TMS320C62x DSP Library Programmer's Reference

Literature Number: SPRU402A
April 2002

Printed on Recycled Paper

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 that third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of 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.

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.

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265

Preface

Read This First

About This Manual

Welcome to the TMS320C62x digital signal processor (DSP) Library, or DSPLIB for short. The DSPLIB is a collection of 33 high-level optimized DSP functions for the TMS320C62x device. This source code library includes C-callable functions (ANSI-C language compatible) for general signal processing math and vector functions.

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

- Overview – an introduction to the TI C62x DSPLIB
- Installation – information on how to install and rebuild DSPLIB
- DSPLIB Functions – a quick reference table listing of routines in the library
- DSPLIB Reference – a description of all DSPLIB functions complete with calling convention, algorithm details, special requirements and implementation notes
- Information about performance, fractional Q format and customer support

How to Use This Manual

The information in this document describes the contents of the TMS320C62x DSPLIB in several different ways.

- Chapter 1 provides a brief introduction to the TI C62x DSPLIB, shows the organization of the routines contained in the library, and lists the features and benefits of the DSPLIB.
- Chapter 2 provides information on how to install, use, and rebuild the TI C62x DSPLIB
- Chapter 3 provides a quick overview of all DSPLIB functions in table format for easy reference. The information shown for each function includes the syntax, a brief description, and a page reference for obtaining more detailed information.

- Chapter 4 provides a list of the routines within the DSPLIB organized into functional categories. The functions within each category are listed in alphabetical order and include arguments, descriptions, algorithms, benchmarks, and special requirements.
- Appendix A describes performance considerations related to the C62x DSPLIB and provides information about the Q format used by DSPLIB functions.
- Appendix B provides information about software updates and customer support.

Notational Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays are shown in a `special typeface`.
- In syntax descriptions, the function or macro appears in a **bold typeface** and the parameters appear in plainface within parentheses. Portions of a syntax that are in **bold** should be entered as shown; portions of a syntax that are within parentheses describe the type of information that should be entered.
- Macro names are written in uppercase text; function names are written in lowercase.
- The TMS320C62x is also referred to in this reference guide as the C62x.

Related Documentation From Texas Instruments

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

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

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

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

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

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

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

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

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

Trademarks

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

Contents

Tables

Chapter 1

Introduction

This chapter provides a brief introduction to the TI C62x 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 C62x DSPLIB

The TI C62x DSPLIB is an optimized DSP Function Library for C programmers using TMS320C62x devices. It includes C-callable, assembly-optimized general-purpose signal-processing routines. These routines are typically used in computationally intensive real-time applications where optimal execution speed is critical. By using these routines, you can achieve execution speeds considerably faster than equivalent code written in standard ANSI C language. In addition, by providing ready-to-use DSP functions, TI DSPLIB can significantly shorten your DSP application development time.

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

The routines contained in the library are organized into the following seven different functional categories:

- Adaptive filtering
  - DSP_firlms2
- Correlation
  - DSP_autocor
- FFT
  - DSP_bitrev_cplx
  - DSP_radix 2
  - DSP_r4fft
  - DSP_fft16x16r
- Filtering and convolution
  - DSP_fir_cplx
  - DSP_fir_gen
  - DSP_fir_r4
  - DSP_fir_r8
  - DSP_fir_sym
  - DSP_iir
  - DSP_iirlat

- 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_move
  - DSP_blk_eswap16
  - DSP_blk_eswap32
  - DSP_blk_eswap64
  - DSP_fltoq15
  - DSP_minerror
  - DSP_q15tofl

1.2 Features and Benefits

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

Chapter 2

Installing and Using DSPLIB

This chapter provides information on how to install and rebuild the TI C62x DSPLIB.

Topic **Page**

2.1 How to Install DSPLIB

Note:

You should read the README.TXT file for specific details of the release.

The archive has the following structure:

```
dsp62x.zip
      |
      +-- README.txt           Top-level README file
      |
      +-- lib
      |   |
      |   +-- dsp62x.lib       Library archive
      |   |
      |   +-- dsp62x.src       Full source archive
      |   |                    (Hand-assembly and headers)
      |   +-- dsp62x_sa.src    Full source archive
      |   |                    (Linear asm and headers)
      |   +-- dsp62x_c.src     Full source archive
      |                        (C and headers)
      |
      +-- include
      |   |
      |   +-- header files     Unpacked header files
      |
      +-- support              Support files
      |
      +-- doc
          |
          +-- dsp62xlib.pdf    This document
```

2.1.1 De-Archive DSPLIB

The *lib* directory contains the library archive and the source archive. Please install the contents of the lib directory in a directory pointed by your C_DIR environment. If you choose to install the contents in a different directory, make sure you update the C_DIR environment variable, for example, by adding the following line in autoexec.bat file:

```
SET C_DIR=<install_dir>/lib;<install_dir>/include;%C_DIR%
```

or under Unix/csh:

```
setenv C_DIR "<install_dir>/lib;<install_dir>/include;
$C_DIR"
```

or under Unix/Bourne Shell:

```
C_DIR="<install_dir>/lib;<install_dir>/include;$C_DIR";
export C_DIR
```

The *include* directory contains the header files necessary to be included in the C code when you call a DSPLIB function from C 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	–1.0	0.9999694824...
Q.31	32	fraction	–1.0	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 these cases.

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

2.2.2 Calling a DSPLIB Function From C

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

- ❑ Include the function header file corresponding to the DSPLIB function
- ❑ Link your code with dsp62x.lib
- ❑ Use a correct linker command file for the platform you use. Remember some functions in dsp62x.lib are written assuming little-endian mode of operation.

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

```
#include <DSP_autocor.h>
```

in your C file and compile and link using

```
cl6x main.c –z –o autocor_drv.out –lrts6200.lib –
ldsp62x.lib
```

Code Composer Studio Users

Assuming your C_DIR environment is correctly set up (as mentioned in section 2.1), you would have to add DSPLIB under Code Composer Studio environment by choosing dsp62x.lib from the menu *Project* → *Add Files to Project*. Also, you should make sure that you link with the run-time support library rts6200.lib.

2.2.3 Calling a DSP Function From Assembly

The C62x 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 C62x C compiler calling conventions. For more information, refer to section 8, *Runtime Environment*, of *TMS320C6000 Optimizing C Compiler User's Guide* (SPRU187).

2.2.4 How DSPLIB is Tested – Allowable Error

DSPLIB is tested under the Code Composer Studio environment against a reference C implementation. You can expect identical results between Reference C implementation and its Assembly implementation when using test routines that deal with fixed-point type results. The test routines that deal with floating points typically allow an error margin of 0.000001 when comparing the results of reference C code and DSPLIB assembly code.

2.2.5 How DSPLIB Deals With Overflow and Scaling Issues

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

2.2.6 Interrupt Behaviour of DSPLIB Functions

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

2.3 How to Rebuild DSPLIB

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

```
mk6x dsp62x.src –l dsp62x.lib
```

Chapter 3

DSPLIB Function Tables

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

Topic **Page**

3.1 Arguments and Conventions Used

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

Table 3–1. Argument Conventions

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

3.2 DSPLIB Functions

The routines included in the DSP library are organized into eight 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 (radix 2)	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_bitrev_cplx (int *x, short *index, int nx)	Complex Bit-Reverse	4-6
void DSP_radix2 (int nx, short x[ ], short w[ ])	Complex Forward FFT (radix 2)	4-9
void DSP_r4fft (int nx, short x[ ], short w[ ])	Complex Forward FFT (radix 4)	4-11
void DSP_fft16x16r(int nx, short *x, short *w, unsigned char *brev, short *y, int radix, int offset, int nmax)	Cache optimized mixed radix FFT with scaling and rounding, digit reversal, out of place. Input and output: 16 bits, Twiddle factor: 16 bits	4-14

Table 3–5. Filtering and Convolution

Functions	Description	Page
void DSP_fir_cplx (short *x, short *h, short *r, int nh, int nx)	Complex FIR Filter (radix 2)	4-23
void DSP_fir_gen (short *x, short *h, short *r, int nh, int nr)	FIR Filter (general purpose)	4-25
void DSP_fir_r4 (short *x, short *h, short *r, int nh, int nr)	FIR Filter (radix 4)	4-27
void DSP_fir_r8 (short *x, short *h, short *r, int nh, int nr)	FIR Filter (radix 8)	4-29
void DSP_fir_sym (short *x, short *h, short *r, int nh, int nr, int s)	Symmetric FIR Filter	4-31
void DSP_iir(short *r1, short *x, short *r2, short *h2, short *h1, int nr)	IIR with 5 Coefficients per Biquad	4-33
void iirlat(short *x, int nx, short *k, int nk, int *b, short *r)	All-pole IIR Lattice Filter	4-35

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	4-37
int DSP_dotprod(short *x, short *y, int nx)	Vector Dot Product	4-38
short DSP_maxval (short *x, int nx)	Maximum Value of a Vector	4-39
int DSP_maxidx (short *x, int nx)	Index of the Maximum Element of a Vector	4-40
short DSP_minval (short *x, int nx)	Minimum Value of a Vector	4-41
void DSP_mul32(int *x, int *y, int *r, short nx)	32-bit Vector Multiply	4-42
void DSP_neg32(int *x, int *r, int nx)	32-bit Vector Negate	4-44
void DSP_recip16 (short *x, short *rfrac, short *rexp, short nx)	16-bit Reciprocal	4-45
int DSP_vecsumsq (short *x, int nx)	Sum of Squares	4-47
void DSP_w_vec(short *x, short *y, short m, short *r, short nr)	Weighted Vector Sum	4-48

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	4-50
void DSP_mat_trans(short *x, short rows, short columns, short *r)	Matrix Transpose	4-52

Table 3–8. Miscellaneous

Functions	Description	Page
short DSP_bexp(int *x, unsigned nx)	Max Exponent of a Vector (for scaling)	4-53
void DSP_blk_move(short *x, short *r, int nx)	Move a Block of Memory	4-54
void blk_eswap16(void *x, void *r, int nx)	Endian-swap a block of 16-bit values	4-55
void blk_eswap32(void *x, void *r, int nx)	Endian-swap a block of 32-bit values	4-57
void blk_eswap64(void *x, void *r, int nx)	Endian-swap a block of 64-bit values	4-59
void DSP_fltoq15 (float *x,short *r, int nx)	Float to Q15 Conversion	4-61
int DSP_minerror (short *GSP0_TABLE,short *errCoefs, int max_index)	Minimum Energy Error Search	4-62
void DSP_q15tofl (short *x, float *r, int nx)	Q15 to Float Conversion	4-64

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 (radix 2)*

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

Arguments

h[nh]	Coefficient Array (Q.15)
x[nh]	Input Array (16-bit)
b	Error from previous FIR (Q.15)
nh	Number of coefficients. Must be multiple of 2.
return long	Output sample (Q.30)

Description This is an Least Mean Squared Adaptive FIR Filter. Given the error from the previous sample and pointer to the next sample it computes an update of the coefficients and then performs the FIR for the given input.

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

```
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 The number of coefficients nh must be a multiple of 2.

Implementation Notes

- The loop is unrolled once.
- ***Bank Conflicts:*** No bank conflicts occur.
- ***Endian:*** The code is ENDIAN NEUTRAL.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	3 * nh/2 + 26 For nh = 24: 62 cycles For nh = 16: 50 cycles
Codesize	256 bytes

4.2 Correlation

DSP_autocor *Autocorrelation*

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

Arguments

r[nr]	Resulting array of autocorrelation.
x[nr+nx]	Input array. Must be word aligned.
nx	Length of autocorrelation. Must be multiple of 8.
nr	Length of lags. Must be a multiple of 2.

Description This routine performs an autocorrelation of an input vector x. The length of the autocorrelation is nx samples. Since nr such autocorrelations are performed, input vector x needs to be of length nx + nr. This produces nr output results which are stored in an output array r.

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

```
void 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 2.
- ❏ x[] must be word aligned.

Implementation Notes

- ❏ The inner loop is unrolled eight times and the outer loop is unrolled twice.
- ❏ The outer loop is conditionally executed in parallel with the inner loop. This allows for a zero overhead outer loop.

- ***Bank Conflicts:*** nr/2 – 1 memory hits occur.
- ***Endian:*** The code is LITTLE ENDIAN.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	nr * nx /2 + 31 + (nr/2 – 1) For nx = 24, nr = 8: 130 cycles For nx = 40, nr = 10: 237 cycles
Codesize	544 bytes

4.3 FFT

DSP_bitrev_cplx *Complex Bit-Reverse*

Function void DSP_bitrev_cplx (int *x, short *index, int nx)

Arguments

x[nx]	Pointer to complex input vector x of size nx. Each element consists of a complex 16-bit data pair.
index[ ]	Array of size ≈ sqrt(nx) created by the routine bitrev_index to allow the fast implementation of the bit reversal.
nx	Number of elements in vector x. nx must be a power of 2.

Description This function bit-reverses the position of elements in the complex vector x. This function is used in conjunction with FFT routines to provide the correct format for the FFT input or output data. The bit-reversal of a bit-reversed order array yields a linear-order array.

The array index[] can be generated by the program bitrev_index provided in the directory ‘support\fft’. This index should be generated at compile time not by the DSP.

Algorithm TI retains all rights, title and interest in this code and only authorizes the use of this code on TI TMS320 DSPs manufactured by TI. This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_bitrev_cplx (int *x, short *index, int nx)
{
   int       i;
   short     i0, i1, i2, i3;
   short     j0, j1, j2, j3;
   int       xi0, xi1, xi2, xi3;
   int       xj0, xj1, xj2, xj3;
   short     t;
   int       a, b, ia, ib, ibs;
   int       mask;
   int       nbits, nbot, ntop, ndiff, n2, halfn;
   short     *xs = (short *) x;
   nbits = 0;
```

```
    i = nx;
    while (i > 1){
       i = i >> 1;
       nbits++;}
    nbot   = nbits >> 1;
    ndiff  = nbits & 1;
    ntop   = nbot + ndiff;
    n2     = 1 << ntop;
    mask   = n2 – 1;
    halfn  = nx >> 1;
    for(i0 = 0; i0 < halfn; i0 += 2) {
       b   = i0 & mask;
       a   = i0 >> nbot;
       if (!b) ia    = index[a];
       ib  = index[b];
       ibs = ib << nbot;

       j0  = ibs + ia;
       t   = i0 < j0;
       xi0 = x[i0];
       xj0 = x[j0];
       if (t){x[i0] = xj0;
          x[j0] = xi0;}
       i1  = i0 + 1;
       j1  = j0 + halfn;
       xi1 = x[i1];
       xj1 = x[j1];
       x[i1] = xj1;
       x[j1] = xi1;
       i3  = i1 + halfn;
       j3  = j1 + 1;
       xi3 = x[i3];
       xj3 = x[j3];
       if (t){x[i3] = xj3;
          x[j3] = xi3;}
    }
}
```

Special Requirements

- ❏ nx must be a power of 2.
- ❏ The index array must be set up by bitrev_index before the function DSP_bitrev_cplx is called.
- ❏ If nx ≤ 4K, one can use the char (8-bit) data type for the “index” variable. This would require changing the LDH when loading index values in the assembly routine to LDB. This would further reduce the size of the Index Table by half its size.

Implementation Notes

- ❏ ***Endian:*** The code is LITTLE ENDIAN.
- ❏ ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	(nx/4 + 2) * 7 + 18 For nx = 256: 480 cycles
Codesize	352 bytes

DSP_radix2 *Complex Forward FFT (radix 2)*

Function void DSP_radix2 (int nx, short x[], short w[])

Arguments

nx — Number of complex elements in vector x. Must be a power of 2 such that $16 \leq nx \leq 32768$.

x[2*nx] — Pointer to input and output sequences.

w[nx] — Pointer to vector of FFT coefficients.

Description This routine is used to compute FFT of a complex sequence of size nx, a power of 2, with “decimation-in-frequency decomposition” method. The output is in bit-reversed order. Each complex value consists of interleaved 16-bit real and imaginary parts. To prevent overflow, input samples may have to be scaled by 1/nx.

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

```
void DSP_radix2 (short x[],short nx,short w[])
{
    short n1,n2,ie,ia,i,j,k,l;
    short xt,yt,c,s;

    n2 = nx;
    ie = 1;
    for (k=nx; k > 1; k = (k >> 1) ) {
        n1 = n2;
        n2 = n2>>1;
        ia = 0;
        for (j=0; j < n2; j++) {
            c = w[2*ia];
            s = w[2*ia+1];
            ia = ia + ie;
            for (i=j; i < nx; i += n1) {
                l = i + n2;
                xt        = x[2*l] – x[2*i];
                x[2*i]    = x[2*i] + x[2*l];
                yt        = x[2*l+1] – x[2*i+1];
```

```
                x[2*i+1]  = x[2*i+1] + x[2*l+1];
                x[2*l]    = (c*xt + s*yt)>>15;
                x[2*l+1]  = (c*yt – s*xt)>>15;
            }
        }
        ie = ie<<1;
    }
}
```

Special Requirements

- ❏ $16 \le nx \le 32768$ (nx is a power of 2)
- ❏ Input x and coefficients w should be in different data sections or memory spaces to eliminate memory bank hits. If this is not possible, they should be aligned on different word boundaries to minimize memory bank hits.
- ❏ x data is stored in the order real[0], image[0], real[1], ...
- ❏ The coefficient array w[] must be setup as follows:

  w[2*i] = –k * cos(i * delta), w[2*i+1] = –k * sin(i * delta)

  where delta = 2*π/nx, k = 32767 and i = 0..nx/2
- ❏ The program tw_radix2 provided in the directory ‘support’ may be used to generate the coefficient array.

Implementation Notes

- ❏ Loads input x and coefficient w as words.
- ❏ Both loops j and i0 shown in the C code are placed in the inner loop of the assembly code.
- ❏ ***Bank Conflicts:*** *S*ee Benchmarks.
- ❏ ***Endian:*** The code is LITTLE ENDIAN.
- ❏ ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	log2(nx) * (4 * nx/2 + 7) + 34 + nx/4 (The term nx/4 is due to bank conflicts) For nx = 256: 4250 cycles
Codesize	800 bytes

DSP_r4fft *Complex Forward FFT (radix 4)*

Function void DSP_r4fft (int nx, short x[ ], short w[ ])

Arguments

nx — Number of complex elements in vector x. Must be a power of 4 such that $4 \le nx \le 65536$

x[2*nx] — Pointer to input and output sequences. Must be aligned at a 4*nx byte boundary.

w[3*nx/4] — Pointer to vector of FFT coefficients.

Description This routine is used to compute FFT of a complex sequence size nx, a power of 4, with “decimation-in-frequency decomposition” method. The output is in digit-reversed order. Each complex value is with interleaved 16-bit real and imaginary parts.

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

```
void DSP_r4fft (int nx, short x[ ], short w[ ])
{
   int    n1, n2, ie, ia1, ia2, ia3, i0, i1, i2, i3,
          j, k;
   short  t, r1, r2, s1, s2, co1, co2, co3, si1,
          si2, si3;

   n2 = nx;
   ie = 1;
   for (k = nx; k > 1; k >>= 2) {
      n1 = n2;
      n2 >>= 2;
      ia1 = 0;
      for (j = 0; j < n2; j++) {
         ia2 = ia1 + ia1;
         ia3 = ia2 + ia1;
         co1 = w[ia1 * 2 + 1];
         si1 = w[ia1 * 2];
         co2 = w[ia2 * 2 + 1];
         si2 = w[ia2 * 2];
```

```
                co3 = w[ia3 * 2 + 1];
                si3 = w[ia3 * 2];
                ia1 = ia1 + ie;
                for (i0 = j; i0 < nx; i0 += n1) {
                    i1 = i0 + n2;
                    i2 = i1 + n2;
                    i3 = i2 + n2;
                    r1 = x[2 * i0] + x[2 * i2];
                    r2 = x[2 * i0] – x[2 * i2];
                    t = x[2 * i1] + x[2 * i3];
                    x[2 * i0] = r1 + t;
                    r1 = r1 – t;
                    s1 = x[2 * i0 + 1] + x[2 * i2 + 1];
                    s2 = x[2 * i0 + 1] – x[2 * i2 + 1];
                    t = x[2 * i1 + 1] + x[2 * i3 + 1];
                    x[2 * i0 + 1] = s1 + t;
                    s1 = s1 – t;
                    x[2 * i2] = (r1 * co2 + s1 * si2) >>
                    15;
                    x[2 * i2 + 1] = (s1 * co2–r1 *
                    si2)>>15;
                    t = x[2 * i1 + 1] – x[2 * i3 + 1];
                    r1 = r2 + t;
                    r2 = r2 – t;
                    t = x[2 * i1] – x[2 * i3];
                    s1 = s2 – t;
                    s2 = s2 + t;
                    x[2 * i1] = (r1 * co1 + s1 * si1)
                    >>15;
                    x[2 * i1 + 1] = (s1 * co1–r1 *
                    si1)>>15;
                    x[2 * i3] = (r2 * co3 + s2 * si3)
                    >>15;
                    x[2 * i3 + 1] = (s2 * co3–r2 *
                    si3)>>15;
                }
            }
            ie <<= 2;
        }
    }
```

Special Requirements

- ❑ $4 \leq nx \leq 65536$ (nx a power of 4)
- ❑ x[] is aligned on a 4*nx Byte (nx*word) boundary for circular buffering
- ❑ Input x[] and coefficients w[] should be in different data sections or memory spaces to eliminate memory bank hits. If this is not possible, they should be aligned on an odd word boundaries to minimize memory bank hits
- ❑ x[] data is stored in the order real[0], image[0], real[1], ...
- ❑ The coefficient array w[] must be setup as follows:

  w[2*i] = k * sin(i * delta), w[2*i+1] = k * cos(i * delta)

  where delta = $2*\pi/nx$, k = 32767 and i = 0..3*nx/4
- ❑ The program tw_r4fft provided in the directory ‘support’ may be used to generate the coefficient array.

Implementation Notes

- ❑ Loads input x and coefficient w as words.
- ❑ Both loops j and i0 shown in the C code are placed in the INNERLOOP of the assembly code.
- ❑ ***Bank Conflicts:*** *S*ee Benchmarks.
- ❑ ***Endian:*** The code is LITTLE ENDIAN.
- ❑ ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	$\log_4(nx) * (10 * nx/4 + 29) + 36 + nx/4$ (The term nx/4 is due to bank conflicts) For nx = 256: 2776 cycles
Codesize	736 bytes

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

Function

void DSP_fft16x16r(int nx, short *x, short *w, unsigned char *brev, short *y, int radix, int offset, int nmax)

Arguments

nx	Length of FFT in complex samples. Must be power of 2 and ≤16384
x[2*nx]	Pointer to complex 16-bit data input
w[2*nx]	Pointer to complex FFT coefficients (see Description)
brev[64]	Pointer to bit reverse table containing 64 entries (See Implementation Notes.)
y[2*nx]	Pointer to complex 16-bit data output (normal order).
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 computes a complex forward split-radix FFT. To avoid overflow this routine applies scaling by ½ after each stage. 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 routine can be called in a multi-pass fashion as to reduce cycle overhead due to cache misses. The code uses a special ordering of FFT coefficients (twiddle factors). The program tw_fft16x16 provided in the directory ‘support’ may be used to generate the coefficient array.

This redundant set of twiddle factors is size 2*N short samples. As pointed out later 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 below:

```
void dft(int n, short x[], short y[])
{
   int k,i, index;
   const double PI = 3  4159654;
   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 takes place at each stage except the last one. This is a divide by 2 to prevent overflow. All shifts are rounded to reduce truncation noise power by 3dB. The function takes the table and input data and calculates the FFT producing the frequency domain data in the y[ ] array. As the FFT allows every input point to effect every output point, in a cache based system this causes cache thrashing. This is mitigated by allowing the main FFT of size N to be divided into several steps, allowing as much data reuse as possible. For example the following function:

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

is equivalent to:

```
DSP_fft16x16r(1024,&x[2*0],  &w[0]    ,y,brev,256,  0,1024);
DSP_fft16x16r(256, &x[2*0],  &w[2*768],y,brev,4,    0,1024);
DSP_fft16x16r(256, &x[2*256],&w[2*768],y,brev,4,  256,1024);
DSP_fft16x16r(256, &x[2*512],&w[2*768],y,brev,4,  512,1024);
DSP_fft16x16r(256, &x[2*768],&w[2*768],y,brev,4,  768,1024);
```

Notice how the first FFT function is called on the entire 1K data set it covers the first pass of the FFT until the butterfly size is 256.

The following 4 FFTs do 256-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 to decompose the FFT the following would be needed :

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

is equivalent to:

```
DSP_fft16x16r(512, &x[0],    &w[0],    y,brev,128,  0,512);
DSP_fft16x16r(128, &x[2*0],  &w[2*384],y,brev,2,    0,512);
DSP_fft16x16r(128, &x[2*128],&w[2*384],y,brev,2,  128,512);
DSP_fft16x16r(128, &x[2*256],&w[2*384],y,brev,2,  256,512);
DSP_fft16x16r(128, &x[2*384],&w[2*384],y,brev,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],      brev,y,N/4,0,    N)
DSP_fft16x16r(N/4,&x[0],      &w[2*3*N/4],brev,y,rad,0,    N)
DSP_fft16x16r(N/4,&x[2*N/4],  &w[2*3*N/4],brev,y,rad,N/4,  N)
DSP_fft16x16r(N/4,&x[2*N/2],  &w[2*3*N/4],brev,y,rad,N/2,  N)
DSP_fft16x16r(N/4,&x[2*3*N/4],&w[2*3*N/4],brev,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” is used to control how many stages of decomposition are performed. It is also used to determine whether a radix-4 or radix-2 decomposition should be performed at the last stage. Hence when “rad” is set to “N/4” the first stage of the transform alone is performed and the code exits. To complete the FFT, four other calls are required to perform N/4 size FFTs. In fact, the ordering of these 4 FFTs amongst themselves does not matter and hence from a cache perspective, it helps to go through the remaining 4 FFTs in exactly the opposite order to the first. This is illustrated as follows:

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

Algorithm

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

```
void DSP_fft16x16r(int nx, short x[], short w[], unsigned char
brev[], short y[], int radix, int offset, int nmax)
{
     int    n1, n2, ie, ia1, ia2, ia3, i0, i1, i2, i3, i, l0;
     short  co1, co2, co3, si1, si2, si3;
     short  xt0, yt0, xt1, yt1, xt2, yt2;
     short  xh0,xh1,xh20,xh21,xl0,xl1,xl20,xl21;
     short * ptr_x0, * y0;
     unsigned int  j0, j1, k0, k1, k, j;
     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;

     n2 = n;
     ie = 1;
     for (k = n; k > radix; k >>= 2)
     {
         n1 = n2;
         n2 >>= 2;
         ia1 = 0;
         for (j = 0; j < n2; j++)
         {
             ia2 = ia1 + ia1;
             ia3 = ia2 + ia1;
             co1 = w[2 * ia1    ];
```

```
                    si1 = w[2 * ia1 + 1];
                    co2 = w[2 * ia2    ];
                    si2 = w[2 * ia2 + 1];
                    co3 = w[2 * ia3    ];
                    si3 = w[2 * ia3 + 1];
                    ia1 = ia1 + ie;
                    for (i0 = j; i0 < n; i0 += n1)
                    {
                        i1 = i0 + n2;
                        i2 = i1 + n2;
                        i3 = i2 + n2;

                        xh0  = x[2 * i0    ] + x[2 * i2    ];
                        xh1  = x[2 * i0 + 1] + x[2 * i2 + 1];
                        xl0  = x[2 * i0    ] – x[2 * i2    ];
                        xl1  = x[2 * i0 + 1] – x[2 * i2 + 1];

                        xh20 = x[2 * i1    ] + x[2 * i3    ];
                        xh21 = x[2 * i1 + 1] + x[2 * i3 + 1];
                        xl20 = x[2 * i1    ] – x[2 * i3    ];
                        xl21 = x[2 * i1 + 1] – x[2 * i3 + 1];

                        x[2 * i0    ] = (xh0 + xh20 + 1)>>1;
                        x[2 * i0 + 1] = (xh1 + xh21 + 1)>>1;

                        xt0  = xh0 – xh20;
                        yt0  = xh1 – xh21;
                        xt1  = xl0 + xl21;
                        yt2  = xl1 + xl20;
                        xt2  = xl0 – xl21;
                        yt1  = xl1 – xl20;

                        x[2 * i2    ]= (xt1 * co1 + yt1 * si1 +
0x00008000)>> 16;
                        x[2 * i2 + 1]= (yt1 * co1 – xt1 * si1 +
0x00008000)>> 16;
```

```
                x[2 * i1    ]= (xt0 * co2 + yt0 * si2 +
0x00008000)>> 16;
                x[2 * i1 + 1]= (yt0 * co2 – xt0 * si2 +
0x00008000)>> 16;
                x[2 * i3    ]= (xt2 * co3 + yt2 * si3 +
0x00008000)>> 16;
                x[2 * i3 + 1]= (yt2 * co3 – xt2 * si3 +
0x00008000)>> 16;
             }
         }

         ie <<= 2;
    }

    j = 0;
    ptr_x0 = x;
    y0 = y;
    l0 = _norm(n) – 17;

    if(radix == 2 || radix  == 4) for (i = 0; i < n; i += 4)
    {


            j0 = (j     ) & 0x3F;
            j1 = (j >> 6) & 0x3F;
            k0 = brev[j0];
            k1 = brev[j1];
            k = (k0 << 6) |  k1;
            if (l0 < 0) k = k << –l0;
            else        k = k >> l0;
            j++;

            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[ ], and twiddle factors w[ ] must be double-word aligned.
- Real values are stored in even word, imaginary in odd.
- Output array is double word aligned.
- All data is in short precision or Q.15 format
- Output results are returned in normal order.
- FFT coefficients (twiddle factors) are generated using the program tw_fft16x16 provided in the directory ‘support’.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Endian:*** The code is LITTLE ENDIAN.
- ***Interruptibility:*** The code is interrupt-tolerant but not 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 has the effect of making the data come out in bit reversed rather than in radix 4 digit reversed order. This simplifies the last pass of the loop. A simple table is used to do the bit reversal out of place.

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

- For more aggressive overflow control the shift in the DC term can be adjusted to 2 and the twiddle factors shifted right by 1. This gives a divide by 4 at each stage. For better accuracy the data can be pre-asserted left by so many bits so that as it builds in magnitude. The divide by 2 prevents too much growth. An optimal point for example with an 8192-point FFT with input data precision of 8 bits is to assert the input 4 bits left to make it 12 bits. This gives an SNR of 68dB at the output. By trying combinations the optimal can be found. If scaling is not required it is possible to replace the MPY by SMPY this will give a shift left by 1 so a shift right by 16 gives a total 15 bit shift right. The DC term must be adjusted to give a zero shift.

Benchmarks

Cycles 2.5 * N * ceil[$\log_4(nx)$] – nx/2 + 164

Code size 1344 bytes

4.4 Filtering and Convolution

DSP_fir_cplx *Complex FIR Filter*

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

Arguments

x[2*(nr+nh–1)]	Pointer to complex input array. Must point to element x[2*(nh–1)].
h[2*nh]	Pointer to complex coefficient array. Coefficients must be in normal order.
r[2*nr]	Pointer to complex output array.
nh	Number of complex coefficients in vector h. Must be a multiple of 2.
nr	Number of complex output samples to calculate. nh * nr must be >=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 coefficients are expected in normal order.

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

```
void 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.
- nr * nh must be >= 4.
- The input data pointer x must point to the (nh)th complex element, i.e. element 2*(nh–1).

Implementation Notes

- The inner loop is unrolled twice.
- The outer loop is conditionally executed in parallel with the inner loop.
- Both the inner and outer loops are software pipelined.
- ***Bank Conflicts:*** No bank conflicts occur.
- ***Endian:*** The code is LITTLE ENDIAN.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles 2* nh * nr + 20

Code size 384 bytes

DSP_fir_gen *FIR Filter (general purpose)*

Function void DSP_fir_gen (short *x, short *h, short *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 ≥5.
nr	Number of output samples to calculate.

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 C equivalent of the assembly code without restrictions. Note that the assembly code is hand 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 nh, the number of coefficients, must be greater than or equal to 5.

Implementation Notes

- The inner loop is unrolled four times, but the last three accumulates are executed conditionally to allow for a number of coefficients that is not a multiple of four.
- The outer loop is unrolled twice, but the last store is executed conditionally to allow for a number of output samples that is not a multiple of two.

- Both the inner and outer loops are software pipelined.
- ***Bank Conflicts:*** No bank conflicts occur.
- ***Endian:*** The code is ENDIAN NEUTRAL.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles [9 + 4 * ceil(nh/4)] * ceil(nr/2) + 18
For nh = 13, nr = 19: 268 cycles

Code size 640 bytes

DSP_fir_r4 *FIR Filter (radix 4)*

Function void DSP_fir_r4 (short *x, short *h, short *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. 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 2.

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 C equivalent of the assembly code without restrictions. Note that the assembly code is hand 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

- ❑ nh, the number of coefficients, must be a multiple of 4 and greater than or equal to 8.
- ❑ nr, the number of outputs computed, must be a multiple of 2.

Implementation Notes

- The routine performs 2 output samples at a time. The inner loop is unrolled four times. The outer loop is unrolled twice. Both the inner and outer loops are software pipelined.
- ***Bank Conflicts:*** No bank conflicts occur.
- ***Endian:*** The code is ENDIAN NEUTRAL.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	(8 + nh) * nr/2 + 14 For nh=20 and nr=42: 602 cycles
Code size	544 bytes

DSP_fir_r8 *FIR Filter (radix 8)*

Function void DSP_fir_r8 (short *x, short *h, short *r, int nh, int nr)

Arguments

x[nr+nh–1]	Pointer to input array of size nr + nh – 1. Must be word aligned.
h[nh]	Pointer to coefficient array of size nh. Coefficients must be in reverse order. Must be word-aligned.
r[nr]	Pointer to output array of size nr.
nh	Number of coefficients. Must be multiple of 8 and ≥ 8.
nr	Number of samples to calculate. Must be multiple of 2.

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 C equivalent of the assembly code without restrictions. Note that the assembly code is hand 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

- ❏ nh, the number of coefficients, must be a multiple of 8 and greater than or equal to 8.
- ❏ nr, the number of outputs computed, must be a multiple of 2.
- ❏ Arrays x[] and h[] must be word-aligned

Implementation Notes

- The assembly routine performs 2 output samples at a time. The inner loop is unrolled eight times. The outer loop is unrolled twice. Both the inner and outer loops are software pipelined.
- ***Bank Conflicts:*** No bank conflicts occur.
- ***Endian:*** The code is LITTLE ENDIAN.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	nh * nr/2 + 28 For nh=24 and nr=42: 532 cycles
Code size	544 bytes

DSP_fir_sym *Symmetric FIR Filter*

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

Arguments

x[nr+2*nh]	Pointer to input array of size nr + 2*nh.
h[nh+1]	Pointer to coefficient array of size nh + 1. Must be word aligned.
r[nr]	Pointer to output array of size nr.
nh	Number of coefficients. Must be multiple of 8.
nr	Number of samples to calculate. Must be multiple of 2.
s	Number of insignificant digits to truncate.

Description This symmetric FIR filter assumes the number of filter coefficients is 2 * nh + 1. It operates on 16-bit data with a 40-bit accumulation. The filter calculates nr output samples.

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

```
void 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.
- ❏ nr must be a multiple of 2.
- ❏ h[ ] must be word aligned.

Implementation Notes

- The load word instruction is used to simultaneously load two values form h[] in a single clock cycle.
- The inner and outer loop is unrolled eight times.
- ***Bank Conflicts:*** No bank conflicts occur.
- ***Endian:*** The code is LITTLE ENDIAN.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	(3 * nh/2 + 10) * nr/2 + 20 For nh=24, nr=42: 986 cycles
Code size	416 bytes

DSP_iir *IIR With 5 Coefficients per Biquad*

Function void DSP_iir (short *r1, short *x, short *r2, short *h2, short *h1, int nr)

Arguments

r1[nr+4]	Output array (used)
x[nr+4]	Input array
r2[nr]	Output array (stored)
h2[5]	Auto-regressive filter coefficients
h1[5]	Moving-average filter coefficients
nr	Number of output samples.

Description The IIR performs an auto-regressive moving-average (ARMA) filter with 4 auto-regressive filter coefficients and 5 moving-average filter coefficients for nr output samples. The output vector is stored in two locations. This routine is used as a high pass filter in the VSELP vocoder. All data is assumed to be 16-bit.

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

```
void DSP_iir(short *r1, short *x, short *r2, short *h2,
short *h1, int nr)
{
   int j,i;
   int sum;
   for (i=0; i<nr; i++){
      sum = h2[0] * x[4+i];
      for (j = 1; j <= 4; j++)
         sum += h2[j]*x[4+i–j]–h1[j]*r1[4+i–j];
      r1[4+i] = (sum >> 15);
      r2[i] = r1[4+i];
   }
}
```

Special Requirements To avoid memory bank conflicts, r1[] and r2[] must be aligned on the next word boundary following the alignment of x[].

Implementation Notes

- Output array r1[] contains nr + 4 locations, r2[] contains nr locations for storing nr output samples. The output samples are stored with an offset of 4 into the r1[ ] array.
- The inner loop is completely unrolled and software pipelined.
- ***Bank Conflicts:*** See Special Requirements.
- ***Endian:*** The code is ENDIAN NEUTRAL.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles 5 * nr + 30
For nr = 40: 230 cycles

Code size 384 bytes

DSP_iirlat *All-pole IIR Lattice Filter*

Function void iirlat(short *x, int nx, short *k, int nk, int *b, short *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 C equivalent of the assembly code without restrictions. Note that the assembly code is hand 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
- ❏ see *Bank Conflicts* for avoiding bank conflicts

Implementation Notes

- ❏ Prolog and epilog of the inner loop are partially collapsed and overlapped to reduce outer loop overhead.
- ❏ ***Bank Conflicts:*** nk should be a multiple of 2, otherwise bank conflicts occur causing a penalty of 2*nx cycles. r[] and k[] should be located in separate memory halves on C620x devices, otherwise bank conflicts occur causing a penalty of nx/4 cycles.
- ❏ ***Endian:*** The code is ENDIAN NEUTRAL.
- ❏ ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	(2 * nk + 8) * nx + 5	(without bank conflicts)
Codesize	352 bytes	

4.5 Math

DSP_dotp_sqr *Vector Dot Product and Square*

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

Arguments

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

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

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

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

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

Implementation Notes

- ❏ Unrolled 4 times to maximize multiplier utilization.
- ❏ ***Bank Conflicts:*** No bank conflicts occur.
- ❏ ***Endian:*** The code is ENDIAN NEUTRAL.
- ❏ ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	nx + 20
Codesize	192 bytes

DSP_dotprod *Vector Dot Product*

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

Arguments

x[nx]	First vector array. Must be word aligned.
y[nx]	Second vector array. Must be word aligned.
nx	Number of elements of vector. Must be multiple of 2.
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 C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

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

Special Requirements

- ❏ The input length nx must be a multiple of 2.
- ❏ Arrays x[] and y[] must be aligned on word boundaries.
- ❏ To avoid bank conflicts the input arrays x[] and y[] should be offset by 4 bytes.

Implementation Notes

- ❏ The loop is unrolled once.
- ❏ ***Bank Conflicts:*** No bank conflicts occur if the input arrays x[] and y[] are offset by 4 8 bytes.
- ❏ ***Endian:*** The code is ENDIAN NEUTRAL.
- ❏ ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	nx / 2 + 12 For nx = 40: 32 cycles
Codesize	160 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. Must be word aligned.
nx	Length of input data vector. Must be multiple of 4 and ≥16.
return short	Maximum value of x[].

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

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand 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 must be a multiple of 4 and greater than or equal to 16.
- ❑ x[] must be word aligned.

Implementation Notes

- ❑ ***Bank Conflicts:*** No bank conflicts occur.
- ❑ ***Endian:*** The code is ENDIAN NEUTRAL.
- ❑ ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	nx / 2 + 21
Codesize	224 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.
nx	Length of input data vector. Must be multiple of 3 and ≥ 3.
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.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand 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 3 and greater than or equal to 3.

Implementation Notes

- ❏ The loop is unrolled three times. After finding a new max value, multiply units are used to move value between registers.
- ❏ ***Bank Conflicts:*** No bank conflicts occur.
- ❏ ***Endian:*** The code is LITTLE ENDIAN.
- ❏ ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	2 * nx / 3 + 13 For nx = 108: 85 cycles
Codesize	224 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. Must be word aligned.
nx	Length of input data vector. Must be multiple of 4 and ≥16.
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 C equivalent of the assembly code without restrictions. Note that the assembly code is hand 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 must be a multiple of 4 and greater than or equal to 16.
- ❏ x[] must be word aligned.

Implementation Notes

- ❏ The input data is loaded using word wide loads.
- ❏ ***Bank Conflicts:*** No bank conflicts occur.
- ❏ ***Endian:*** The code is ENDIAN NEUTRAL.
- ❏ ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	nx/2 + 21
Codesize	224 bytes

DSP_mul32 *32-bit Vector Multiply*

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

Arguments

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

Description The function performs a Q.31 x Q.31 multiply and returns the upper 32 bits of the 64-bit result (Q.30 format). 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. Results are accurate to least significant bit.

Algorithm In the comments below, X and Y are the two input values. Xhigh and Xlow represent the upper and lower 16 bits of X. This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand 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 2 and greater than or equal to 8.
- Input and output vectors must be word aligned.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Endian:*** The code is ENDIAN NEUTRAL.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles 1.5 * nx + 26

Codesize 224 bytes

DSP_neg32 *32-bit Vector Negate*

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

Arguments

x[nx]	Pointer to input data vector 1 of size nx with 32-bit elements. Must be word aligned.
r[nx]	Pointer to output data vector of size nx with 32-bit elements. Must be word aligned.
nx	Number of elements of input and output vectors. Must be a multiple of 2 and ≥4.

Description This function negates the elements of a vector (32-bit elements).

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

```
void DSP_neg32(int *x, int *r, int nx)
{
   int i;
   for (i = 0; i < nx; i++)
      r[i] = –x[i];
}
```

Special Requirements

- ❏ nx must be a multiple of 2 and greater than or equal to 4.
- ❏ The arrays x[] and r[] must be word aligned.

Implementation Notes

- ❏ ***Bank Conflicts:*** No bank conflicts occur.
- ❏ ***Endian:*** The code is ENDIAN NEUTRAL.
- ❏ ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	nx + 18
Codesize	128 bytes

DSP_recip16 *16-bit Reciprocal*

Function void DSP_recip16 (short *x, short *rfrac, short *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] * 2^rexp[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 C equivalent of the assembly code without restrictions. Note that the assembly code is hand 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

- The conditional subtract instruction, SUBC, is used for division. SUBC is used once for every bit of quotient needed (15).
- ***Bank Conflicts:*** No bank conflicts occur.
- ***Endian:*** The code is ENDIAN NEUTRAL.
- ***Interruptibility:*** The code is interruptible.

Benchmarks

Cycles 8 * nx + 14

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 2 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 C equivalent of the assembly code without restrictions. Note that the assembly code is hand 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 2 and greater than or equal to 8.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Endian:*** The code is ENDIAN NEUTRAL.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	nx/2 + 19
Codesize	192 bytes

DSP_w_vec *Weighted Vector Sum*

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

Arguments

x[nr]	Vector being weighted. Must be word aligned.
y[nr]	Summation vector. Must be word aligned.
m	Weighting factor (–32767 <= m <= 32767)
r[nr]	Output vector. Must be word aligned.
nr	Dimensions of the vectors. Must be multiple of 4.

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

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

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

Special Requirements

- ❏ nr must be a multiple of 4 and greater than or equal to 4.
- ❏ Vectors x[], y[ ] and r[] must be word aligned.
- ❏ m must not be –32768.

Implementation Notes

- ❏ This loop is unrolled 4x to make full use of the available memory bandwidth.
- ❏ SMPY is used in conjunction with shifts and masks so that the m * x[i] terms may be packed pairs within 32-bit registers. This allows us to use packed-data processing for the rest of the algorithm, thereby maximizing our load/store bandwidth.

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Endian:*** The code is ENDIAN NEUTRAL.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles 3 * nr/4 + 19
For nr = 256: 211 cycles
For nr = 1000: 770 cycles

Codesize 192 bytes

4.5 Matrix

DSP_mat_mul *Matrix Multiplication*

Function void DSP_mat_mul(short *x, int r1, int c1, short *y, int c2, short *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.

Description 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 C equivalent of the assembly code without restrictions. Note that the assembly code is hand 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

- ❏ The outer two loops are unrolled 2x. For odd-sized dimensions, we end up doing extra multiplies along the edges. This offsets the overhead of the nested loop structure, though.
- ❏ The outer two levels of loop nest are collapsed, further reducing the overhead of the looping structure.
- ❏ ***Bank Conflicts:*** Arrays x[] and y[] should be placed in different memory halves on C620x and C670x devices to avoid bank conflicts.
- ❏ ***Endian:*** The code is ENDIAN NEUTRAL.
- ❏ ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles 0.5 * (r1' * c2' * c1) + 3 * (r1' * c2') + 24
with r1' = r1 + (r1&1) and c2' = c2 + (c2&1)
(r1 and c2 rounded up to next even)
For r1= 1, c1= 1, c2= 1: 38 cycles
For r1= 8, c1=20, c2= 8: 856 cycles

Codesize 448 bytes

DSP_mat_trans *Matrix Transpose*

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

Arguments

x[rows*columns]	Pointer to input matrix.
rows	Number of rows in the input matrix.
columns	Number of columns in the input matrix.
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 C equivalent of the assembly code without restrictions. Note that the assembly code is hand 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 none

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Endian:*** The code is ENDIAN NEUTRAL.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	6 * floor(rows * columns / 3) + 11
Codesize	192 bytes

4.6 Miscellaneous

DSP_bexp *Block Exponent Implementation*

Function int DSP_bexp(int *x, unsigned nx)

Arguments

x[nx]	Pointer to input vector of size nx. Must be word aligned.
nx	Number of elements in input vector. Must be multiple of 2 and ≥6.
return int	Return value is the minimum exponent.

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

```
int DSP_bexp(const int *x, unsigned nx)
{
    int i;
    unsigned mask, mag;
    mask = 0;
    for (i = 0; i < nx; i++)
        mask |= x[i] ^ (x[i] >> 31);
    for (mag = 0; (1 << mag) < mask; mag++)
        ;
    return 31 – mag;
}
```

Special Requirements nx must be a multiple of 2 and at least 6.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Endian:*** The code is ENDIAN NEUTRAL.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	nx + 17 For nx = 32: 49 cycles
Codesize	128 bytes

DSP_blk_move *Block Move*

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

Arguments

x [nx]	Block of data to be moved. Must be word aligned.
r [nx]	Destination of block of data. Must be word aligned.
nx	Number of elements in block. Must be multiple of 2 and ≥4.

Description This routine moves nx 16-bit elements from one memory location pointed to by x to another pointed to by r.

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

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

Special Requirements

- ❏ nx must be a multiple of 2 and greater than or equal to 4.
- ❏ x[] and r[] must be word aligned.

Implementation Notes

- ❏ Twin input and output pointers are used.
- ❏ Unrolled 4 times to use parallel LDWs and STWs.
- ❏ Peeled off half-iteration to allow multiple of 2 instead of multiple of 4.
- ❏ Return branch issued from loop kernel to save cycles.
- ❏ ***Bank Conflicts:*** No bank conflicts occur.
- ❏ ***Endian:*** The code is ENDIAN NEUTRAL.
- ❏ ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	MAX(2 * (nx >> 2) + 15, 21) For nx <= 14: 21 cycles For nx = 16: 23 cycles
Codesize	128 bytes

DSP_blk_eswap16 *Endian-swap a block of 16-bit values*

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

Arguments

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

r [nx] Destination array. Must be 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 is meant to facilitate moving big-endian data to a little-endian system or vice-versa.

When the r pointer is non-NULL, the endian-swap occurs out-of-place, similar to a block move. When the r pointer is NULL, the endian-swap occurs in-place, allowing the swap to occur without using any additional storage.

Algorithm

This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand 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 in the very specific case that ”r == NULL” so that the operation occurs in-place.
- The input array and output array are expected to be word aligned, and a multiple of 8 half-words must be processed.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles 0.375 * nx + 19

Codesize 256 bytes

DSP_blk_eswap32 *Endian-swap a block of 32-bit values*

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

Arguments

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

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

nx Number of 32-bit values to swap. Must be multiple of 2 and ≥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 is meant to facilitate moving big-endian data to a little-endian system or vice-versa.

When the r pointer is non-NULL, the endian-swap occurs out-of-place, similar to a block move. When the r pointer is NULL, the endian-swap occurs in-place, allowing the swap to occur without using any additional storage.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand 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 in the very specific case that ”r == NULL” so that the operation occurs in-place.
- ❏ The input array and output array are expected to be word aligned, and a multiple of 2 words must be processed.
- ❏ The input array must be at least 4 words long.

Implementation Notes

- ❏ ***Bank Conflicts:*** No bank conflicts occur.
- ❏ ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles 1.5 * nx + 20

Codesize 224 bytes

DSP_blk_eswap64 *Endian-swap a block of 64-bit values*

Function void blk_eswap64(void *x, void *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 is meant to facilitate moving big-endian data to a little-endian system or vice-versa.

When the r pointer is non-NULL, the endian-swap occurs out-of-place, similar to a block move. When the r pointer is NULL, the endian-swap occurs in-place, allowing the swap to occur without using any additional storage.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand 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 in the very specific case that ”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.
- ❏ The input array must be at least 2 double-words long.

Implementation Notes

- ❏ ***Bank Conflicts:*** No bank conflicts occur.
- ❏ ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles 3 * nx + 20

Codesize 224 bytes

DSP_fltoq15 *Float to Q15 Conversion*

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

Arguments

x[nx] Pointer to floating-point input vector of size nx. x[] should contain the numbers normalized between [–1.0,1.0).

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 will be rounded towards negative infinity. All values that exceed the size limit will be saturated to 0x7fff if value is positive and 0x8000 if value is negative.

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

```
void DSP_fltoq15(float x[], short r[], int nx)
{
int i, a;

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

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

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

Special Requirements nx must be a multiple of 2.

Implementation Notes

- ❏ Loop is unrolled twice.
- ❏ ***Bank Conflicts:*** No bank conflicts occur.
- ❏ ***Endian:*** The code is ENDIAN NEUTRAL.
- ❏ ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles 7 * nx/2 + 12

Codesize 320 bytes

DSP_minerror *Minimum Energy Error Search*

Function int DSP_minerror (short *GSP0_TABLE, short *errCoefs, int max_index)

Arguments

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

Description Performs a dot product on 256 pairs of 9 element vectors and searches for the pair of vectors which produces the maximum dot product result. This is a large part of the VSELP vocoder codebook search.

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

```
int minerr
(
    const short *restrict GSP0_TABLE,
    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 Arrays GSP0_TABLE[] and errCoefs[] must be word aligned.

Implementation Notes

- ❑ Inner loop is completely unrolled.
- ❑ Word-wide loads are used to read in GSP0_TABLE and errCoefs.
- ❑ ***Bank Conflicts:*** Align GSP0_TABLE[] and errCoefs[] at different memory banks to avoid 4 bank conflicts.
- ❑ ***Endian:*** The code is LITTLE ENDIAN.
- ❑ ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles 1189

Codesize 576 bytes

DSP_q15tofl *Q15 to Float Conversion*

Function void DSP_q15tofl (short *x, float *r, int 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 C equivalent of the assembly code without restrictions. Note that the assembly code is hand 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

- Loop is unrolled twice
- ***Bank Conflicts:*** No bank conflicts occur.
- ***Endian:*** The code is ENDIAN NEUTRAL.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	5/2 * nx + 18
Codesize	288 bytes

Appendix A

Performance/Fractional Q Formats

This appendix describes performance considerations related to the C62x DSPLIB and provides information about the Q format used by DSPLIB functions.

Topic **Page**

A.1 Performance Considerations

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

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

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

A.2 Fractional Q Formats

Unless specifically noted, DSPLIB functions use 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

C62x 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 C62x DSPLIB, contact Texas Instruments at dsph@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 ∝-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: 16-bit endian swap

DSP_blk_eswap32: 32-bit endian swap

DSP_blk_eswap64: 64-bit endian swap

DSP_blk_move: Block move

DSP_dotp_sqr: Vector dot product and square.

DSP_dotprod: Vector dot product.

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

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_iir: IIR with 5 coefficients per biquad.

DSP_iirlat_fwd: Forward lattice (radix 2).

DSP_lat_inv: Inverse lattice (radix 2).

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_mat_mul: Matrix multiplication.

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 of the 512 frames in the cache into which the current access is mapped.

indirect addressing: An addressing mode in which an address points to another pointer rather than to the actual data; this mode is prohibited in RISC architecture.

instruction fetch packet: A group of up to eight instructions held in memory for execution by the CPU.

internal interrupt: A hardware interrupt caused by an on-chip peripheral.

interrupt: A signal sent by hardware or software to a processor requesting attention. An interrupt tells the processor to suspend its current operation, save the current task status, and perform a particular set of instructions. Interrupts communicate with the operating system and prioritize tasks to be performed.

interrupt service fetch packet (ISFP): A fetch packet used to service interrupts. If eight instructions are insufficient, the user must branch out of this block for additional interrupt service. If the delay slots of the branch do not reside within the ISFP, execution continues from execute packets in the next fetch packet (the next ISFP).

interrupt service routine (ISR): A module of code that is executed in response to a hardware or software interrupt.

interrupt service table (IST): A table containing a corresponding entry for each of the 16 physical interrupts. Each entry is a single-fetch packet and has a label associated with it.

internal peripherals: Devices connected to and controlled by a host device. The C6x internal peripherals include the direct memory access (DMA) controller, multichannel buffered serial ports (McBSPs), host port interface (HPI), external memory-interface (EMIF), and runtime support timers.

IST: See *interrupt service table.*

L

least significant bit (LSB): The lowest-order bit in a word.

linker: A software tool that combines object files to form an object module, which can be loaded into memory and executed.

little endian: An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher-numbered addresses. Endian ordering is specific to hardware and is determined at reset. See also *big endian*.

M

μ-law companding: See *compress and expand (compand)*.

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.

multichannel buffered serial port (McBSP): An on-chip full-duplex circuit that provides direct serial communication through several channels to external serial devices.

multiplexer: A device for selecting one of several available signals.

N

nonmaskable interrupt (NMI): An interrupt that can be neither masked nor disabled.

O

object file: A file that has been assembled or linked and contains machine language object code.

off chip: A state of being external to a device.

on chip: A state of being internal to a device.

P

peripheral: A device connected to and usually controlled by a host device.

program cache: A fast memory cache for storing program instructions allowing for quick execution.

program memory: Memory accessed through the C6x's program fetch interface.

PWR: Power; see *PWR module*.

PWR module: PWR is an API module that is used to configure the power-down control registers, if applicable, and to invoke various power-down modes.

R

random-access memory (RAM): A type of memory device in which the individual locations can be accessed in any order.

register: A small area of high speed memory located within a processor or electronic device that is used for temporarily storing data or instructions. Each register is given a name, contains a few bytes of information, and is referenced by programs.

reduced-instruction-set computer (RISC): A computer whose instruction set and related decode mechanism are much simpler than those of microprogrammed complex instruction set computers. The result is a higher instruction throughput and a faster real-time interrupt service response from a smaller, cost-effective chip.

reset: A means of bringing the CPU to a known state by setting the registers and control bits to predetermined values and signaling execution to start at a specified address.

RTOS: Real-time operating system.

S

service layer: The top layer of the 2-layer chip support library architecture providing high-level APIs into the CSL and BSL. The service layer is where the actual APIs are defined and is the layer the user interfaces to.

synchronous-burst static random-access memory (SBSRAM): RAM whose contents does not have to be refreshed periodically. Transfer of data is at a fixed rate relative to the clock speed of the device, but the speed is increased.

synchronous dynamic random-access memory (SDRAM): RAM whose contents is refreshed periodically so the data is not lost. Transfer of data is at a fixed rate relative to the clock speed of the device.

syntax: The grammatical and structural rules of a language. All higher-level programming languages possess a formal syntax.

system software: The blanketing term used to denote collectively the chip support libraries and board support libraries.

T

tag: The 18 most significant bits of the program address. This value corresponds to the physical address of the fetch packet that is in that frame.

timer: A programmable peripheral used to generate pulses or to time events.

TIMER module: TIMER is an API module used for configuring the timer registers.

W

word: A multiple of eight bits that is operated upon as a unit. For the C6x, a word is 32 bits in length.

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

W