Q.15−format operands supported on some benchmarks
- Benchmarks (cycle and code size)
- Tested against C model
- The provided precompiled library was compiled using Code Generation Tools v6.0.16

1.3 Optimization Techniques

Many of the optimization techniques used throughout the DSPLIB is described in the ”Hand−Tuning Loops and Control Code on the TMS320C6000” (SPRA666) application note. For a complete list of optimization techniques please refer to the “T*MS320C6000 Optimizing Compiler v 6.0 Beta User’s Guide* (Rev. N)” (SPRU187N) user’s guide. The precompiled library that comes with the package is compiled using the source code in the “c64plus\dsplib\src\” directory and *Code Generation Tools v6.0.16*. The source code is provided to allow the user to do further optimizations for their specific application.

Chapter 2

Installing and Using DSPLIB

This chapter provides information on how to install and rebuild the TI C64x+ DSPLIB.

Topic **Page**

2.1 How to Install DSPLIB

> **Note:**
>
> You should read the README.txt file for specific details of the release.

The DSPLIB installation provides the directory structure below:

```
c64plus
|
+--dsplib
   |
   +--docs                Library documentation
   |
   +--example             Example to show DSPLIB usage
   |
   +--src                 Source code with CCS project
                           examples
   |   |+--[Kernels]
   |
   |--dsplib64plus.h      Header file containing kernel
                          definitions
   |
   |--dsplib64plus.lib    Precompiled library
   |
   |--dsplib64plus.pjt    Provided project to rebuild
                          library
   |
   |--README.txt          Top-level README file
```

After completing the installation of DSPLIB, follow the instructions in sections 2.2.2 and 2.2.3 on how to call the functions from your source code.

2.2 Using DSPLIB

2.2.1 DSPLIB Arguments and Data Types

2.2.1.1 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	−0.9999694824...	0.9999694824...
Q.31	32	fraction	−0.99999999953...	0.99999999953...
IEEE float	32	floating point	1.17549435e−38	3.40282347e+38
IEEE double	64	floating point	2.2250738585072014e−308	1.7976931348623157e+308

Unless specifically noted, DSPLIB operates on Q.15-fractional data type elements. Appendix A presents an overview of Fractional Q formats.

2.2.1.2 DSPLIB Arguments

TI DSPLIB functions typically operate over vector operands for greater efficiency. Even though these routines can be used to process short arrays, or even scalars (unless a minimum size requirement is noted), they will be slower for those 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, unless specifically noted.
- In-place computation is not allowed, unless specifically noted: Source operand cannot be equal to destination operand.

2.2.2 Calling a DSPLIB Function From C

In addition to installing the DSPLIB software, follow these steps to include a DSPLIB function to the code:

- Include the dsplib64plus.h header file
- Link the code with the dsplib64plus.lib
- Use a correct linker command file for the platform used

The source code and CCS project in the “c64plus\dsplib\example\” directory shows how to use the DSPLIB in a Code Composer Studio C environment.

2.2.3 Calling a DSP Function From Assembly

The C64x+ 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 C64x+ C compiler calling conventions. For more information, see Section 8 (Runtime Environment) of *TMS320C6000 Optimizing C Compiler User’s Guide* (SPRU187).

2.2.4 DSPLIB Testing − Allowable Error

DSPLIB is tested under the Code Composer Studio environment against a reference C implementation. You can expect identical results between Reference C implementation and its Optimized C implementation when using test routines that focus on fixed point type results.

2.2.5 DSPLIB Overflow and Scaling Issues

The DSPLIB functions implement the same functionality of the reference C code. You must conform to the range requirements specified in the API function, and in addition, restrict the input range so that the outputs do not overflow.

In FFT functions, twiddle factors are generated with a fixed scale factor; i.e., 32767($=2^{15-1}$) for all 16-bit FFT functions, 1073741823($=2^{30-1}$) for DSP_fft32x32s, 2147483647($=2^{31-1}$) for all other 32-bit FFT functions. Because DSP_fft16x16r and DSP_fft32x32s perform scaling by 2 at each radix-4 stage, the input data must be scaled by $2^{(\log2(nx)-cei[\log4(nx)-1])}$ to completely prevent overflow. In all other FFT functions, the input data must be scaled by $2^{(\log2(nx))}$ because no scaling is done by the functions.

2.2.6 Interrupt Behavior of DSPLIB Functions

All of the functions in this library are designed to be used in systems with interrupts. Thus, it is not necessary to disable interrupts when calling any of these functions. The functions in the library will disable interrupts as needed to protect the execution of code in tight loops and so on. Library functions have three categories:

- **Fully-interruptible:** These functions do not disable interrupts. Interrupts are blocked by at most 5 to 10 cycles at a time (not counting stalls) by branch delay slots.
- **Partially-interruptible:** These functions disable interrupts for long periods of time, with small windows of interruptibility. Examples include a function with a nested loop, where the inner loop is non-interruptible and the outer loop permits interrupts between executions of the inner loop.
- **Non-interruptible:** These functions disable interrupts for nearly their entire duration. Interrupts may happen for a short time during the setup and exit sequence.

Note that all three function categories tolerate interrupts. That is, an interrupt can occur at any time without affecting the function correctness. The interruptibility of the function only determines how long the kernel might delay the processing of the interrupt.

2.3 How to Rebuild DSPLIB

If you would like to rebuild DSPLIB, follow these steps:

1. Start Code Composer Studio (version 3.2 or later).
2. Open the 'dsplib64plus.pjt' project by clicking Project −> Open in the menu and browse to the 'c64plus\dsplib\' directory.
3. Rebuild the project by clicking on Project−> Build in the menu.

Once CCS has finished compiling the new library you will find the file 'dsplib64plus_rebuild.lib' in the 'c64plus\dsplib\Release\' directory.

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 used 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)

Some C64x+ functions have additional restrictions due to optimization using new features such as higher multiply throughput. While these new functions perform better, they can also lead to problems if not carefully used. Therefore, the new functions are named with any additional restrictions. Three types of restrictions are specified to a pointer: minimum buffer size (M), buffer alignment (A), and the number of elements in the buffer to be a multiple of an integer (X).The following convention has been used when describing the arguments for each individual function:

A kernel function `foo` with two parameters, m and n, with the following restrictions:

m −> Minimum buffer size = 8, buffer alignment = double word, buffer needs to be a multiple of 8 elements

n −> Minimum buffer size = 32, buffer alignment = word , buffer needs to be a multiple of 16 elements

This function would be named: `foo_mM8A8X8_nM32A4X16.`

3.2 DSPLIB Functions

The routines included in the DSP library are organized into seven functional categories and listed below in alphabetical order.

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

3.3 DSPLIB Function Tables

Table 3−2. Adaptive Filtering

Functions	Description	Page
long DSP_firlms2(short *h, short *x, short b, int nh)	LMS FIR	4-2

Table 3−3. Correlation

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

Table 3−4. FFT

Functions	Description	Page
void DSP_fft16x16(short *w, int nx, short *x, short *y)	Complex out of place, Forward FFT mixed radix with digit reversal. Input/Output data in Re/Im order.	4-8
void DSP_fft16x16_imre(short *w, int nx, short *x, short *y)	Complex out of place, Forward FFT mixed radix with digit reversal. Input/Output data in Im/Re order.	4-11
void DSP_fft16x16r(int nx, short *x, short *w, unsigned char *brev, short *y, int offset, int n_max)	Mixed radix FFT with scaling and rounding, digit reversal, out of place. Input and output: 16 bits, Twiddle factor: 16 bits.	4-12
void DSP_fft16x32(short *w, int nx, int *x, int *y)	Extended precision, mixed radix FFT, rounding, digit reversal, out of place. Input and output: 32 bits, Twiddle factor: 16 bits.	4-22
void DSP_fft32x32(int *w, int nx, int *x, int *y)	Extended precision, mixed radix FFT, rounding, digit reversal, out of place. Input and output: 32 bits, Twiddle factor: 32 bits.	4-24
void DSP_fft32x32s(int *w, int nx, int *x, int *y)	Extended precision, mixed radix FFT, digit reversal, out of place., with scaling and rounding. Input and output: 32 bits, Twiddle factor: 32 bits.	4-26

Table 3−4. FFT (Continued)

Functions	Description	Page
void DSP_ifft16x16(short *w, int nx, short *x, short *y)	Complex out of place, Inverse FFT mixed radix with digit reversal. Input/Output data in Re/Im order.	4-26
void DSP_ifft16x16_imre(short *w, int nx, short *x, short *y)	Complex out of place, Inverse FFT mixed radix with digit reversal. Input/Output data in Re/Im order.	4-26
void DSP_ifft16x32(short *w, int nx, int *x, int *y)	Extended precision, mixed radix IFFT, rounding, digit reversal, out of place. Input and output: 32 bits, Twiddle factor: 16 bits.	4-32
void DSP_ifft32x32(int *w, int nx, int *x, int *y)	Extended precision, mixed radix IFFT, digit reversal, out of place, with scaling and rounding. Input and output: 32 bits, Twiddle factor: 32 bits.	4-34

Table 3−5. Filtering and Convolution

Functions	Description	Page
void DSP_fir_cplx(short *x, short *h, short *r, int nh, int nr)	Complex FIR Filter (nh is a multiple of 2)	4-36
void DSP_fir_cplx_hM4X4(short *x, short *h, short *r, int nh, int nr)	Complex FIR Filter (nh is a multiple of 4)	4-36
void DSP_fir_gen (short *x, short *h, short *r, int nh, int nr)	FIR Filter (any nh)	4-40
void DSP_fir_gen_hM17_rA8X8 (short *x, short *h, short *r, int nh, int nr)	FIR Filter (r[] must be double word aligned, nr must be multiple of 8)	4-40
void DSP_fir_r4 (short *x, short *h, short *r, int nh, int nr)	FIR Filter (nh is a multiple of 4)	4-44
void DSP_fir_r8 (short *x, short *h, short *r, int nh, int nr)	FIR Filter (nh is a multiple of 8)	4-48
void DSP_fir_r8_hM16_rM8A8X8 (short *x, short *h, short *r, int nh, int nr)	FIR Filter (r[] must be double word aligned, nr is a multiple of 8)	4-48
void DSP_fir_sym (short *x, short *h, short *r, int nh, int nr, int s)	Symmetric FIR Filter (nh is a multiple of 8)	4-50

Table 3−5. Filtering and Convolution (Continued)

Functions	Description	Page
void DSP_iir(short Input, short *Coefs, int nCoefs, short State)	IIR Filter	4-52
void DSP_iir_lat(short *x, int nx, short *k, int nk, int *b, short *r)	All−pole IIR Lattice Filter	4-54

Table 3−6. Math

Functions	Description	Page
int DSP_dotp_sqr(int G, short *x, short *y, int *r, int nx)	Vector Dot Product and Square	4-56
int DSP_dotprod(short *x, short *y, int nx)	Vector Dot Product	4-58
short DSP_maxval (short *x, int nx)	Maximum Value of a Vector	4-60
int DSP_maxidx (short *x, int nx)	Index of the Maximum Element of a Vector	4-61
short DSP_minval (short *x, int nx)	Minimum Value of a Vector	4-63
void DSP_mul32(int *x, int *y, int *r, short nx)	32-bit Vector Multiply	4-64
void DSP_neg32(int *x, int *r, short nx)	32-bit Vector Negate	4-66
void DSP_recip16 (short *x, short *rfrac, short *rexp, short nx)	16-bit Reciprocal	4-67
int DSP_vecsumsq (short *x, int nx)	Sum of Squares	4-69
void DSP_w_vec(short *x, short *y, short m, short *r, short nr)	Weighted Vector Sum	4-70

Table 3−7. Matrix

Functions	Description	Page
void DSP_mat_mul(short *x, int r1, int c1, short *y, int c2, short *r, int qs)	Matrix Multiplication	4-71
void DSP_mat_trans(short *x, short rows, short columns, short *r)	Matrix Transpose	4-73

Table 3−8. Miscellaneous

Functions	Description	Page
short DSP_bexp(int *x, short nx)	Max Exponent of a Vector (for scaling)	4-74
void DSP_blk_eswap16(void *x, void *r, int nx)	Endian-swap a block of 16-bit values	4-76
void DSP_blk_eswap32(void *x, void *r, int nx)	Endian-swap a block of 32-bit values	4-78
void DSP_blk_eswap64(void *x, void *r, int nx)	Endian-swap a block of 64-bit values	4-80
void DSP_blk_move(short *x, short *r, int nx)	Move a Block of Memory	4-82
void DSP_fltoq15 (float *x,short *r, short nx)	Float to Q15 Conversion	4-83
int DSP_minerror (short *GSP0_TABLE,short *errCoefs, int *savePtr_ret)	Minimum Energy Error Search	4-85
void DSP_q15tofl (short *x, float *r, short nx)	Q15 to Float Conversion	4-87

Chapter 4

DSPLIB Reference

This chapter provides a list of the 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 Adaptive Filtering

DSP_firlms2 *LMS FIR*

Function long DSP_firlms2(short * restrict h, short * restrict x, short b, int nh)

Arguments

h[nh]	Coefficient Array
x[nh+1]	Input Array
b	Error from previous FIR
nh	Number of coefficients. Must be multiple of 4.
return long	Return value

Description The Least Mean Square Adaptive Filter computes an update of all nh coefficients by adding the weighted error times the inputs to the original coefficients. The input array includes the last nh inputs followed by a new single sample input. The coefficient array includes nh coefficients.

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

```
long DSP_firlms2(short h[ ],short x[ ], short b,
int nh)
{
   int              i;
   long          r = 0;
   for (i = 0; i < nh; i++) {
      h[i] += (x[i] * b) >> 15;
      r += x[i + 1] * h[i];
   }
   return r;
}
```

Special Requirements

- ❏ This routine assumes 16-bit input and output.
- ❏ The number of coefficients nh must be a multiple of 4.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.
- The loop is unrolled 4 times.

Benchmarks

Cycles	3 * nh/4 + 11
Codesize	192 bytes

4.2 Correlation

DSP_autocor *AutoCorrelation*

Function void DSP_autocor(short * restrict r, short * restrict x, int nx, int nr)

Arguments

r[nr]	Output array
x[nx+nr]	Input array. Must be double-word aligned.
nx	Length of autocorrelation. Must be a multiple of 8.
nr	Number of lags. Must be a multiple of 4.

Description This routine accepts an input array of length nx + nr and performs nr autocorrelations each of length nx producing nr output results. This is typically used in VSELP code.

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

```
void DSP_autocor(short r[ ],short x[ ], int nx, int nr)
{
    int i,k,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 >> 15);
    }
}
```

Special Requirements

- ❏ nx must be a multiple of 8.
- ❏ nr must be a multiple of 4.
- ❏ x[ ] must be double-word aligned.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.

Benchmarks

Cycles	5/32 * nr * nx + 5 * nr + 20
Codesize	512 bytes

4.3 FFT

DSP_fft16x16 *Complex Forward Mixed Radix 16 x 16-bit FFT*

Function void DSP_fft16x16(short * restrict w, int nx, short * restrict x, short * restrict y)

Arguments

w[2*nx]	Pointer to complex Q.15 FFT coefficients.
nx	Length of FFT in complex samples. Must be a power of 2 or 4, and 16 ≤ nx ≤ 65536.
x[2*nx]	Pointer to complex 16-bit data input.
y[2*nx]	Pointer to complex 16-bit data output.

Description This routine computes a complex forward mixed radix FFT with rounding and digit reversal. Input data x[ ], output data y[ ], and coefficients w[ ] are 16-bit. The output is returned in the separate array y[ ] in normal order. Each complex value is stored with interleaved real and imaginary parts. The code uses a special ordering of FFT coefficients (also called twiddle factors) and memory accesses to improve performance in the presence of cache.

Algorithm All stages are radix-4 except the last one, which can be radix-2 or radix-4, depending on the size of the FFT. All stages except the last one scale by two the stage output data.

Special Requirements

- ❏ In-place computation is *not* allowed.
- ❏ The size of the FFT, nx, must be a power of 2 or 4, and 16 ≤ nx ≤ 65536.
- ❏ The arrays for the complex input data x[ ], complex output data y[ ], and twiddle factors w[ ] must be double-word aligned.
- ❏ The input and output data are complex, with the real/imaginary components stored in adjacent locations in the array. The real components are stored at even array indices, and the imaginary components are stored at odd array indices. All data are in short precision or Q.15 format.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.

The routine uses $\log_4(nx) - 1$ stages of radix-4 transform and performs either a radix-2 or radix-4 transform on the last stage depending on nx. If nx is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform. The conventional Cooley Tukey FFT is written using three loops. The outermost loop “k” cycles through the stages. There are log N to the base 4 stages in all. The loop “j” cycles through the groups of butterflies with different twiddle factors, and loop “i” reuses the twiddle factors for the different butterflies within a stage. Note the following:

Stage	Groups	Butterflies With Common Twiddle Factors	Groups*Butterflies
1	N/4	1	N/4
2	N/16	4	N/4
..	..	..	..
logN	1	N/4	N/4

The following statements can be made based on above observations:

1) Inner loop “i0” iterates a variable number of times. In particular, the number of iterations quadruples every time from 1..N/4. Hence, software pipelining a loop that iterates a variable number of times is not profitable.

2) Outer loop “j” iterates a variable number of times as well. However, the number of iterations is quartered every time from N/4 ..1. Hence, the behavior in (a) and (b) are exactly opposite to each other.

3) If the two loops “i” and “j” are coalesced together then they will iterate for a fixed number of times, namely N/4. This allows us to combine the “i” and “j” loops into one loop. Optimized implementations will make use of this fact.

In addition,, the Cooley Tukey FFT accesses three twiddle factors per iteration of the inner loop, as the butterflies that reuse twiddle factors are lumped together. This leads to accessing the twiddle factor array at three points, each separated by “ie”. Note that “ie” is initially 1, and is quadrupled with every iteration. Therefore, these three twiddle factors are not even contiguous in the array.

To vectorize the FFT, it is desirable to access the twiddle factor array using double word wide loads and fetch the twiddle factors needed. To do this, a modified twiddle factor array is created, in which the factors WN/4, WN/2, W3N/4 are arranged to be contiguous. This eliminates the separation between twiddle factors within a butterfly. However, this implies that we maintain a redundant version of the twiddle factor array as the loop is traversed from one stage to another. Hence, the size of the twiddle factor array increases as compared to the normal Cooley Tukey FFT. The modified twiddle factor array is of size “2 * N” where the conventional Cooley Tukey FFT is of size “3N/4” where N is the number of complex points to be transformed. The routine that generates the modified twiddle factor array was presented earlier. With the above transformation of the FFT, both the input data and the twiddle factor array can be accessed using double-word wide loads to enable packed data processing.

The final stage is optimized to remove the multiplication as w0 = 1. This stage also performs digit reversal on the data, so the final output is in natural order. In addition, if the number of points to be transformed is a power of 2, the final stage applies a radix-2 pass instead of a radix-4. In any case, the outputs are returned in normal order.

The code performs the bulk of the computation in place. However, because digit-reversal cannot be performed in-place, the final result is written to a separate array, y[].

Benchmarks

Codesize 0x300 bytes

MIPS (CPU Cycles)			
N	**Nat C**	**INT C**	**SA**
16	303	106	104
32	644	177	162
64	1146	283	242
128	2789	588	518
256	5427	1158	934
512	13214	2559	2170
1024	26268	5369	4218
2048	62375	11922	9870
4096	124581	25228	19598
8192	288944	55461	45218
16384	577710	116895	90274

DSP_fft16x16_imre *Complex Forward Mixed Radix 16 x 16-bit FFT, With Im/Re Order*

Function

void DSP_fft16x16_imre(short * restrict w, int nx, short * restrict x, short * restrict y)

Arguments

w[2*nx] Pointer to complex Q.15 FFT coefficients.

nx Length of FFT in complex samples. Must be a power of 2 or 4, and $16 \le nx \le 65536$.

x[2*nx] Pointer to complex 16-bit data input.

y[2*nx] Pointer to complex 16-bit data output.

Description

This routine computes a complex forward mixed radix FFT with rounding and digit reversal. Input data x[ ], output data y[ ], and coefficients w[ ] are 16-bit. The output is returned in the separate array y[ ] in normal order. Each complex value is stored with interleaved **imaginary** and **real** parts. The code uses a special ordering of FFT coefficients (also called twiddle factors) and memory accesses to improve performance in the presence of cache.

Algorithm

All stages are radix-4 except the last one, which can be radix-2 or radix-4, depending on the size of the FFT. All stages except the last one scale by two the stage output data.

Special Requirements

- ❏ In-place computation is *not* allowed.
- ❏ The size of the FFT, nx, must be power a of 2 or 4, and $16 \le nx \le 65536$.
- ❏ The arrays for the complex input data x[ ], complex output data y[ ], and twiddle factors w[ ] must be double-word aligned.
- ❏ The input and output data are complex, with the **imaginary/real** components stored in adjacent locations in the array. The imaginary components are stored at even array indices, and the real components are stored at odd array indices. All data are in short precision or Q.15 format.

Implementation Notes

- ❏ ***Interruptibility:*** The code is interruptible.

The routine uses $\log_4(nx) - 1$ stages of radix-4 transform and performs either a radix-2 or radix-4 transform on the last stage depending on nx. If nx is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform. The conventional Cooley Tukey FFT is written using three loops. The outermost loop “k” cycles through the stages. There are log N to the base 4 stages in all. The loop “j” cycles through the groups of butterflies with different twiddle factors, and loop “i” reuses the twiddle factors for the different butterflies within a stage. Note the following:

Stage	Groups	Butterflies With Common Twiddle Factors	Groups*Butterflies
1	N/4	1	N/4
2	N/16	4	N/4
..	..	..	..
logN	1	N/4	N/4

The following statements can be made based on above observations:

1) Inner loop “i0” iterates a variable number of times. In particular, the number of iterations quadruples every time from 1..N/4. Hence, software pipelining a loop that iterates a variable number of times is not profitable.

2) Outer loop “j” iterates a variable number of times as well. However, the number of iterations is quartered every time from N/4 ..1. Hence, the behavior in (a) and (b) are exactly opposite to each other.

3) If the two loops “i” and “j” are coalesced together then they will iterate for a fixed number of times, namely N/4. This allows us to combine the “i” and “j” loops into one loop. Optimized implementations will make use of this fact.

In addition, the Cooley Tukey FFT accesses three twiddle factors per iteration of the inner loop, as the butterflies that reuse twiddle factors are lumped together. This leads to accessing the twiddle factor array at three points, each separated by “ie”. Note that “ie” is initially 1, and is quadrupled with every iteration. Therefore these three twiddle factors are not even contiguous in the array.

To vectorize the FFT, it is desirable to access twiddle factor array using double word wide loads and fetch the twiddle factors needed. To do this, a modified twiddle factor array is created, in which the factors WN/4, WN/2, W3N/4 are arranged to be contiguous. This eliminates the separation between twiddle factors within a butterfly. However, this implies that we maintain a redundant version of the twiddle factor array as the loop is traversed from one stage to another. Hence, the size of the twiddle factor array increases as compared to the normal Cooley Tukey FFT. The modified twiddle factor array is of size “2 * N”, where the conventional Cooley Tukey FFT is of size “3N/4”, where N is the number of complex points to be transformed. The routine that generates the modified twiddle factor array was presented earlier. With the above transformation of the FFT, both the input data and the twiddle factor array can be accessed using double-word wide loads to enable packed data processing.

The final stage is optimized to remove the multiplication as w0 = 1. This stage also performs digit reversal on the data, so the final output is in natural order. In addition, if the number of points to be transformed is a power of 2, the final stage applies a DSP_radix2 pass instead of a radix 4. In any case, the outputs are returned in normal order.

The code performs the bulk of the computation in place. However, because digit-reversal cannot be performed in-place, the final result is written to a separate array, y[].

Benchmarks

Codesize 0x2E0 bytes

MIPS (CPU Cycles)			
N	**Nat C**	**INT C**	**SA**
16	316	131	104
32	688	194	162
64	1246	322	242
128	3046	669	518
256	5956	1325	934
512	14476	2944	2170
1024	28810	6144	4218
2048	68242	13715	9870
4096	136336	28819	19598
8192	315544	63654	45218
16384	630934	133286	90274

DSP_fft16x16r *Complex Forward Mixed Radix 16 x 16-bit FFT With Rounding*

Function

void DSP_fft16x16r(int nx, short * restrict x, short * restrict w, short * restrict y, int radix, int offset, int nmax)

Arguments

nx	Length of FFT in complex samples. Must be power of 2 or 4, and $16 \le nx \le 65536$
x[2*nx]	Pointer to complex 16-bit data input
w[2*nx]	Pointer to complex FFT coefficients
y[2*nx]	Pointer to complex 16-bit data output
radix	Smallest FFT butterfly used in computation used for decomposing FFT into sub-FFTs. See notes.
offset	Index in complex samples of sub-FFT from start of main FFT.
nmax	Size of main FFT in complex samples.

Description

This routine implements a complex forward mixed radix FFT with scaling, rounding and digit reversal. Input data x[ ], output data y[ ], and coefficients w[ ] are 16-bit. The output is returned in the separate array y[ ] in normal order. Each complex value is stored as interleaved 16-bit real and imaginary parts. The code uses a special ordering of FFT coefficients (also called twiddle factors).

This redundant set of twiddle factors is size 2*N short samples. As pointed out in subsequent sections, dividing these twiddle factors by 2 will give an effective divide by 4 at each stage to guarantee no overflow. The function is accurate to about 68dB of signal to noise ratio to the DFT function as follows.

```
void dft(int n, short x[], short y[])
{
   int k,i, index;
   const double PI = 3.14159654;
   short * p_x;
   double 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 = (double)p_x[0];
       fx_1 = (double)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] = (short)2*fy_0/sqrt(n);
     y[2*k+1] = (short)2*fy_1/sqrt(n);
   }
}
```

Scaling by 2 (i.e., >>1) takes place at each radix-4 stage except the last one. A radix-4 stage could give a maximum bit-growth of 2 bits, which would require scaling by 4. To completely prevent overflow, the input data must be scaled by $2^{(BT-BS)}$, where BT (total number of bit growth) = $\log_2(nx)$ and BS (number of scales by the functions) = ceil[$\log_4(nx)-1$]. All shifts are rounded to reduce truncation noise power by 3dB.

The function takes the twiddle factors and input data, and calculates the FFT producing the frequency domain data in the y[ ] array. As the FFT allows every input point to affect every output point, which causes cache thrashing in a cache based system. 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, see the following function:

```
DSP_fft16x16r(1024,&x[0],    &w[0],    y,4,    0,1024);
```

is equivalent to:

```
DSP_fft16x16r(1024,&x[2*0],  &w[0]    ,y,256,  0,1024);
DSP_fft16x16r(256, &x[2*0],  &w[2*768],y,4,    0,1024);
DSP_fft16x16r(256, &x[2*256],&w[2*768],y,4,  256,1024);
DSP_fft16x16r(256, &x[2*512],&w[2*768],y,4,  512,1024);
DSP_fft16x16r(256, &x[2*768],&w[2*768],y,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-point 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, the following would be needed to decompose the FFT:

```
DSP_fft16x16r(512, &x[0],    &w[0],    y,2,    0,512);
```

is equivalent to:

```
DSP_fft16x16r(512, &x[0],    &w[0],    y,128,  0,512);
DSP_fft16x16r(128, &x[2*0],  &w[2*384],y,2,    0,512);
DSP_fft16x16r(128, &x[2*128],&w[2*384],y,2,  128,512);
DSP_fft16x16r(128, &x[2*256],&w[2*384],y,2,  256,512);
DSP_fft16x16r(128, &x[2*384],&w[2*384],y,2,  384,512);
```

The twiddle factor array is composed of $\log_4(N)$ 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.

```
DSP_fft16x16r(N,  &x[0],      &w[0],      y,N/4,0,    N)
```

```
DSP_fft16x16r(N/4,&x[0],      &w[2*3*N/4],y,rad,0,    N)
DSP_fft16x16r(N/4,&x[2*N/4],  &w[2*3*N/4],y,rad,N/4,  N)
DSP_fft16x16r(N/4,&x[2*N/2],  &w[2*3*N/4],y,rad,N/2,  N)
DSP_fft16x16r(N/4,&x[2*3*N/4],&w[2*3*N/4],y,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” controls how many stages of decomposition are performed. It also determines whether a radix4 or DSP_radix2 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, thus, 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:

```
DSP_fft16x16r(N,  &x[0],      &w[0],      y,N/4,0,     N)
DSP_fft16x16r(N/4,&x[2*3*N/4],&w[2*3*N/4],y,rad,3*N/4, N)
DSP_fft16x16r(N/4,&x[2*N/2],  &w[2*3*N/4],y,rad,N/2,   N)
DSP_fft16x16r(N/4,&x[2*N/4],  &w[2*3*N/4],y,rad,N/4,   N)
DSP_fft16x16r(N/4,&x[0],      &w[2*3*N/4],y,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:

```
DSP_fft16x16r(N, &x[0], &w[0], y, rad, 0, N)
```

Algorithm

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

```
void fft16x16r
(
    int             n,
    short           *ptr_x,
    short           *ptr_w,
    short           *y,
    int             radix,
    int             offset,
    int             nmax
)
{
    int   i, l0, l1, l2, h2, predj;
    int   l1p1,l2p1,h2p1, tw_offset, stride, fft_jmp;
```

```
    short xt0, yt0, xt1, yt1, xt2, yt2;
    short si1,si2,si3,co1,co2,co3;
    short xh0,xh1,xh20,xh21,xl0,xl1,xl20,xl21;
    short x_0, x_1, x_l1, x_l1p1, x_h2 , x_h2p1, x_l2, x_l2p1;
    short *x,*w;
    short *ptr_x0, *ptr_x2, *y0;
    unsigned int j, k, j0, j1, k0, k1;
    short x0, x1, x2, x3, x4, x5, x6, x7;
    short xh0_0, xh1_0, xh0_1, xh1_1;
    short xl0_0, xl1_0, xl0_1, xl1_1;
    short yt3, yt4, yt5, yt6, yt7;
    unsigned a, num;
    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>>1; i += 2)
        {
            co1 = w[j+0];
            si1 = w[j+1];
            co2 = w[j+2];
            si2 = w[j+3];
            co3 = w[j+4];
            si3 = w[j+5];
            j += 6;
            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] = ((short)(xh0 + xh20))>>1;
  ptr_x0[1] = ((short)(xh1 + xh21))>>1;
  ptr_x2 = ptr_x0;
  x += 2;
  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;
  l1p1 = l1+1;
  h2p1 = h2+1;
  l2p1 = l2+1;
  ptr_x2[l1  ] = (xt1 * co1 + yt1 * si1 +
0x00008000) >> 16;
  ptr_x2[l1p1] = (yt1 * co1 - xt1 * si1 +
0x00008000) >> 16;
  ptr_x2[h2  ] = (xt0 * co2 + yt0 * si2 +
 0x00008000) >> 16;
```

```
        ptr_x2[h2p1] = (yt0 * co2 - xt0 * si2 +
      x00008000) >> 16;
        ptr_x2[l2  ] = (xt2 * co3 + yt2 * si3 +
      0x00008000) >> 16;
        ptr_x2[l2p1] = (yt2 * co3 - xt2 * si3 +
      0x00008000) >> 16;
    }
    tw_offset += fft_jmp;
    stride = stride>>2;
} /* end while */
j = offset>>2;
ptr_x0 = ptr_x;
y0 = y;
/* determine _norm(nmax) - 17 */
l0 = 31;
if (((nmax>>31)&1)==1)
    num = ~nmax;
else
    num = nmax;
if (!num)
    l0 = 32;
else
{
    a=num&0xFFFF0000; if (a) { l0-=16; num=a; }
    a=num&0xFF00FF00; if (a) { l0-= 8; num=a; }
    a=num&0xF0F0F0F0; if (a) { l0-= 4; num=a; }
    a=num&0xCCCCCCCC; if (a) { l0-= 2; num=a; }
    a=num&0xAAAAAAAA; if (a) { l0-= 1; }
}
l0 -= 1;
l0 -= 17;
if(radix == 2 || radix  == 4)
    for (i = 0; i < n; i += 4)
    {
            /* reversal computation */
            j0 = (j     ) & 0x3F;
            j1 = (j >> 6) & 0x3F;
```

```
k0 = _bitr(j0) >> 10;
k1 = _bitr(j1) >> 10;
k = (k0 << 6) |  k1;
if (l0 < 0)
  k = k << -l0;
else
  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>>1;
                  y0[k] = yt2; y0[k+1] = yt3;
                  k += n>>1;
                  y0[k] = yt4; y0[k+1] = yt5;
                  k += n>>1;
                  y0[k] = yt6; y0[k+1] = yt7;
            }
}
```

Special Requirements

- In-place computation is *not* allowed.
- nx must be a power of 2 or 4.
- Complex input data x[ ], twiddle factors w[ ], and output array y[ ] must be double-word aligned.
- Real values are stored in even word, imaginary in odd.
- All data are in short precision or Q.15 format. Allowed input dynamic range is $16 - (\log_2(nx) - \text{ceil}[\log_4(nx) - 1])$.
- Output results are returned in normal order.
- The FFT coefficients (twiddle factors) are generated using the function gen_twiddle_fft16x16 provided in the 'c64plus\dsplib\src\DSP_fft16x16r'

directory. The scale factor must be 32767.5. The input data must be scaled by $2^{(\log2(nx)-ceil[\log4(nx)-1])}$ to completely prevent overflow.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.
- The routine uses $\log_4(nx) - 1$ stages of radix-4 transform and performs either a radix-2 or radix-4 transform on the last stage depending on nx. If nx is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform.
- A special sequence of coefficients used as generated above produces the FFT. This collapses the inner 2 loops in the traditional Burrus and Parks implementation.
- 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 butterfly is bit reversed; i.e. the inner 2 points of the butterfly are crossed over. This makes the data come out in bit reversed rather than in radix 4 digit reversed order. This simplifies the last pass of the loop. The BITR instruction does the bit reversal out of place.

Benchmarks

Codesize 0x240 bytes

MIPS (CPU Cycles)			
N	**Nat C**	**INT C**	**SA**
16	293	126	103
32	625	199	175
64	1111	336	263
128	2721	715	557
256	5295	1396	1005
512	12953	3199	2307
1024	25751	6536	4483
2048	61345	14995	10393
4096	122527	30876	20633
8192	284841	69799	47279
16384	569511	143536	94383

DSP_fft16x32 *Complex Forward Mixed Radix 16 x 32-bit FFT With Rounding*

Function void DSP_fft16x32(short * restrict w, int nx, int * restrict x, int * restrict y)

Arguments

w[2*nx]	Pointer to complex Q.15 FFT coefficients.
nx	Length of FFT in complex samples. Must be power of 2 or 4, and $16 \le nx \le 65536$.
x[2*nx]	Pointer to complex 32-bit data input.
y[2*nx]	Pointer to complex 32-bit data output.

Description This routine computes an extended precision complex forward mixed radix FFT with rounding and digit reversal. Input data x[ ] and output data y[ ] are 32-bit, coefficients w[ ] are 16-bit. The output is returned in the separate array y[ ] in normal order. Each complex value is stored with interleaved real and imaginary parts. The code uses a special ordering of FFT coefficients (also called twiddle factors) and memory accesses to improve performance in the presence of cache. The C code to generate the twiddle factors is provided with this library in the 'c64plus\dsplib\src\DSP_fft16x32' directory.

Algorithm For further details, see the source code of the C and Optimized C version of this function that is provided in the 'c64plus\dsplib\src\DSP_fft16x32' directory.

Special Requirements

- ❏ In-place computation is *not* allowed.
- ❏ The size of the FFT, nx, must be a power of 2 or 4 and $16 \le nx \le 65536$.
- ❏ The arrays for the complex input data x[ ], complex output data y[ ], and twiddle factors w[ ] must be double-word aligned.
- ❏ The input and output data are complex, with the real/imaginary components stored in adjacent locations in the array. The real components are stored at even array indices, and the imaginary components are stored at odd array indices.
- ❏ The FFT coefficients (twiddle factors) are generated using the function gen_twiddle_fft16x32 provided in the 'c64plus\dsplib\src\DSP_fft16x32' directory. The scale factor must be 32767.5. No scaling is done with the function; thus the input data must be scaled by $2^{(\log2(nx)-\text{ceil}[\log4(nx)-1])}$ to completely prevent overflow.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.
- The routine uses $\log_4(nx) - 1$ stages of radix-4 transform and performs either a radix-2 or radix-4 transform on the last stage depending on nx. If nx is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform.
- See the fft16x16t implementation notes, as similar ideas are used.

Benchmarks

Codesize 0x5A0

MIPS (CPU Cycles)			
N	**Nat C**	**INT C**	**SA**
16	372	161	140
32	884	296	246
64	1622	457	382
128	4182	978	857
256	8212	1827	1577
512	20572	4204	3748
1024	40986	8429	7332
2048	98978	19478	17343
4096	197792	39703	34495
8192	464040	90432	80090
16384	927910	184641	159962

DSP_fft32x32 *Complex Forward Mixed Radix 32 x 32-bit FFT With Rounding*

Function void DSP_fft32x32(int * restrict w, int nx, int * restrict x, int * restrict y)

Arguments

w[2*nx]	Pointer to complex 32-bit FFT coefficients.
nx	Length of FFT in complex samples. Must be power of 2 or 4, and $16 \le nx \le 65536$.
x[2*nx]	Pointer to complex 32-bit data input.
y[2*nx]	Pointer to complex 32-bit data output.

Description This routine computes an extended precision complex forward mixed radix FFT with rounding and digit reversal. Input data x[ ], output data y[ ], and coefficients w[ ] are 32-bit. The output is returned in the separate array y[ ] in normal order. Each complex value is stored with interleaved real and imaginary parts. The code uses a special ordering of FFT coefficients (also called twiddle factors) and memory accesses to improve performance in the presence of cache. The C code to generate the twiddle factors is provided with this library in the 'c64plus\dsplib\src\DSP_fft32x32' directory.

Algorithm For further details, see the source code of the C and Optimized C version of this function that is provided in the 'c64plus\dsplib\src\DSP_fft32x32' directory.

Special Requirements

- In-place computation is *not* allowed.
- The size of the FFT, nx, must be a power of 2 or 4 and $16 \le nx \le 65536$.
- The arrays for the complex input data x[ ], complex output data y[ ], and twiddle factors w[ ] must be double-word aligned.
- The input and output data are complex, with the real/imaginary components stored in adjacent locations in the array. The real components are stored at even array indices, and the imaginary components are stored at odd array indices.
- The FFT coefficients (twiddle factors) are generated using the function gen_twiddle_fft32x32 provided in the 'c64plus\dsplib\src\DSP_fft32x32' directory. The scale factor must be 2147483647.5 No scaling is done with the function; thus the input data must be scaled by $2^{\log2(nx)}$ to completely prevent overflow.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.
- The routine uses $\log_4(nx) - 1$ stages of radix-4 transform and performs either a radix-2 or radix-4 transform on the last stage depending on nx. If nx is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform.

Benchmarks

Codesize 0x380 bytes

MIPS (CPU Cycles)			
N	**Nat C**	**INT C**	**SA**
16	417	150	149
32	1132	288	270
64	2146	492	422
128	5813	1132	966
256	11507	2208	1782
512	29430	5264	4278
1024	58740	10644	8374
2048	143479	24996	19926
4096	286837	50856	39638
8192	678008	116920	92406
16384	1355894	237756	184566

DSP_fft32x32s *Complex Forward Mixed Radix 32 x 32-bit FFT With Scaling*

Function void DSP_fft32x32s(int * restrict w, int nx, int * restrict x, int * restrict y)

Arguments

w[2*nx]	Pointer to complex 32-bit FFT coefficients.
nx	Length of FFT in complex samples. Must be power of 2 or 4, and 16 ≤ nx ≤ 65536.
x[2*nx]	Pointer to complex 32-bit data input.
y[2*nx]	Pointer to complex 32-bit data output.

Description This routine computes an extended precision complex forward mixed radix FFT with scaling, rounding and digit reversal. Input data x[ ], output data y[ ], and coefficients w[ ] are 32-bit. The output is returned in the separate array y[ ] in normal order. Each complex value is stored with interleaved real and imaginary parts. The code uses a special ordering of FFT coefficients (also called twiddle factors) and memory accesses to improve performance in the presence of cache. The C code to generate the twiddle factors is provided with this library in the 'c64plus\dsplib\src\DSP_fft32x32s' directory.

Algorithm For further details, see the source code of the C and Optimized C version of this function that is provided in the 'c64plus\dsplib\src\DSP_fft32x32s' directory.

Special Requirements

- ❏ In-place computation is *not* allowed.
- ❏ The size of the FFT, nx, must be a power of 2 or 4 and 16 ≤ nx ≤ 65536.
- ❏ The arrays for the complex input data x[ ], complex output data y[ ], and twiddle factors w[ ] must be double-word aligned.
- ❏ The input and output data are complex, with the real/imaginary components stored in adjacent locations in the array. The real components are stored at even array indices, and the imaginary components are stored at odd array indices.

- The FFT coefficients (twiddle factors) are generated using the function gen_twiddle_fft32x32s provided in the 'c64plus\dsplib\src\DSP_fft32x3s' directory. The scale factor must be 1073741823.5. No scaling is done with the function; thus the input data must be scaled by $2^{(\log2(nx) - ceil[\log4(nx) - 1])}$ to completely prevent overflow.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.
- Scaling is performed at each stage by shifting the results right by 1, preventing overflow.
- The routine uses $\log_4(nx) - 1$ stages of radix-4 transform and performs either a radix-2 or radix-4 transform on the last stage depending on nx. If nx is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform.

Benchmarks

Codesize 0x3C0 bytes

MIPS (CPU Cycles)			
N	**Nat C**	**INT C**	**SA**
16	404	150	133
32	1086	284	271
64	2052	506	423
128	5530	1220	971
256	10936	2410	1787
512	27902	5876	4287
1024	55676	11898	8383
2048	135810	28292	19939
4096	271488	57482	39651
8192	6411158	133268	92423
16384	1282180	270490	184583

DSP_ifft16x16 *Complex Inverse Mixed Radix 16 x 16-bit FFT With Rounding*

Function void DSP_ifft16x16(short * restrict w, int nx, short * restrict x, short * restrict y)

Arguments

w[2*nx]	Pointer to complex Q.15 FFT coefficients.
nx	Length of FFT in complex samples. Must be power of 2 or 4, and $16 \le nx \le 65536$.
x[2*nx]	Pointer to complex 16-bit data input.
y[2*nx]	Pointer to complex 16-bit data output.

Description

This routine computes a complex inverse mixed radix IFFT with rounding and digit reversal. Input data x[ ], output data y[ ], and coefficients w[ ] are 16-bit. The output is returned in the separate array y[ ] in normal order. Each complex value is stored with interleaved real and imaginary parts. The code uses a special ordering of IFFT coefficients (also called twiddle factors) and memory accesses to improve performance in the presence of cache.

The fft16x16 can be used to perform IFFT, by first conjugating the input, performing the FFT, and conjugating again. This allows fft16x16 to perform the IFFT as well. However, if the double conjugation needs to be avoided, then this routine uses the same twiddle factors as the FFT and performs an IFFT. The change in the sign of the twiddle factors is adjusted for in the routine. Hence, this routine uses the same twiddle factors as the fft16x16 routine.

Algorithm

For further details, see the source code of the C and Optimized C version of this function that is provided in the 'c64plus\dsplib\src\DSP_ifft16x16' directory.

Special Requirements

- ❏ In-place computation is *not* allowed.
- ❏ The size of the FFT, nx, must be a power of 2 or 4 and $16 \le nx \le 65536$.
- ❏ The arrays for the complex input data x[ ], complex output data y[ ], and twiddle factors w[ ] must be double-word aligned.
- ❏ The input and output data are complex, with the real/imaginary components stored in adjacent locations in the array. The real components are stored at even array indices, and the imaginary components are stored at odd array indices.
- ❏ Scaling by two is performed after each radix-4 stage except the last one.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.
- The routine uses $\log_4(nx) - 1$ stages of radix-4 transform and performs either a radix-2 or radix-4 transform on the last stage depending on nx. If nx is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform.

Benchmarks

Codesize 0x2E0

MIPS (CPU Cycles)			
N	**Nat C**	**INT C**	**SA**
16	302	193	99
32	654	403	157
64	1172	724	237
128	2878	1735	513
256	5612	3464	929
512	13718	8347	2165
1024	27284	17052	4213
2048	64926	40111	9865
4096	129692	82096	19593
8192	301222	188611	45213
16384	602276	385220	90269

DSP_ifft16x16_imre *Complex Inverse Mixed Radix 16 x 16-bit FFT With Im/Re Order*

Function void DSP_ifft16x16_imre(short * restrict w, int nx, short * restrict x, short * restrict y)

Arguments

w[2*nx]	Pointer to complex Q.15 FFT coefficients.
nx	Length of FFT in complex samples. Must be power of 2 or 4, and 16 ≤ nx ≤ 65536.
x[2*nx]	Pointer to complex data input.
y[2*nx]	Pointer to complex data output.

Description This routine computes a complex inverse mixed radix IFFT with rounding and digit reversal. Input data x[ ], output data y[ ], and coefficients w[ ] are 16-bit. The output is returned in the separate array y[ ] in normal order. Each complex value is stored with interleaved imaginary and real parts. The code uses a special ordering of IFFT coefficients (also called twiddle factors) and memory accesses to improve performance in the presence of cache.

The fft16x16_imre can be used to perform IFFT, by first conjugating the input, performing the FFT, and conjugating again. This allows fft16x16_imre to perform the IFFT as well. However, if the double conjugation needs to be avoided, then this routine uses the same twiddle factors as the FFT and performs an IFFT. The change in the sign of the twiddle factors is adjusted for in the routine. Hence, this routine uses the same twiddle factors as the fft16x16_imre routine.

Algorithm For further details, see the source code of the C and Optimized C version of this function that is provided in the 'c64plus\dsplib\src\DSP_ifft16x16_imre' directory.

Special Requirements

- ❏ In-place computation is *not* allowed.
- ❏ The size of the FFT, nx, must be a power of 2 or 4, and 16 ≤ nx ≤ 65536.
- ❏ The arrays for the complex input data x[ ], complex output data y[ ], and twiddle factors w[ ] must be double-word aligned.
- ❏ The input and output data are complex, with the **imaginary/real** components stored in adjacent locations in the array. The imaginary components are stored at even array indices, and the real components are stored at odd array indices.
- ❏ Scaling by two is performed after each radix-4 stage except the last one.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.
- The routine uses $\log_4(nx) - 1$ stages of radix-4 transform and performs either a radix-2 or radix-4 transform on the last stage depending on nx. If nx is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform.

Benchmarks

Codesize 0x400 bytes

MIPS (CPU Cycles)			
N	**Nat C**	**INT C**	**SA**
16	319	198	112
32	690	415	175
64	1252	734	255
128	3043	1751	536
256	5953	3478	952
512	14408	8367	2193
1024	28678	17070	4241
2048	67725	40135	9898
4096	135307	82118	19626
8192	312466	188639	45251
16384	624784	385246	90307

DSP_ifft16x32 *Complex Inverse Mixed Radix 16 x 32-bit FFT With Rounding*

Function void DSP_ifft16x32(short * restrict w, int nx, int * restrict x, int * restrict y)

Arguments

w[2*nx]	Pointer to complex Q.15 FFT coefficients.
nx	Length of FFT in complex samples. Must be power of 2 or 4, and 16 ≤ nx ≤ 65536.
x[2*nx]	Pointer to complex 32-bit data input.
y[2*nx]	Pointer to complex 32-bit data output.

Description

This routine computes an extended precision complex inverse mixed radix FFT with rounding and digit reversal. Input data x[ ] and output data y[ ] are 32-bit, coefficients w[ ] are 16-bit. The output is returned in the separate array y[ ] in normal order. Each complex value is stored with interleaved real and imaginary parts. The code uses a special ordering of FFT coefficients (also called twiddle factors) and memory accesses to improve performance in the presence of cache.

fft16x32 can be reused to perform IFFT, by first conjugating the input, performing the FFT, and conjugating again. This allows fft16x32 to perform the IFFT as well. However, if the double conjugation needs to be avoided, then this routine uses the same twiddle factors as the FFT and performs an IFFT. The change in the sign of the twiddle factors is adjusted for in the routine. Hence, this routine uses the same twiddle factors as the fft16x32 routine.

Algorithm

For further details, see the source code of the C and Optimized C version of this function that is provided in the 'c64plus\dsplib\src\DSP_ifft16x32' directory.

Special Requirements

- In-place computation is *not* allowed.
- The size of the FFT, nx, must be a power of 2 or 4 and 16 ≤ nx ≤ 65536.
- The arrays for the complex input data x[ ], complex output data y[ ], and twiddle factors w[ ] must be double-word aligned.
- The input and output data are complex, with the real/imaginary components stored in adjacent locations in the array. The real components are stored at even array indices, and the imaginary components are stored at odd array indices.

- The FFT coefficients (twiddle factors) are generated using the function gen_twiddle_ifft16x32 provided in the 'c64plus\dsplib\src\DSP_ifft16x32' directory. The scale factor must be 32767.5. No scaling is done with the function; thus the input data must be scaled by $2^{\log2(nx)}$ to completely prevent overflow.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.
- The routine uses $\log_4(nx) - 1$ stages of radix-4 transform and performs either a radix-2 or radix-4 transform on the last stage depending on nx. If nx is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform.

Benchmarks

Codesize 0x360 bytes

MIPS (CPU Cycles)			
N	**Nat C**	**INT C**	**SA**
16	326	190	133
32	725	297	247
64	1311	453	391
128	3264	986	901
256	6382	1794	1669
512	15703	4255	4003
1024	31253	8295	7843
2048	74654	19716	18625
4096	149148	39180	37057
8192	347301	91433	86239
16384	694435	182577	172255

DSP_ifft32x32 *Complex Inverse Mixed Radix 32 x 32-bit FFT With Rounding*

Function void DSP_ifft32x32(int * restrict w, int nx, int * restrict x, int * restrict y)

Arguments

w[2*nx]	Pointer to complex 32-bit FFT coefficients.
nx	Length of FFT in complex samples. Must be power of 2 or 4, and $16 \le nx \le 65536$.
x[2*nx]	Pointer to complex 32-bit data input.
y[2*nx]	Pointer to complex 32-bit data output.

Description

This routine computes an extended precision complex inverse mixed radix FFT with rounding and digit reversal. Input data x[ ], output data y[ ], and coefficients w[ ] are 32-bit. The output is returned in the separate array y[ ] in normal order. Each complex value is stored with interleaved real and imaginary parts. The code uses a special ordering of FFT coefficients (also called twiddle factors) and memory accesses to improve performance in the presence of cache.

fft32x32 can be reused to perform IFFT, by first conjugating the input, performing the FFT, and conjugating again. This allows fft32x32 to perform the IFFT as well. However, if the double conjugation needs to be avoided, then this routine uses the same twiddle factors as the FFT and performs an IFFT. The change in the sign of the twiddle factors is adjusted for in the routine. Hence, this routine uses the same twiddle factors as the fft32x32 routine.

Algorithm

For further details, see the source code of the C and Optimized C version of this function that is provided in the 'c64plus\dsplib\src\DSP_ifft32x32' directory.

Special Requirements

- In-place computation is *not* allowed.
- The size of the FFT, nx, must be a power of 2 or 4 and $16 \le nx \le 65536$.
- The arrays for the complex input data x[ ], complex output data y[ ], and twiddle factors w[ ] must be double-word aligned.
- The input and output data are complex, with the real/imaginary components stored in adjacent locations in the array. The real components are stored at even array indices, and the imaginary components are stored at odd array indices.

- The FFT coefficients (twiddle factors) are generated using the function gen_twiddle_ifft32x32 provided in the 'c64plus\dsplib\src\DSP_ifft32x32' directory. The scale factor must be 2147483647.5. No scaling is done with the function; thus the input data must be scaled by $2^{log2(nx)}$ to completely prevent overflow.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.
- The routine uses $log_4(nx) - 1$ stages of radix-4 transform and performs either a radix-2 or radix-4 transform on the last stage depending on nx. If nx is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform.

Benchmarks

Codesize 0x3A0 bytes

MIPS (CPU Cycles)			
N	**Nat C**	**INT C**	**SA**
16	380	148	145
32	968	297	264
64	1810	451	416
128	4812	1036	958
256	9498	1894	1774
512	24124	4559	4268
1024	48122	8905	8364
2048	117116	21234	19914
4096	234106	42220	39626
8192	552060	98581	92392
16384	1103994	196879	184552

4.4 Filtering and Convolution

DSP_fir_cplx *Complex FIR Filter*

Function

void DSP_fir_cplx (short * restrict x, short * restrict h, short * restrict r, int nh, int nr)

Arguments

x[2*(nr+nh−1)]	Complex input data. x must point to x[2*(nh−1)].
h[2*nh]	Complex coefficients (in normal order).
r[2*nr]	Complex output data.
nh	Number of complex coefficients. Must be a multiple of 2.
nr	Number of complex output samples. Must be a multiple of 4.

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 pointer to input array x must point to the (nh)th complex sample; i.e., element 2*(nh−1), upon entry to the function. The coefficients are expected in normal order.

Algorithm

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

```
void DSP_fir_cplx(short *x, short *h, short *r,short nh, short
nr)
{
    short i,j;
    int 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 >> 15);
```

```
            r[i+1] = (imag >> 15);
        }
    }
```

Special Requirements

- The number of coefficients nh must be a multiple of 2.
- The number of output samples nr must be a multiple of 4.

Implementation Notes

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

Benchmarks

Cycles	nr * (nh + 1.5) + 65
Codesize	960 bytes

DSP_fir_cplx_hM4X4 *Complex FIR Filter*

Function void DSP_fir_cplx _hM4X4(short * restrict x, short * restrict h, short * restrict r, int nh, int nr)

Arguments

x[2*(nr+nh−1)]	Complex input data. x must point to x[2*(nh−1)].
h[2*nh]	Complex coefficients (in normal order).
r[2*nr]	Complex output data.
nh	Number of complex coefficients. Must be a multiple of 4.
nr	Number of complex output samples. Must be a multiple of 4.

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 pointer to input array x must point to the (nh)th complex sample; i.e., element 2*(nh−1), upon entry to the function. The coefficients are expected in normal order.

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

```
void DSP_fir_cplx(short *x, short *h, short *r,short nh, short
nr)
{
    short i,j;
    int 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 >> 15);
        r[i+1] = (imag >> 15);
    }
}
```

Special Requirements

- The number of coefficients nh must be larger or equal to 4 and a multiple of 4.
- The number of output samples nr must be a multiple of 4.

Implementation Notes

- ***Interruptibility:*** The code is fully interruptible.

Benchmarks

Cycles	nr * (nh* 3) + 62
Codesize	960 bytes

DSP_fir_gen *FIR Filter*

Function

void DSP_fir_gen (short * restrict x, short * restrict h, short * 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 (coefficients must be in reverse order).
r[nr]	Pointer to output array of size nr. Must be word aligned.
nh	Number of coefficients. Must be ≥5.
nr	Number of samples to calculate. Must be a multiple of 4.

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[ ]. It operates on 16-bit data with a 32-bit accumulate. The filter calculates nr output samples using nh coefficients.

Algorithm

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

```
void DSP_fir_gen(short *x, short *h, short *r, int nh, int nr)
{
    int i, j, sum;

    for (j = 0; j < nr; j++) {
        sum = 0;
        for (i = 0; i < nh; i++)
            sum += x[i + j] * h[i];
        r[j] = sum >> 15;
    }
}
```

Special Requirements

- The number of outputs computed, nr, must be a multiple of 4 and greater than or equal to 4.
- Array r[ ] must be word aligned.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.
- Load double-word instruction is used to simultaneously load four values in a single clock cycle.

Benchmarks

Cycles: 1/2* nr * (1/2* nh + 9) + 26
Codesize: 416 bytes

DSP_fir_gen_hM17_rA8X8 *FIR Filter*

Function void DSP_fir_gen_hM17_rA8X8 (short * restrict x, short * restrict h, short * 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 (coefficients must be in reverse order).
r[nr]	Pointer to output array of size nr. Must be double word aligned.
nh	Number of coefficients.
nr	Number of samples to calculate. Must be a multiple of 8.

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[ ]. It operates on 16-bit data with a 32-bit accumulate. The filter calculates nr output samples using nh coefficients.

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

```
void DSP_fir_gen(short *x, short *h, short *r, int nh, int nr)
{
    int i, j, sum;

    for (j = 0; j < nr; j++) {
        sum = 0;
        for (i = 0; i < nh; i++)
            sum += x[i + j] * h[i];
        r[j] = sum >> 15;
    }
}
```

Special Requirements

- ❑ The number of outputs computed, nr, must be a multiple of 8 and greater than or equal to 8.
- ❑ Array r[ ] must be word aligned.

Implementation Notes

- ❑ ***Interruptibility:*** The code is fully interruptible.
- ❑ Load double-word instruction is used to simultaneously load four values in a single clock cycle.

Benchmarks

Cycles: nr * (1/4* nh + 5) + 45
Codesize: 800 bytes

DSP_fir_r4 *FIR Filter (when the number of coefficients is a multiple of 4)*

Function

void DSP_fir_r4 (short * restrict x, short * restrict h, short * 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 (coefficients must be in reverse order).
r[nr]	Pointer to output array of size nr.
nh	Number of coefficients. Must be multiple of 4 and ≥8.
nr	Number of samples to calculate. Must be multiple of 4.

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[ ]. This FIR operates on 16-bit data with a 32-bit accumulate. The filter calculates nr output samples using nh coefficients.

Algorithm

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

```
void DSP_fir_r4(short *x, short *h, short *r, int nh, int nr)
{
    int i, j, sum;

    for (j = 0; j < nr; j++) {
        sum = 0;
        for (i = 0; i < nh; i++)
            sum += x[i + j] * h[i];
        r[j] = sum >> 15;
    }
}
```

Special Requirements

- The number of coefficients, nh, must be a multiple of 4 and greater than or equal to 8. Coefficients must be in reverse order.
- The number of outputs computed, nr, must be a multiple of 4 and greater than or equal to 4.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.
- The load double-word instruction is used to simultaneously load four values in a single clock cycle.

Benchmarks

Cycles	nr * (1/4 * nh + 4) + 15
Codesize	320 bytes

DSP_fir_r8 *FIR Filter (when the number of coefficients is a multiple of 8)*

Function void DSP_fir_r8 (short * restrict x, short * h, short * 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 (coefficients must be in reverse order).
r[nr]	Pointer to output array of size nr. Must be word aligned.
nh	Number of coefficients. Must be multiple of 8, ≥ 8.
nr	Number of samples to calculate. Must be multiple of 4.

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[ ]. This FIR operates on 16-bit data with a 32-bit accumulate. The filter calculates nr output samples using nh coefficients.

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

```
void DSP_fir_r8 (short *x, short *h, short *r, int nh, int nr)
{
    int i, j, sum;

    for (j = 0; j < nr; j++) {
        sum = 0;
        for (i = 0; i < nh; i++)
            sum += x[i + j] * h[i];
        r[j] = sum >> 15;
        }
}
```

Special Requirements

- The number of coefficients, nh, must be a multiple of 8 and greater than or equal to 8. Coefficients must be in reverse order.
- The number of outputs computed, nr, must be a multiple of 4 and greater than or equal to 4.

- Array r[ ] must be word aligned.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.
- The load double-word instruction is used to simultaneously load four values in a single clock cycle.

Benchmarks

Cycles	nr * (1/4* nh + 5.25) + 16
Codesize	544 bytes

DSP_fir_r8_hM16_rM8A8X8 *FIR Filter (the number of coefficients is a multiple of 8)*

Function

void DSP_fir_r8_hM16_rM8A8X8 (short * restrict x, short * h, short * 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 (coefficients must be in reverse order).
r[nr]	Pointer to output array of size nr. Must be double word aligned.
nh	Number of coefficients. Must be multiple of 8, ≥ 16.
nr	Number of samples to calculate. Must be multiple of 8, ≥.8.

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[ ]. This FIR operates on 16-bit data with a 32-bit accumulate. The filter calculates nr output samples using nh coefficients.

Algorithm

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

```
void DSP_fir_r8 (short *x, short *h, short *r, int nh, int nr)
{
    int i, j, sum;

    for (j = 0; j < nr; j++) {
        sum = 0;
        for (i = 0; i < nh; i++)
            sum += x[i + j] * h[i];
        r[j] = sum >> 15;
        }
}
```

Special Requirements

- ❑ The number of coefficients, nh, must be a multiple of 8 and greater than or equal to 16. Coefficients must be in reverse order.
- ❑ The number of outputs computed, nr, must be a multiple of 8 and greater than or equal to 8.
- ❑ Array r[ ] must be double word aligned.

Implementation Notes

- ❑ ***Interruptibility:*** The code is interruptible.
- ❑ The load double-word instruction is used to simultaneously load four values in a single clock cycle.

Benchmarks

Cycles	1/2 * nr * (1/2* nh + 9) + 19
Codesize	544 bytes

DSP_fir_sym *Symmetric FIR Filter*

Function

void DSP_fir_sym (short * restrict x, short * restrict h, short * restrict r, int nh, int nr, int s)

Arguments

x[nr+2*nh]	Pointer to input array of size nr + 2*nh. Must be double-word aligned.
h[nh+1]	Pointer to coefficient array of size nh + 1. Coefficients are in normal order and only half (nh+1 out of 2*nh+1) are required. Must be double-word aligned.
r[nr]	Pointer to output array of size nr. Must be word aligned.
nh	Number of coefficients. Must be multiple of 8. The number of original symmetric coefficients is 2*nh+1.
nr	Number of samples to calculate. Must be multiple of 4.
s	Number of insignificant digits to truncate; e.g., 15 for Q.15 input data and coefficients.

Description

This function applies a symmetric filter to the input samples. The filter tap array h[] provides ‘nh+1’ total filter taps. The filter tap at h[nh] forms the center point of the filter. The taps at h[nh − 1] through h[0] form a symmetric filter about this central tap. The effective filter length is thus 2*nh+1 taps.

The filter is performed on 16-bit data with 16-bit coefficients, accumulating intermediate results to 40-bit precision. The accumulator is rounded and truncated according to the value provided in ‘s’. This allows a variety of Q-points to be used.

Algorithm

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

```
void DSP_fir_sym(short *x, short *h, short *r, int nh, int nr,
int s)
{
    int              i, j;
    long             y0;
    long             round = (long) 1 << (s – 1);
    for (j = 0; j < nr; j++) {
        y0 = round;
```

```
        for (i = 0; i < nh; i++)
            y0 += (short) (x[j + i] + x[j + 2 * nh – i]) * h[i];
        y0 += x[j + nh] * h[nh];
        r[j] = (int) (y0 >> s);
    }
}
```

Special Requirements

- ❑ nh must be a multiple of 8. The number of original symmetric coefficients is 2*nh+1. Only half (nh+1) are required.
- ❑ nr must be a multiple of 4.
- ❑ x[ ] and h[ ] must be double-word aligned.
- ❑ r[ ] must be word aligned.

Implementation Notes

- ❑ ***Interruptibility:*** The code is interruptible.
- ❑ The load double-word instruction is used to simultaneously load four values in a single clock cycle.

Benchmarks

Cycles	3/4 * nr + 33
Codesize	576 bytes

DSP_iir *IIR With 5 Coefficients*

Function short DSP_iir(short x, short * restrict h, int nh, short * restrict b)

Arguments

x	Input value (16−bit).
h[nh]	Coefficient input vector.
nh	Number of coefficients.
b[nh]	State vector.

Description This function implements an IIR filter, with a number of biquad stages given by nh / 4. It accepts a single sample of input and returns a single sample of output. Coefficients are expected to be in the range [−2.0, 2.0) with Q14 precision.

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

```
short DSP_iir (short Input, const short * Coefs, int
nCoefs, short * State)
{
    int x, p0, p1, i, j;
    x = (int) Input;
    for (i = j = 0; i < nCoefs; i += 4, j += 2) {
        p0 = Coefs[i + 2] * State[j] + Coefs[i + 3] * State[j +
        1];
        p1 = Coefs[i] * State[j] + Coefs[i + 1] * State[j + 1];

        State[j + 1] = State[j];

        State[j] = x + (p0 >> 14);
        x += (p0 + p1) >> 14;
    }

    return x;
}
```

Special Requirements

- nh must be a multiple of 8 and greater than or equal to 8.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.

Benchmarks

Cycles	4 * nr + 15
Codesize	192 bytes

DSP_iir_lat *All-Pole IIR Lattice Filter*

Function

void DSP_iir_lat(short * restrict x, int nx, short * restrict k, int nk, int * restrict b, short * restrict r)

Arguments

x[nx]	Input vector (16-bit).
nx	Length of input vector.
k[nk]	Reflection coefficients in Q.15 format.
nk	Number of reflection coefficients/lattice stages. Must be >=4. Make multiple of 2 to avoid bank conflicts.
b[nk+1]	Delay line elements from previous call. Should be initialized to all zeros prior to the first call.
r[nx]	Output vector (16-bit).

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

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

```
   void iirlat(short *x, int nx, short *k, int nk, int *b,
short *r)
{
    int rt;     /* output       */
    int i, j;


    for (j=0; j<nx; j++)
    {
        rt = x[j] << 15;
        for (i = nk – 1; i >= 0; i--)
```

```
        {
            rt       = rt   - (short)(b[i] >> 15) * k[i];
            b[i + 1] = b[i] + (short)(rt   >> 15) * k[i];
        }
        b[0] = rt;
        r[j] = rt >> 15;
    }
}
```

Special Requirements

- ❏ nk must be >= 4.
- ❏ No special alignment requirements

Implementation Notes

- ❏ ***Interruptibility:*** The code is interruptible.

Benchmarks

Cycles	nx * (1.25 * nk + 24) + 15
Codesize	288 bytes

4.5 Math

DSP_dotp_sqr *Vector Dot Product and Square*

Function int DSP_dotp_sqr(int G, short * restrict x, short * restrict y, int * restrict r, int nx)

Arguments

G	Calculated value of G (used in the VSELP coder).
x[nx]	First vector array
y[nx]	Second vector array
r	Result of vector dot product of x and y.
nx	Number of elements. Must be multiple of 4, and ≥12.
return int	New value of G.

Description This routine performs an nx element dot product of x[ ] and y[ ] and stores it in r. It also squares each element of y[ ] and accumulates it in G. G is passed back to the calling function in register A4. This computation of G is used in the VSELP coder.

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

```
int DSP_dotp_sqr (int G,short *x,short *y,int *r,
int nx)
{
    short *y2;
    short *endPtr2;
    y2 = x;
    for (endPtr2 = y2 + nx; y2 < endPtr2; y2++){
        *r += *y * *y2;
        G += *y * *y;
        y++;
    }
    return(G);
}
```

Special Requirements nx must be a multiple of 4 and greater than or equal to 12.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.

Benchmarks

Cycles	nx/2 + 19
Codesize	128 bytes

DSP_dotprod *Vector Dot Product*

Function int DSP_dotprod(short * restrict x, short * restrict y, int nx)

Arguments

x[nx]	First vector array. Must be double-word aligned.
y[nx]	Second vector array. Must be double word-aligned.
nx	Number of elements of vector. Must be multiple of 4.
return int	Dot product of x and y.

Description This routine takes two vectors and calculates their dot product. The inputs are 16-bit short data and the output is a 32-bit number.

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

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

Special Requirements

- ❑ The input length must be a multiple of 4.
- ❑ The input data x[ ] and y[ ] are stored on double-word aligned boundaries.

Implementation Notes

- ***Interruptibility:*** The code is fully interruptible.
- The code is unrolled 4 times to enable full memory and multiplier bandwidth to be utilized.

Benchmarks

Cycles	nx / 4 + 19
Codesize	96 bytes

DSP_maxval *Maximum Value of Vector*

Function short DSP_maxval (short *x, int nx)

Arguments

x[nx]	Pointer to input vector of size nx.
nx	Length of input data vector. Must be multiple of 8 and ≥32.
return short	Maximum value of a vector.

Description This routine finds the element with maximum value in the input vector and returns that value.

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

```
short DSP_maxval(short x[ ], int nx)
{
    int i, max;
    max = –32768;

    for (i = 0; i < nx; i++)
        if (x[i] > max)
            max = x[i];
    return max;
}
```

Special Requirements nx is a multiple of 8 and greater than or equal to 32.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.

Benchmarks

Cycles	nx / 8 + 13
Codesize	125 bytes

DSP_maxidx *Index of Maximum Element of Vector*

Function int DSP_maxidx (short *x, int nx)

Arguments

x[nx]	Pointer to input vector of size nx. Must be double-word aligned.
nx	Length of input data vector. Must be multiple of 16 and ≥ 48.
return int	Index for vector element with maximum value.

Description This routine finds the max value of a vector and returns the index of that value.

The input array is treated as 16 separate columns that are interleaved throughout the array. If values in different columns are equal to the maximum value, then the element in the leftmost column is returned. If two values within a column are equal to the maximum, then the one with the lower index is returned. Column takes precedence over index.

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

```
int DSP_maxidx(short x[ ], int nx)
{
    int max, index, i;
    max = –32768;
    for (i = 0; i < nx; i++)
        if (x[i] > max) {
            max = x[i];
            index = i;
        }
    return index;
}
```

Special Requirements

- ❏ nx must be a multiple of 16 and greater than or equal to 32.
- ❏ The input vector x[ ] must be double-word aligned.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.
- The code is unrolled 16 times to enable the full bandwidth of LDDW and MAX2 instructions to be utilized. This splits the search into 16 sub-ranges. The global maximum is then found from the list of maximums of the sub-ranges. Then, using this offset from the sub-ranges, the global maximum and the index of it are found using a simple match. For common maximums in multiple ranges, the index will be different to the above C code.

Benchmarks

Cycles	9 * nx / 64 + 70
Codesize	320 bytes

DSP_minval *Minimum Value of Vector*

Function short DSP_minval (short *x, int nx)

Arguments

x [nx]	Pointer to input vector of size nx.
nx	Length of input data. Must be a multiple of 4 and ≥ 8.
return short	Maximum value of a vector.

Description This routine finds the minimum value of a vector and returns the value.

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

```
short DSP_minval(short x[ ], int nx)
{
    int i, min;
    min = 32767;

    for (i = 0; i < nx; i++)
        if (x[i] < min)
            min = x[i];
    return min;
}
```

Special Requirements nx is a multiple of 4 and greater than or equal to 8.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.
- The input data is loaded using double word wide loads, and the MIN2 instruction is used to get to the minimum.

Benchmarks

Cycles	nx / 8 +16
Codesize	128 bytes

DSP_mul32 *32-Bit Vector Multiply*

Function void DSP_mul32(int * restrict x, int * restrict y, int * restrict r, short nx)

Arguments

x[nx]	Pointer to input data vector 1 of size nx. Must be double-word aligned.
y[nx]	Pointer to input data vector 2 of size nx. Must be double-word aligned.
r[nx]	Pointer to output data vector of size nx. Must be double-word aligned.
nx	Number of elements in input and output vectors. Must be multiple of 8 and ≥16.

Description The function performs a Q.31 x Q.31 multiply and returns the upper 32 bits of the result. The result of the intermediate multiplies are accumulated into a 40-bit long register pair, as there could be potential overflow. The contribution of the multiplication of the two lower 16-bit halves are not considered. The output is in Q.30 format. Results are accurate to least significant bit.

Algorithm In the comments below, X and Y are the input values. Xhigh and Xlow represent the upper and lower 16 bits of X. This is the natural C equivalent of the optimized intrinsic C code without restrictions. Note that the intrinsic C code is optimized and restrictions may apply.

```
void DSP_mul32(const int *x, const int *y, int *r,
short nx)
{
    short     i;
    int       a,b,c,d,e;
    for(i=nx;i>0;i--)
    {
        a=*(x++);
        b=*(y++);
        c=_mpyluhs(a,b); /* Xlow*Yhigh */
        d=_mpyhslu(a,b); /* Xhigh*Ylow */
        e=_mpyh(a,b);  /* Xhigh*Yhigh */
        d+=c;              /* Xhigh*Ylow+Xlow*Yhigh */
        d=d>>16;     /* (Xhigh*Ylow+Xlow*Yhigh)>>16 */
```

```
            e+=d;       /* Xhigh*Yhigh + */
                        /* (Xhigh*Ylow+Xlow*Yhigh)>>16 */
            *(r++)=e;
        }
    }
```

Special Requirements

- nx must be a multiple of 4 and greater than or equal to 4.
- Input and output vectors must be double-word aligned.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.

Benchmarks

Cycles	3 * nx/4 + 12
Codesize	128 bytes

DSP_neg32 *32-Bit Vector Negate*

Function void DSP_neg32(int * restrict x, int * restrict r, short nx)

Arguments

x[nx]	Pointer to input data vector 1 of size nx with 32-bit elements. Must be double-word aligned.
r[nx]	Pointer to output data vector of size nx with 32-bit elements. Must be double-word aligned.
nx	Number of elements of input and output vectors. Must be a multiple of 4 and ≥8.

Description This function negates the elements of a vector (32-bit elements). The input and output arrays must not be overlapped except for where the input and output pointers are exactly equal.

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

```
void DSP_neg32(int *x, int *r, short nx)
{
    short i;
    for(i=nx; i>0; i--)
        *(r++)=-*(x++);
}
```

Special Requirements

- ❑ nx must be a multiple of 4 and greater than or equal to 4.
- ❑ The arrays x[ ] and r[ ] must be double-word aligned.

Implementation Notes

- ❑ ***Interruptibility:*** The code is interruptible.

Benchmarks

Cycles	nx/2 + 11
Codesize	96 bytes

DSP_recip16 *16-Bit Reciprocal*

Function

void DSP_recip16(short * restrict x, short * restrict rfrac, short * restrict rexp, short nx)

Arguments

x[nx] Pointer to Q.15 input data vector of size nx.

rfrac[nx] Pointer to Q.15 output data vector for fractional values.

rexp[nx] Pointer to output data vector for exponent values.

nx Number of elements of input and output vectors.

Description

This routine returns the fractional and exponential portion of the reciprocal of an array x[ ] of Q.15 numbers. The fractional portion rfrac is returned in Q.15 format. Since the reciprocal is always greater than 1, it returns an exponent such that:

```
(rfrac[i] * 2rexp[i]) = true reciprocal
```

The output is accurate up to the least significant bit of rfrac, but note that this bit could carry over and change rexp. For a reciprocal of 0, the procedure will return a fractional part of 7FFFh and an exponent of 16.

Algorithm

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

```
void DSP_recip16(short *x, short *rfrac, short *rexp, short
nx)
{
    int i,j,a,b;
    short neg, normal;
    for(i=nx; i>0; i--)
    {
        a=*(x++);
        if(a<0)             /* take absolute value */
        {
            a=-a;
            neg=1;
        }
        else neg=0;
        normal=_norm(a);    /* normalize number */
```

```
        a=a<<normal;
        *(rexp++)=normal-15;    /* store exponent */
        b=0x80000000;        /* dividend = 1 */
        for(j=15;j>0;j--)
           b=_subc(b,a);     /* divide */
        b=b&0x7FFF;          /* clear remainder
                             /* (clear upper half) */
        if(neg) b=-b;     /* if originally
                             /* negative, negate */
        *(rfrac++)=b;     /* store fraction */
    }
}
```

Special Requirements None

Implementation Notes

- ***Interruptibility:*** The code is interruptible.
- The conditional subtract instruction, SUBC, is used for division. SUBC is used once for every bit of quotient needed (15).

Benchmarks

Cycles	9 * nx + 22
Codesize	224 bytes

DSP_vecsumsq *Sum of Squares*

Function int DSP_vecsumsq (short *x, int nx)

Arguments

x[nx]	Input vector
nx	Number of elements in x. Must be multiple of 4 and ≥8.
return int	Sum of the squares

Description This routine returns the sum of squares of the elements contained in the vector x[ ].

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

```
int DSP_vecsumsq(short x[ ], int nx)
{
    int i, sum=0;

    for(i=0; i<nx; i++)
    {
        sum += x[i]*x[i];
    }
    return(sum);
}
```

Special Requirements

- nx must be a multiple of 4 and greater than or equal to 4.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.
- The code is unrolled 4 times to enable full memory and multiplier bandwidth to be utilized.

Benchmarks

Cycles	nx/4 + 12
Codesize	64 bytes

DSP_w_vec *Weighted Vector Sum*

Function void DSP_w_vec(short * restrict x, short * restrict y, short m, short * restrict r, short nr)

Arguments

x[nr]	Vector being weighted. Must be double-word aligned.
y[nr]	Summation vector. Must be double-word aligned.
m	Weighting factor
r[nr]	Output vector
nr	Dimensions of the vectors. Must be multiple of 8 and ≥8.

Description This routine is used to obtain the weighted vector sum. Both the inputs and output are 16-bit numbers.

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

```
void DSP_w_vec(short x[ ],short y[ ],short m,
short r[ ],short nr)
{
    short i;

    for (i=0; i<nr; i++) {
        r[i] = ((m * x[i]) >> 15) + y[i];
    }
}
```

Special Requirements

- ❑ nr must be a multiple of 8 and ≥ 8.
- ❑ Vectors x[ ] and y[ ] must be double-word aligned.

Implementation Notes

- ❑ ***Interruptibility:*** The code is interruptible.
- ❑ Input is loaded in double-words.
- ❑ Use of packed data processing to sustain throughput.

Benchmarks

Cycles	3 * nx/8 + 20
Codesize	160 bytes

4.6 Matrix

DSP_mat_mul *Matrix Multiplication*

Function

void DSP_mat_mul(short * restrict x, int r1, int c1, short * restrict y, int c2, short * restrict r, int qs)

Arguments

x [r1*c1]	Pointer to input matrix of size r1*c1.
r1	Number of rows in matrix x.
c1	Number of columns in matrix x. Also number of rows in y.
y [c1*c2]	Pointer to input matrix of size c1*c2.
c2	Number of columns in matrix y.
r [r1*c2]	Pointer to output matrix of size r1*c2.
qs	Final right−shift to apply to the result.

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.

The values stored in the matrices are assumed to be fixed-point or integer values. All intermediate sums are retained to 32-bit precision, and no overflow checking is performed. The results are right-shifted by a user-specified amount, and then truncated to 16 bits.

Algorithm

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

```
void DSP_mat_mul(short *x, int r1, int c1, short *y, int c2,
short *r, int qs)
{
    int i, j, k;
    int 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 >> qs;
        }
}
```

Special Requirements

- The arrays x[], y[], and r[] are stored in distinct arrays. That is, in-place processing is not allowed.
- The input matrices have minimum dimensions of at least 1 row and 1 column, and maximum dimensions of 32767 rows and 32767 columns.

Implementation Notes

- ***Interruptibility:*** This code blocks interrupts during its innermost loop. Interrupts are not blocked otherwise. As a result, interrupts can be blocked for up to 0.25*c1' + 16 cycles at a time.
- The 'i' loop and 'k' loops are unrolled 2x. The 'j' loop is unrolled 4x. For dimensions that are not multiples of the various loops' unroll factors, this code calculates extra results beyond the edges of the matrix. These extra results are ultimately discarded. This allows the loops to be unrolled for efficient operation on large matrices while not losing flexibility.

Benchmarks

Cycles 0.25 * (r1' * c2' * c1') + 2.25 * (r1' * c2') + 11, where:
r1' = 2 * ceil(r1/2.0) (r1 rounded up to next even)
c1' = 2 * ceil(c1/2.0) (c1 rounded up to next even)
c2' = 4 * ceil(c2/4.0) (c2 rounded up to next mult of 4)
For r1= 1, c1= 1, c2= 1: 33 cycles
For r1= 8, c1=20, c2= 8: 475 cycles

Codesize 512 bytes

DSP_mat_trans *Matrix Transpose*

Function

void DSP_mat_trans(short * restrict x, short rows, short columns, short * restrict r)

Arguments

x[rows*columns]	Pointer to input matrix.
rows	Number of rows in the input matrix. Must be a multiple of 4.
columns	Number of columns in the input matrix. Must be a multiple of 4.
r[columns*rows]	Pointer to output data vector of size rows*columns.

Description

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

Algorithm

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

```
void DSP_mat_trans(short *x, short rows, short columns, short
*r)
{
    short i,j;
    for(i=0; i<columns; i++)
        for(j=0; j<rows; j++)
            *(r+i*rows+j)=*(x+i+columns*j);
}
```

Special Requirements

- Rows and columns must be a multiple of 4.
- Matrices are assumed to have 16-bit elements.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.
- Data from four adjacent rows, spaced “columns” apart are read, and a local 4x4 transpose is performed in the register file. This leads to four double words, that are “rows” apart. These loads and stores can cause bank conflicts; hence, non-aligned loads and stores are used.

Benchmarks

Cycles	(columns * rows / 16) * 9 + 17
Codesize	352 bytes

4.7 Miscellaneous

DSP_bexp *Block Exponent Implementation*

Function short DSP_bexp(const int *x, short nx)

Arguments

x[nx]	Pointer to input vector of size nx. Must be double-word aligned.
nx	Number of elements in input vector. Must be multiple of 8.
return short	Return value is the maximum exponent that may be used in scaling.

Description Computes the exponents (number of extra sign bits) of all values in the input vector x[ ] and returns the minimum exponent. This will be useful in determining the maximum shift value that may be used in scaling a block of data.

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

```
short DSP_bexp(const int *x, short nx)
{
    int       min_val =_norm(x[0]);
    short     n;
    int       i;
    for(i=1;i<nx;i++)
    {
        n =_norm(x[i]);  /* _norm(x) = number of */
                         /* redundant sign bits  */
        if(n<min_val) min_val=n;
    }
    return min_val;
}
```

Special Requirements

- ❏ nx must be a multiple of 8.
- ❏ The input vector x[ ] must be double-word aligned.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.

Benchmarks

Cycles	5/8 * nx + 8
Codesize	256 bytes

DSP_blk_eswap16 *Endian-Swap a Block of 16-Bit Values*

Function void blk_eswap16(void * restrict x, void * restrict r, int nx)

Arguments

x [nx]	Source data. Must be double-word aligned.
r [nx]	Destination array. Must be double-word aligned.
nx	Number of 16-bit values to swap. Must be multiple of 8.

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

```
void DSP_blk_eswap16(void *x, void *r, int  nx)
{
    int i;
    char *_x, *_r;

    if (r)
    {
        _x = (char *)x;
        _r = (char *)r;
    } else
    {
        _x = (char *)x;
        _r = (char *)r;
    }

    for (i = 0; i < nx; i++)
    {
        char t0, t1;
        t0 = _x[i*2 + 1];
        t1 = _x[i*2 + 0];
        _r[i*2 + 0] = t0;
        _r[i*2 + 1] = t1;
    }
}
```

Special Requirements

- Input and output arrays do not overlap, except when “r == NULL” so that the operation occurs in-place.
- The input array and output array are expected to be double-word aligned, and a multiple of 8 half-words must be processed.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.

Benchmarks

Cycles	nx / 4 + 8
Codesize	192 bytes

DSP_blk_eswap32 *Endian-Swap a Block of 32-Bit Values*

Function void blk_eswap32(void * restrict x, void * restrict r, int nx)

Arguments

x [nx] Source data. Must be double-word aligned.

r [nx] Destination array. Must be double-word aligned.

nx Number of 32-bit values to swap. Must be multiple of 4.

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

```
void DSP_blk_eswap32(void *x, void *r, int  nx)
{
    int i;
    char *_x, *_r;

    if (r)
    {
        _x = (char *)x;
        _r = (char *)r;
    } else
    {
        _x = (char *)x;
        _r = (char *)r;
    }

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

```
            t2 = _x[i*4 + 1];
            t3 = _x[i*4 + 0];
            _r[i*4 + 0] = t0;
            _r[i*4 + 1] = t1;
            _r[i*4 + 2] = t2;
            _r[i*4 + 3] = t3;
        }
    }
```

Special Requirements

- ❏ Input and output arrays do not overlap, except where “r == NULL” so that the operation occurs in-place.
- ❏ The input array and output array are expected to be double-word aligned, and a multiple of 4 words must be processed.

Implementation Notes

- ❏ ***Interruptibility:*** The code is interruptible.

Benchmarks

Cycles	nx / 2 + 11
Codesize	224 bytes

DSP_blk_eswap64 *Endian-Swap a Block of 64-Bit Values*

Function void blk_eswap64(void * restrict x, void * restrict r, int nx)

Arguments

x[nx]	Source data. Must be double-word aligned.
r[nx]	Destination array. Must be double-word aligned.
nx	Number of 64-bit values to swap. Must be multiple of 2.

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

```
void DSP_blk_eswap64(void *x, void *r, int  nx)
{
    int i;
    char *_x, *_r;

    if (r)
    {
        _x = (char *)x;
        _r = (char *)r;
    } else
    {
        _x = (char *)x;
        _r = (char *)r;
    }

    for (i = 0; i < nx; i++)
    {
        char t0, t1, t2, t3, t4, t5, t6, t7;
        t0 = _x[i*8 + 7];
        t1 = _x[i*8 + 6];
```

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

Special Requirements

- ❏ Input and output arrays do not overlap, except when “r == NULL” so that the operation occurs in-place.
- ❏ The input array and output array are expected to be double-word aligned, and a multiple of 2 double-words must be processed.

Implementation Notes

- ❏ ***Interruptibility:*** The code is interruptible.

Benchmarks

Cycles	nx + 11
Codesize	224 bytes

DSP_blk_move *Block Move (Overlapping)*

Function void DSP_blk_move(short * restrict x, short * restrict r, int nx)

Arguments

x [nx]	Block of data to be moved.
r [nx]	Destination of block of data.
nx	Number of elements in block. Must be multiple of 8 and ≥32.

Description This routine moves nx 16-bit elements from one memory location pointed to by x to another pointed to by r. The source and destination blocks can be overlapped.

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

```
void DSP_blk_move(short *x, short *r, int nx)
{
    int i;

    if( r < x )
    {
        for (I = 0; I < nx; i++)
            r[i] = x[i];
    } else
    {
        for (I = nx-1; I >= 0; i--)
            r[i] = x[i];
    }
}
```

Special Requirements

- ❑ nx must be a multiple of 8 and greater than or equal to 8.

Implementation Notes

- ❑ Twin input and output pointers are used.
- ❑ ***Interruptibility:*** The code is fully interruptible.

Benchmarks

Cycles	nx / 4 + 6
Codesize	64 bytes

DSP_fltoq15 *Float to Q15 Conversion*

Function void DSP_fltoq15(float * restrict x, short * restrict r, short nx)

Arguments

x[nx]	Pointer to floating-point input vector of size nx. x should contain the numbers normalized between [−1,1).
r[nx]	Pointer to output data vector of size nx containing the Q.15 equivalent of vector x.
nx	Length of input and output data vectors. Must be multiple of 2.

Description Convert the IEEE floating point numbers stored in vector x[ ] into Q.15 format numbers stored in vector r[ ]. Results are truncated toward zero. Values that exceed the size limit will be saturated to 0x7fff if value is positive and 0x8000 if value is negative. All values too small to be correctly represented will be truncated to 0.

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

```
void fltoq15(float x[], short r[], short nx)
{
    int i, a;

    for(i = 0; i < nx; i++)
    {
        a = 32768 * x[i];

        // saturate to 16-bit //
        if (a>32767)  a =  32767;
        if (a<-32768) a = -32768;

        r[i] = (short) a;
    }
}
```

Special Requirements nx must be a multiple of 2.

Implementation Notes

- Loop is unrolled twice.
- ***Interruptibility:*** The code is interruptible.

Benchmarks

Cycles	2 * nx + 10
Codesize	192 bytes

DSP_minerror *Minimum Energy Error Search*

Function

int minerror (short * restrict GSP0_TABLE, short * restrict errCoefs, int * restrict max_index)

Arguments

GSP0_TABLE[9*256]	GSP0 terms array. Must be double-word aligned.
errCoefs[9]	Array of error coefficients.
max_index	Pointer to GSP0_TABLE[max_index] found.
return int	Maximum dot product result.

Algorithm

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

```
int minerr
(
    const short *restrict GSP0_TABLE,
    const short *restrict errCoefs,
    int         *restrict max_index
)
{
    int 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 Array GSP0_TABLE[] must be double-word aligned.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.
- The load double-word instruction is used to simultaneously load four values in a single clock cycle.
- The inner loop is completely unrolled.
- The outer loop is 4 times unrolled.

Benchmarks

Cycles	2 * nx + 10
Codesize	1120 bytes

DSP_q15tofl *Q15 to Float Conversion*

Function void DSP_q15tofl(short * restrict x, float * restrict r, short nx)

Arguments

x[nx]	Pointer to Q.15 input vector of size nx.
r[nx]	Pointer to floating-point output data vector of size nx containing the floating-point equivalent of vector x.
nx	Length of input and output data vectors. Must be multiple of 2.

Description Converts the values stored in vector x[ ] in Q.15 format to IEEE floating point numbers in output vector r[ ].

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

```
void DSP_q15tofl(short *x, float *r, int nx)
{
    int i;
    for (i=0;i<nx;i++)
        r[i] = (float) x[i] / 0x8000;
}
```

Special Requirements nx must be a multiple of 2.

Implementation Notes

- ***Interruptibility:*** The code is interruptible.
- Loop is unrolled twice

Benchmarks

Cycles	9 * nx / 4 + 12
Codesize	704 bytes

Appendix A

Performance/Fractional Q Formats

This appendix describes performance considerations related to the C64x+ DSPLIB and provides information about the Q format used by DSPLIB functions.

Topic **Page**

A.1 Performance Considerations

The ceil( ) is used in some benchmark formulas to accurately describe the number of cycles. It returns a number rounded up, away from zero, to the nearest integer. For example, ceil(1.1) returns 2.

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 L1 memory. Any extra cycles due to placement of code or data in L2/external 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.

Many considerations need to be taken when designing an overall system. The DSPLIB benchmarks are tested in a standalone environment to test the functions. Better performance may be achieved by tuning the function(s) for an overall system

For more information on additional stall cycles due to memory hierarchy, see the *Signal Processing Examples Using TMS320C64x Digital Signal Processing Library* (SPRA884). The *TMS320C6000 DSP Cache User's Guide* (SPRU656A) presents how to optimize algorithms and function calls for better cache performance.

A.2 Fractional Q Formats

Unless specifically noted, DSPLIB functions use Q15 format, or to be more exact, Q0.15. 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 Q3.12 Format

Q.3.12 format places the sign bit after the fourth binary digit from the right, and the next 12 bits contain the two's complement fractional component. The approximate allowable range of numbers in Q.3.12 representation is (−8,8) and the finest fractional resolution is $2^{-12} = 2.441 \times 10^{-4}$.

Table A−1. Q3.12 Bit Fields

Bit	15	14	13	12	11	10	9	…	0
Value	S	I3	I2	I1	Q11	Q10	Q9	…	Q0

A.2.2 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−2. Q.15 Bit Fields

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

A.2.3 Q.31 Format

Q.31 format spans two 16-bit memory words. The 16-bit word stored in the lower memory location contains the 16 least significant bits, and the higher memory location contains the most significant 15 bits and the sign bit. The approximate allowable range of numbers in Q.31 representation is (−1,1) and the finest fractional resolution is $2^{-31} = 4.66 \times 10^{-10}$.

Table A−3. Q.31 Low Memory Location Bit Fields

Bit	15	14	13	12	…	3	2	1	0
Value	Q15	Q14	Q13	Q12	…	Q3	Q2	Q1	Q0

Table A−4. Q.31 High Memory Location Bit Fields

Bit	15	14	13	12	…	3	2	1	0
Value	S	Q30	Q29	Q28	…	Q19	Q18	Q17	Q16

Appendix B

Software Updates and Customer Support

This appendix provides information about software updates and customer support.

Topic **Page**

B.1 DSPLIB Software Updates

C64x+ 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 C64x+ DSPLIB, contact Texas Instruments at softwaresupport@ti.com.

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.

CCS: Code Composer Studio.

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).

DSP_autocor: Autocorrelation.

DSP_bexp: Block exponent implementation.

DSP_bitrev_cplx: Complex bit reverse.

DSP_blk_eswap16: Endian-swap a block of 16-bit values.

DSP_blk_eswap32: Endian-swap a block of 32-bit values.

DSP_blk_eswap64: Endian-swap a block of 64-bit values.

DSP_blk_move: Block move.

DSP_dotp_sqr: Vector dot product and square.

DSP_dotprod: Vector dot product.

DSP_fft: Complex forward FFT with digital reversal.

DSP_fft16x16r: Complex forward mixed radix 16- x 16-bit FFT with rounding.

DSP_fft16x16t: Complex forward mixed radix 16- x 16-bit FFT with truncation.

DSP_fft16x32: Complex forward mixed radix 16- x 32-bit FFT with rounding.

DSP_fft32x32: Complex forward mixed radix 32- x 32-bit FFT with rounding.

DSP_fft32x32s: Complex forward mixed radix 32- x 32-bit FFT with scaling.

DSP_fir_cplx: Complex FIR filter (radix 2).

DSP_fir_gen: FIR filter (general purpose).

DSP_firlms2: LMS FIR (radix 2).

DSP_fir_r4: FIR filter (radix 4).

DSP_fir_r8: FIR filter (radix 8).

DSP_fir_sym: Symmetric FIR filter (radix 8).

DSP_fltoq15: Float to Q15 conversion.

DSP_ifft16x32: Complex inverse mixed radix 16- x 32-bit FFT with rounding.

DSP_ifft32x32: Complex inverse mixed radix 32- x 32-bit FFT with rounding.

DSP_iir: IIR with 5 coefficients per biquad.

DSP_mat_mul: Matrix multiplication.

DSP_mat_trans: Matrix transpose.

DSP_maxidx: Index of the maximum element of a vector.

DSP_maxval: Maximum value of a vector.

DSP_minerror: Minimum energy error search.

DSP_minval: Minimum value of a vector.

DSP_mul32: 32-bit vector multiply.

DSP_neg32: 32-bit vector negate.

DSP_q15tofl: Q15 to float conversion.

DSP_radix2: Complex forward FFT (radix 2).

DSP_recip16: 16-bit reciprocal.

DSP_r4fft: Complex forward FFT (radix 4).

DSP_vecsumsq: Sum of squares.

DSP_w_vec: Weighted vector sum.

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 frame is used out 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, you 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 interface layer.

synchronous-burst static random-access memory (SBSRAM): RAM whose contents do 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 are 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

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

W

IMPORTANT NOTICE

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

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

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

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

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

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

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

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

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

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

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

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