TMS320C62x DSP Library Programmer's Reference

Literature Number SPRU402
March 2000

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgement, including those pertaining to warranty, patent infringement, and limitation of liability.

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

CERTAIN APPLICATIONS USING SEMICONDUCTOR PRODUCTS MAY INVOLVE POTENTIAL RISKS OF DEATH, PERSONAL INJURY, OR SEVERE PROPERTY OR ENVIRONMENTAL DAMAGE ("CRITICAL APPLICATIONS"). TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS. INCLUSION OF TI PRODUCTS IN SUCH APPLICATIONS IS UNDERSTOOD TO BE FULLY AT THE CUSTOMER'S RISK.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, warranty or endorsement thereof.

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 32 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 ’62x 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, warranty, and 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 ’62x 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 ’62x DSPLIB and provides information about the Q format used by DSPLIB functions.
- Appendix B provides information about warranty issues, 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.

TMS320C6201/C6701 Peripherals Reference Guide (literature number SPRU190) describes common peripherals available on the TMS320C6201/6701 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.

Contents

1 Introduction 1-1

Provides an introduction to the TI 'C62x DSPLIB, shows the organization of the routines contained in the library, and explains the features and benefits of the DSPLIB.

1.1 Introduction to the TI '62x DSPLIB 1-2
1.2 Features and Benefits 1-4

2 Installing and Using DSPLIB 2-1

Provides information on how to install and rebuild the TI 'C62x DSPLIB.

2.1 How to Install DSPLIB 2-2
2.2 Using DSPLIB 2-4
2.2.1 DSPLIB Arguments and Data Types 2-4
2.2.2 Calling a DSPLIB Function From C 2-5
2.2.3 Calling a DSP Function From Assembly 2-5
2.2.4 How DSPLIB is Tested – Allowable Error 2-6
2.2.5 How DSPLIB Deals With Overflow and Scaling Issues 2-6
2.3 How to Rebuild DSPLIB 2-6

3 DSPLIB Function Tables 3-1

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

3.1 Arguments and Conventions Used 3-2
3.2 DSPLIB Functions 3-3
3.3 DSPLIB Function Tables 3-3

4 DSPLIB Reference 4-1

Provides a list of the functions in the DSP library (DSPLIB) organized into functional catagories.

4.1 Adaptive Filtering 4-2
4.2 Correlation 4-4
4.3 FFT 4-6
4.4 Filtering and Convolution 4-14
4.5 Math 4-27
4.6 Matrix 4-40
4.7 Miscellaneous 4-43

A Performance/ Fractional Q Formats A-1

Describes performance considerations related to the ’62x DSPLIB and provides information about the Q format used by DSPLIB functions.

A.1 Performance Considerations A-2
A.2 Fractional Q Formats A-2
A.2.1 Q3.12 Format A-2
A.2.2 Q.15 Format A-3
A.2.3 Q.31 Format A-3

B Warranty and Support B-1

Provides information about warranty issues, software updates, and customer support.

B.1 Warranty B-2
B.2 DSPLIB Software Updates B-2
B.3 DSPLIB Customer Support B-2

C Glossary C-1

Defines terms and abbreviations used in this book.

Tables

2–1 DSPLIB Data Types 2-4
3–1 Argument Conventions 3-2
3–2 Adaptive Filtering 3-3
3–3 Correlation 3-3
3–4 FFT 3-3
3–5 Filtering and Convolution 3-4
3–6 Math 3-4
3–7 Matrix 3-5
3–8 Miscellaneous 3-5
A–1 Q3.12 Bit Fields A-2
A–2 Q.15 Bit Fields A-3
A–3 Q.31 Low Memory Location Bit Fields A-3
A–4 Q.31 High Memory Location Bit Fields A-3

Chapter 1

Introduction

This chapter provides a brief introduction to the TI ’62x 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	1-2
1.2	Features and Benefits	1-4

1.1 Introduction to the TI ’62x DSPLIB

The TI ’C62x DSPLIB is an optimized DSP Function Library for C programmers using TMS320C62x devices. It includes many 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
  - firlms2
- Correlation
  - autocor
- FFT
  - bitrev_cplx
  - radix 2
  - r4fft
- Filtering and convolution
  - fir_cplx
  - fir_gen
  - fir_r4
  - fir_r8
  - fir_sym
  - iir
  - iir_cas4
  - lat_fwd
  - lat_inv

- Math
  - dotp_sqr
  - dotprod
  - maxval
  - maxidx
  - minval
  - mul32
  - neg32
  - recip16
  - vecsumsq
  - w_vec
- Matrix
  - mmul
  - mat_trans
- Miscellaneous
  - bexp
  - blk_move
  - fltoq15
  - minerror
  - q15tofl
  - v_srch

1.2 Features and Benefits

- ❏ Hand-coded assembly-optimized routines
- ❏ C-callable routines, fully compatible with the TI ’C6x compiler
- ❏ Fractional Q15-format operands supported on some benchmarks
- ❏ Benchmarks (time and code)
- ❏ Tested against C model

NOTE: Although the code provided in this software release has been optimized for the TI ’C62x DSP, it will also be operational on other members of the TI ’C6000 DSP family as new devices are made available.

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	2-2
2.2	Using DSPLIB	2-4
2.3	How to Rebuild DSPLIB	2-6

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
                               (asm and headers)
      |
     +-- include
      |    |
      |   +-- header files     Unpacked header files
     +-- doc
           |
          +-- dsp62xlib.pdf    pdf document of API
```

Step 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
```

Code Composer Studio Users

If you set up a project under Code Composer Studio, you could add DSPLIB by choosing dsp62x.lib from the menu *Project –> Add Files to Project*. Also, you should make sure that you link with the correct run-time support library and DSPLIB by having the following lines in your linker command file:

```
–lrts6201.lib
–ldsp62x.lib
```

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

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
Q15	16	fraction	–0.9999694824...	0.9999694824...
Q31	32	fraction	–0.99999999953...	0.99999999953...
IEEE float	32	floating point	1.17549435e–38	3.40282347e+38
IEEE double	64	floating point	2.2250738585072014e–308	1.7976931348623157e+308

Unless specifically noted, DSPLIB operates on Q15-fractional data type elements. Appendix A presents an overview of Fractional Q formats.

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 allowed (unless specifically noted): Source operand can be equal to destination operand to conserve memory.

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 most 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 <autocor.h>
```

in your C file and compile and link using

```
cl6x main.c–z–o autocor_drv.out –lrts6201.lib ldsp62x.lib
```

Code Composer Studio Users

Assuming your C_DIR environment is correctly set up (as mentioned in Section 2.1, *How to Install DSPLIB*), 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 correct run-time support library and DSPLIB by having the following lines in your linker command file:

```
–lrts6201.lib
–ldsp62x.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* (Literature Number 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.

2.3 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 3-2
3.2 DSPLIB Functions 3-3
3.3 DSPLIB Function Tables 3-3
Table 3–1 Argument Conventions 3-2
Table 3–2 Adaptive Filtering 3-3
Table 3–3 Correlation 3-3
Table 3–4 FFT 3-3
Table 3–5 Filtering and Convolution 3-4
Table 3–6 Math 3-4
Table 3–7 Matrix 3-5
Table 3–8 Miscellaneous 3-5

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 firlms2(short h[ ], short x[ ], short b, int nh)	LMS FIR (radix 2)	4-2

Table 3–3. Correlation

Functions	Description	Page
void autocor(short r[ ],short x[ ], int nx, int nr)	Autocorrelation	4-4

Table 3–4. FFT

Functions	Description	Page
void bitrev_cplx (int *x, short *index, int nx)	Complex Bit-Reverse	4-6
void radix2 (int nx, short x[ ], short w[ ])	Complex Forward FFT (radix 2)	4-9
void r4fft (int nx, short x[ ], short w[ ])	Complex Forward FFT (radix 4)	4-11

Table 3–5. Filtering and Convolution

Functions	Description	Page
void fir_cplx (short *x, short *h, short *r, int nh, int nx)	Complex FIR Filter (radix 2)	4-14
void fir_gen (short *x, short *h, short *r, int nh, int nr)	FIR Filter (general purpose)	4-15
void fir_r4 (short *x, short *h, short *r, int nh, int nr)	FIR Filter (radix 4)	4-17
void fir_r8 (short *x, short *h, short *r, int nh, int nr)	FIR Filter (radix 8)	4-18
void fir_sym (short *x, short *h, short *r, int nh, int nr, int s)	Symmetric FIR Filter (radix 8)	4-20
void iir(short *r1, short *x, short *r2, short *h2, short *h1, int nr)	IIR with 5 Coefficients per Biquad	4-21
void iir_cas4(int n, short *c, int *d, int *r)	IIR with 4 Coefficients per Biquad	4-23
int lat_fwd (int r, int nx, short x[ ], short h[ ])	Forward Lattice (radix 2)	4-24
int lat_inv (short h[ ], int nx, short x[ ], int r)	Inverse Lattice (radix 2)	4-25

Table 3–6. Math

Functions	Description	Page
int dotp_sqr(int G, short *x, short *y, int *r, int nx)	Vector Dot Product and Square	4-27
int dotprod(short *x, short *y, int nx)	Vector Dot Product	4-28
short maxval (short *x, int nx)	Maximum Value of a Vector	4-29
int maxidx (short *x, int nx)	Index of the Maximum Element of a Vector	4-30
short minval (short *x, int nx)	Minimum Value of a Vector	4-31
void mul32(int *x, int *y, int *r, short nx)	32-bit Vector Multiply	4-32
void neg32(int *x, int *r, short nx)	32-bit Vector Negate	4-34
void recip16 (short *x, short *rfrac, short *rexp, short nx)	16-bit Reciprocal	4-35
int vecsumsq (short *x, int nx)	Sum of Squares	4-37
void w_vec(short *x, short *y, short m, short *r, short nr)	Weighted Vector Sum	4-38

Table 3–7. Matrix

Functions	Description	Page
void mmul(short *x, short r1, short c1, short *y, short r2, short c2, short *r)	Matrix Multiplication	4-40
void mat_trans(short *x, short rows, short columns, short *r)	Matrix Transpose	4-41

Table 3–8. Miscellaneous

Functions	Description	Page
short bexp(int *x, short nx)	Max Exponent of a Vector (for scaling)	4-43
void blk_move(short *x, short *r, int nx)	Move a Block of Memory	4-44
void fltoq15 (float *x,short *r, short nx)	Float to Q15 Conversion	4-45
int minerror (short *GSP0_TABLE,short *errCoefs, int savePtr_ret)	Minimum Energy Error Search	4-46
void q15tofl (short *x, float *r, short nx)	Q15 to Float Conversion	4-48
int v_srch (int numBasis, short *R, short *wiPtr, short *TABLE, short *wBasisPtr, short *D)	Codebook Search for VSELP	4-49

Chapter 4

DSPLIB Reference

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

Topic **Page**

4.1 Adaptive Filtering 4-2
4.2 Correlation 4-4
4.3 FFT 4-6
4.4 Filtering and Convolution 4-14
4.5 Math 4-27
4.6 Matrix Functions 4-40
4.7 Miscellaneous 4-43

4.1 Adaptive Filtering

4.1.1 firlms2 — *LMS FIR (radix 2)*

long firlms2(short h[ ], short x[ ], short b, int nh)

Arguments

h[nh]	Coefficient Array
x[nh]	Input Array
b	Error from previous FIR
nh	Number of coefficients
return long	return value

Description

Least Mean Square Adaptive Filter. Computes an update of all nh coefficients by adding the weighted error times the inputs to the original coefficients. This assumes single sample input followed by the last nh–1 inputs and 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.

```
long 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) >> 16;
      r += x[i + 1] * h[i];
   }
   return r;
}
```

Special Requirements

- ❏ This routine assumes 16-bit input and output.
- ❏ The number of coefficients must be a multiple of 2.

Implementation Notes

- ❏ The loop is unrolled once.
- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	1.5 * nh + 16
Codesize	288 bytes

4.2 Correlation

4.2.1 autocor — *Autocorrelation*

void autocor(short r[ ],short x[ ], int nx, int nr)

Arguments

r[nr]	Resulting array of autocorrelation
x[nx]	Input array
nx	Length of input array vector multiple of 8
nr	Length of autocorrelation multiple of 2

Description

This routine performs the autocorrelation of the input array x. It is assumed that the length of the input array, x, is a multiple of 8 and the length of the output array, r, is a multiple of 2. The assembly routine performs 2 output samples at a time. This is typically used in VSELP code.

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 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[0] must on a word boundary

Implementation Notes

- ❏ The inner loop is unrolled eight times, thus the length of the input array must be a multiple of eight.
- ❏ The outer loop is unrolled twice, thus the length of output array must be a multiple of 2.

- ❏ The outer loop is conditionally executed in parallel with the inner loop. This allows for a zero overhead outer loop.
- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	(nx/2)*nr + 16 + nr/4 (16 + nr/4 is from memory bank hits)
Codesize	544 bytes

4.3 FFT

4.3.1 bitrev_cplx — *Complex Bit-Reverse*

void bitrev_cplx (int *x, short *index, int nx)

Arguments

x[nx]	Pointer to complex input vector x of size nx
nx	Number of elements in vector x. nx must be a power of 2.
index[ ]	Array of size ≈ sqrt(nx) created by the routine digitrev_index to allow the fast implementation of the bit reversal

Description

This function bit-reverses the position of elements in 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.

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 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;}
                        }
                     }
*************************************************************************

 Use The Routine below To Generate the Index Table for

 Bit/Digit Reversing of Radix–2 and Radix–4 Routines

*************************************************************************

This routine calculates the index for digitrev of length n (length of index is
2^(radix*ceil(k/radix)) where n = 2^k

 in other words

 Either:sqrt(n) when n=2^even# Or: sqrt(2)*sqrt(n) when n=2^odd# [radix 2]
```

```
    sqrt(n) when n=4^even# Or: sqrt(4)*sqrt(n) when n=4^odd# [radix 4]
 Note: the variable “radix” is 2 for radix–2 and 4 for radix–4
*************************************************************************
                        void digitrev_index(short *index, int n, int radix){
                           int   i,j,k;
                           short nbits, nbot, ntop, ndiff, n2, raddiv2;
                           nbits = 0;
                           i = n;
                           while (i > 1){
                              i = i >> 1;
                              nbits++;
                           }

                           raddiv2  = radix >> 1;
                           nbot  = nbits >> raddiv2;
                           nbot  = nbot << raddiv2 – 1;
                           ndiff = nbits & raddiv2;
                           ntop  = nbot + ndiff;
                           n2    = 1 << ntop;

                           index[0] = 0;
                           for ( i = 1, j = n2/radix + 1; i < n2 – 1; i++){
                              index[i] = j – 1;
                              for(k = n2/radix; k*(radix–1) < j;k /= radix)
                                        j –= k*(radix–1);
                              j += k;
                           }
                           index[n2 – 1] = n2 – 1;
                        }
```

Special Requirements

- ❏ nx must be a power of 2
- ❏ The array setup by digitrev_index must be set up before the function 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

- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	(nx / 4 + 2) * 7 + 14
Codesize	480 bytes

4.3.2 radix2 — *Complex Forward FFT (radix 2)*

void radix2 (int nx, short x[ ], short w[ ])

Arguments

nx	Number of complex elements in vector x. Must be a power of 2 such that 4 ≤ nx ≤ 65536.
x[2*nx]	Pointer to input and output sequences. Size 2*nx elements.
w[nx]	Pointer to vector of FFT coefficients of size nx elements.

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 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 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

- ❏ 2 <= nx <= 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], ...
- ❏ w coefficients are stored in the order k*(–cos[0*delta]), k*(–sin[0*delta]), k*(–cos[1*delta]), ... where delta = 2*π/nx, k = 32767

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.
- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	$\log_2(nx) * (15 + 4 * (nx/2 – 2)) + 18 + [mbh]$ [mbh] (memory bank hits) = 0 if x and w are in different memory spaces; otherwise: [mbh] = [N/8–1] on C67xx, except 1 mem bank hit if N=8 [mbh] = [N/4] on C62xx
Codesize	960 bytes

4.3.3 r4fft — *Complex Forward FFT (radix 4)*

void 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 ≤ nx ≤ 65536.
x[2*nx]	Pointer to input and output sequences. Size 2*nx elements.
w[nx]	Pointer to vector of FFT coefficients of size nx elements.

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 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 <= nx <= 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], ...
- w coefficients are stored in the order k*sin[0*delta], k*cos[0*delta], k*sin[1*delta], ... where delta = 2*π/N, k = 32767

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.
- The assembly implementation assumes little-endian byte ordering.
- The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	$\log_4(nx)$ * (21 + 10 * (nx/4 + 1)) + 22 + [mbh] [mbh] (memory bank hits) = 0 if x and w are in different memory spaces. otherwise: [mbh] = [N/8+1] on 67xx, except 1 mem bank hit if N=8 [mbh] = [N/4] on 62xx
Codesize	800 bytes

4.4 Filtering and Convolution

4.4.1 fir_cplx — *Complex FIR Filter (radix 2)*

void fir_cplx (short *x, short *h, short *r, int nh, int nx)

Arguments

x[2*nx]	Pointer to input array of size 2*nx
h[2*nh]	Pointer to coefficient array of size 2*nh
r[2*nx]	Pointer to output array of size 2*nx
nh	Number of coefficients in vector h. Must be a multiple of 2.
nx	Number of samples to calculate

Description

This function implements the FIR filter for complex input data. This function has no memory bank hits regardless of where x, h, and r arrays are located in memory. The filter is nx 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.

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 fir_cplx(short *x, short *h, short *r,
short nh, short nx)
{
   short i,j;
   int imag, real;
   for (i = 0; i < 2*nx; 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

- ❑ nh, the number of coefficients, must be even and greater than or equal to 2.
- ❑ nx, the number of outputs computed, must be greater than or equal to 1.
- ❑ nh * nx must be greater than or equal to 4.

Implementation Notes

- ❑ The inner loop is unrolled twice, thus nh must be a multiple of two.
- ❑ The outer loop is conditionally executed in parallel with the inner loop. This allows for a zero overhead outer loop.
- ❑ Both the inner and outer loops are software pipelined.
- ❑ The assembly implementation assumes little-endian byte ordering.
- ❑ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	2 * nx * nh + 10
Codesize	416 bytes

4.4.2 fir_gen — *FIR Filter (general purpose)*

void 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
r[nr]	Pointer to output array of size nr
nh	Number of coefficients nh ≥ 5
nr	Number of samples to calculate nr ≥ 1

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 assumes the number of filter coefficients is greater than or equal to 5. It operates on 16-bit data with a 32-bit accumulate. This routine has no memory hits regardless of where x, h, and r arrays are located in memory. The filter is nr output samples and nh coefficients. The assembly routine performs 2 output samples at a time.

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 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.
- ❏ nr, the number of outputs computed, must be greater than or equal to 1.

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 the case when the number of output samples is not a multiple of two.
- ❏ Both the inner and outer loops are software pipelined.
- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	(9 + nh) * nr/2 + 15
Codesize	672 bytes

4.4.3 fir_r4 — *FIR Filter (radix 4)*

void 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
r[nr]	Pointer to output array of size nr
nh	Number of coefficients nh ≥ 8, multiple of 4
nr	Number of samples to calculate nr ≥ 2, even

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. This routine has no memory hits regardless of where x, h, and r arrays are located in memory. The filter is nr output samples and nh coefficients. The assembly routine performs 2 output samples at a time.

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 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 even and greater than or equal to 2.

Implementation Notes

- ❏ The inner loop is unrolled four times, thus the number of filter coefficients must be a multiple of four.
- ❏ The outer loop is unrolled twice, thus the number of output samples must be a multiple of two.
- ❏ Both the inner and outer loops are software pipelined.
- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	(9 + nh) * nr/2 + 6
Codesize	576 bytes

4.4.4 fir_r8 — *FIR Filter (radix 8)*

void 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
h[nh]	Pointer to coefficient array of size nh
r[nr]	Pointer to output array of size nr
nh	Number of coefficients nh >= 8, multiple of 8
nr	Number of samples to calculate nr >= 2, even

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 ac-

cumulate. This routine has no memory hits regardless of where x, h, and r arrays are located in memory. The filter is nr output samples and nh coefficients. The assembly routine performs 2 output samples at a time.

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 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 even and greater than or equal to 2.

Implementation Notes

- ❏ The inner loop is unrolled eight times, thus the number of filter coefficients must be a multiple of eight.
- ❏ The outer loop is unrolled twice, thus the number of output samples must be a multiple of two.
- ❏ Both the inner and outer loops are software pipelined.
- ❏ The outer loop is conditionally executed in parallel with the inner loop. This allows for a zero overhead outer loop.
- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	nh * nr/2 + 13
Codesize	512 bytes

4.4.5 fir_sym — *Symmetric FIR Filter (radix 8)*

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

Arguments

x[nr+2nh]	Pointer to input array of size nr + 2nh
h[2*nh+1]	Pointer to coefficient array of size 2nh+1
r[nr]	Pointer to output array of size nr
nh	Number of coefficients nh ≥ 8, multiple of 8
nr	Number of samples to calculate nr ≥ 2, even
s	Number of insignificant digits to truncate

Description

This symmetric FIR filter assumes the number of filter coefficients is 2nh + 1 and the number of output samples is a multiple of 2. It operates on 16-bit data with a 40-bit accumulation. This routine has no memory hits. However, h should be word-aligned. The filter is nr output samples and 2nh+1 coefficients. The assembly routine performs 2 output samples at a time.

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 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 even and greater than or equal to 2.
- ❏ nr, the number of outputs computed, must be even and greater than or equal to 2.
- ❏ h, the coefficient array, must be word aligned.

Implementation Notes

- ❏ The inner loop is unrolled twice, thus nh must be a multiple of two.
- ❏ The outer loop is unrolled twice, thus the number of output samples must be a multiple of two.
- ❏ Both the inner and outer loops are software pipelined.
- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	(3nh/2+ 10)* nr/2 + 17
Codesize	480 bytes

4.4.6 iir — *IIR with 5 Coefficients per Biquad*

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

Arguments

r1[nr]	Output array (used)
x[nr+4]	Input array
r2[nr]	Output array (stored)
h1[4]	Filter coefficients
h2[5]	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 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 hits r1 must be aligned on the next halfword boundary following the alignment of x. Otherwise, there is a total of nr memory hits (once per outer loop.)

Implementation Notes

- ❏ The inner loop is completely unrolled and software pipelined (i.e., each time the 5-cycle loop is executed, the inner loop of the C code is executed.)
- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	5*nr + 16
Codesize	416 bytes

4.4.7 iir_cas4 — IIR with 4 Coefficients per Biquad

void iir_cas4(int n, short *c, int *d, int *r)

Arguments

n	Number of cascaded biquads
c[4n]	array containing –a1, –a2, b1, b2 biquad coefs
d[2n]	array of the delayed states within biquads
r[2]	inputs r[0] and r[1] (also outputs)

Description

This function performs a cascaded biquad IIR filter with the direct form II structure (4 multiplies.) It performs two samples at a time. Coefficients are stored in the order –a1, –a2, b1, b2 for each successive biquad located in the c array. Both outputs are stored back to the location of the inputs r[0] and r[1]. The inputs and outputs are 32-bit values while the coefficients are 16-bit values.

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 iir_cas4(int n, short *c, int *d, int *r)
{
   int  k0, k1, i;
   for (i = 0; i < n; i++) {
      k0   = c[4*i+1]*(d[2*i+1] >> 16) +
      c[4*i+0]*(d[2*i+0]>> 16) +r[0];
      r[0] = c[4*i+3]*(d[2*i+1] >> 16) +
      c[4*i+2]*(d[2*i+0]>> 16) + k0;
      d[2*i+1] = k0;
      k1   = c[4*i+1]*(d[2*i+0] >> 16) +
      c[4*i+0]*(k0 >> 16) + r[1];
      r[1] = c[4*i+3]*(d[2*i+0] >> 16) +
      c[4*i+2]*(k0 >> 16) + k1;
      d[2*i+0] = k1;
   }
}
```

Special Requirements The d and c array pointers must be placed on opposite word boundaries to avoid memory hits.

Implementation Notes

- ❏ The loop is written so that one biquad for each of the two inputs is completed every time through the loop. There is an extra priming delay for the second input so that the biquads new delayed state k0 is calculated based on the first input (i.e., the second input is being processed by the biquad preceding the biquad which is processing the first input.)
- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	4*n + 16
Codesize	256 bytes

4.4.8 lat_fwd *Forward Lattice (radix 2)*

int lat_fwd (int r, int nx, short x[ ], short h[ ])

Arguments

r	Result of forward synthesis
nx	Number of data samples nx >=2, even
x[nx]	Array of filter grains
h[nx]	Array of coefficients
return int	Return value

Description This routine implements a forward synthesis lattice filter and stores the result in r. The filter consists of nx data samples. The value of r is calculated by doing a multiply accumulate on the coefficients and filter gains. New coefficients are also calculated. The x and h arrays contain 16-bit 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 lat_fwd(int r, int nx, short x[ ], short h[ ])
{
   int             i;
   r –= h[nx – 1] * x[nx – 1];
   for (i = nx – 2; i >= 0; i––) {
      r –= h[i] * x[i];
      h[i + 1] = h[i] + ((x[i] * (r >> 16)) >> 16);
   }
   h[0] = r >> 16;
   return r;
}
```

Special Requirements

- ❏ nx, the number of data samples, must be even and greater than or equal to 2.
- ❏ Vectors h and x should be aligned on different half word boundaries to avoid memory hits.

Implementation Notes

- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	2*nx + 18
Codesize	160 bytes

4.4.9 lat_inv *Inverse Lattice (radix 2)*

int lat_inv (short h[ ], int nx, short x[ ], int r)

Arguments

h[nx]	Array of coefficients
nx	Number of coefficients nx >=2, even
x[nx]	Array of filter grains
r	Result of inverse analysis
return int	Result of inverse analysis

Description

This routine implements an inverse analysis lattice filter and stores the result in r. The filter consists of nx stages. The value of r is calculated by doing a multiply accumulate on the coefficients and filter gains. New coefficients are also calculated.

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 lat_inv(short h[ ], int nx, short x[ ], int r)
{
   int            i;
   short          c, a;
   c = r >> 16;
   for (i = 0; i < nx; i++) {
      a = h[i] + ((x[i] * (r >> 16)) >> 16);
      r += h[i] * x[i];
      h[i] = c;
      c = a;
   }
   return r;
}
```

Special Requirements

- ❏ nx, the number of data samples, must be even and ≥ 2.
- ❏ Vectors h and x should be aligned on different half word boundaries to avoid memory hits.

Implementation Notes

- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	1.5*nx + 10
Codesize	256 bytes

4.5 Math

4.5.1 dotp_sqr — *Vector Dot Product & Square*

int dotp_sqr(int G, short *x, short *y, int *r, int nx)

Arguments

G	Calculated value of G (used in the VSELP coder)
x[nx]	First vector array
y[nx]	Second vector array
r	Result of vector dot product of x and y
nx	Number of elements in vector x
return int	New value of G

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

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

Special Requirements n should be an even number.

Implementation Notes

- ❏ Vectors x and y should be aligned on opposite word boundaries to avoid memory hits.
- ❏ Load words are used to load two 16-bit values at a time

- ❏ The loop is unrolled once
- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	nx + 8
Codesize	192

4.5.2 dotprod *Vector Dot Product*

int dotprod(short *x, short *y, int nx)

Arguments

x[nx]	First vector array
y[nx]	Second vector array
nx	Number of elements of vector
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 dotprod(short x[ ],short y[ ], int nx)
{
   int sum;
   int i;
   sum = 0;
   for(i=0; i<nx; i++){
      sum += (x[i] * y[i]);
   }
   return (sum);
}
```

Special Requirements

- ❏ nx is an even number greater than 2
- ❏ Vectors x and y should be aligned on word boundaries

Implementation Notes

- ❏ Load words are used to load two 16-bit values at a time
- ❏ The loop is unrolled once
- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	nx / 2 + 8
Codesize	192 bytes

4.5.3 maxval *Maximum Value of a Vector*

short maxval (short *x, int nx)

Arguments

x[nx]	Pointer to input vector of size nx
nx	Length of input data vector
return short	Maximum value of a vector

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

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

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

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

Special Requirements

- ❏ nx is a multiple of 6
- ❏ Vector x[ ] should be aligned on word boundary

Implementation Notes

- ❏ The loop is unrolled 6 times.
- ❏ After finding a new max value, this function uses the multiply (M) units to move value between registers.
- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	nx / 2 +13
Codesize	320 bytes

4.5.4 maxidx — *Index of the Maximum Element of a Vector*

int maxidx (short *x, int nx)

Arguments

x[nx]	Pointer to input vector of size nx
nx	Length of input data vector (nx >= 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 the value. After finding a new max value, it uses multiply (M) units to move value between registers.

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 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 >= 3
- ❑ nx is a multiple of 3
- ❑ vector x[ ] should be aligned on half-word boundaries

Implementation Notes

- ❑ The loop is unrolled 3 times.
- ❑ After finding a new max value, this function uses the multiply (M) units to move value between registers.
- ❑ The assembly implementation assumes little-endian byte ordering.
- ❑ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	2 * (nx / 3) +12
Codesize	288 bytes

4.5.5 minval *Minimum Value of a Vector*

short minval (short *x, int nx)

Arguments

x [nx]	Pointer to input vector of size nx
nx	Length of input data vector
return short	Maximum value of a vector

Description This routine finds the minimum value of a vector and returns the value. After finding a new minimum value, it uses multiply units to move value between registers.

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

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

Special Requirements

- ❏ nx is a multiple of 6
- ❏ Vector x should be aligned on a word boundary
- ❏ Little-Endian configuration

Implementation Notes

- ❏ The loop is unrolled 6 times.
- ❏ After finding a new min value, it uses multiply units to move value between registers.
- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	nx / 2 +13
Codesize	320 bytes

4.5.6 mul32 — *32-bit Vector Multiply*

void mul32(int *x, int *y, int *r, short nx)

Arguments

x[nx]	Pointer to input data vector 1 of size nx. In-place processing allowed (x can be = y = r)
y[nx]	Pointer to input data vector 2 of size nx
r[nx]	Pointer to output data vector of size nx
nx	Number of elements in input and output vectors.

Description

This function multiplies two 32-bit Q31 vector elements and produces a vector of 32-bit Q31 numbers.

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 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

- ❏ Inputs and output vectors are assumed to be in Q31 format.
- ❏ Output is accurate to least significant bit.

Implementation Notes

- ❏ Results are accurate up to least significant bit.
- ❏ 16-bit multiplies followed by shifts and adds were used to emulate a 32-bit multiply.
- ❏ The innermost loop was unrolled 4 times and pipelined. Conditional stores are used so that nx does not have to be a multiple of 4.
- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	17 + 6 * [(nx–2) / 4]
Codesize	704 bytes

4.5.7 neg32 — *32-bit Vector Negate*

void neg32(int *x, int *r, short nx)

Arguments

x[nx]	Pointer to input data vector 1 of size nx with 32-bit elements. In-place processing allowed (x can = r)
r[nx]	Pointer to output data vector of size nx with 32-bit elements.
nx	Number of elements of input and output vectors.

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 neg32(int *x, int *r, short nx)
{
   short i;
   for(i=nx; i>0; i--)
      *(r++)=-*(x++);
}
```

Special Requirements

- ❏ Input and output are 32-bit signed integers.
- ❏ To prevent memory bank conflicts, if both x and r are in the same data section (memory space), they should both start at 64-bit boundaries.

Implementation Notes

- ❏ The loop is unrolled 7 times and pipelined. Conditional stores are used so that nx does not have to be a multiple of 7.
- ❏ The stores are scheduled to eliminate memory bank hits when vectors are properly aligned.

- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	13 + 7 * [nx / 7]
Codesize	416 bytes

4.5.8 recip16 — *16-bit Reciprocal*

void recip16 (short *x, short *rfrac, short *rexp, short nx)

Arguments

x[nx]	Pointer to input data vector of size nx
rfrac[nx]	Pointer to 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 a Q15 number. Since the reciprocal is always greater than 1, it returns an exponent such that:

$(rfrac[i] * 2^{rexp[i]}) = true\ reciprocal$

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 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

- ❑ x and rfrac are Q15 format.
- ❑ 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 to 0, the procedure will return a fractional part of 7FFFh and an exponent of 16.

Implementation Notes

- ❑ The conditional subtract instruction, SUBC, is used for division. SUBC is used once for every bit of quotient needed (15).
- ❑ The kernel processes two divisions in parallel, but note that nx does not have to be a multiple of 2.
- ❑ The assembly implementation assumes little-endian byte ordering.
- ❑ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	14 + 17 * [nx/2]
Codesize	672 bytes

4.5.9 vecsumsq — *Sum of Squares*

int vecsumsq (short *x, int nx)

Arguments

x[nx]	Input vector
nx	Number of elements in x
return int	Sum of the squares

Description This routine takes one vector with (nx) number of elements. It calculates the square of each element and accumulates the results and returns the sum.

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

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

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

Special Requirements Number of elements nx >= 2 (even OR odd)

Implementation Notes

- ❑ The assembly implementation assumes little-endian byte ordering.
- ❑ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	nx / 2 + 9
Codesize	224 bytes

4.5.10 w_vec — *Weighted Vector Sum*

void w_vec(short *x, short *y, short m, short *r, short nr)

Arguments

x[nr]	Vector being weighted
y[nr]	Summation vector
m	Weighting factor
r[nr]	Output vector
nr	Dimensions of the vectors

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 w_vec(short x[ ],short y[ ],short m,
short r[ ],short nr)
{
   short i;

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

Special Requirements

- ❏ nr >= 3
- ❏ Vectors x and y should be aligned on word boundary.

Implementation Notes

- ❏ Loading the input in word to double the performance.
- ❏ Using AND (.L) instead of EXTU (.S) to obtain y[2*i] from the word containing y[2*i+1] and y[2*i] to reduce the requirement on .S unit.
- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	nr + 10 (even nr) nr + 11 (odd nr)
Codesize	256 bytes

4.6 Matrix

4.6.1 mmul — *Matrix Multiplication*

void mmul(short *x, short r1, short c1, short *y, short r2, short c2, short *r)

Arguments

x [r1*c1]	Pointer to input matrix of size c1*r1
r1	number of rows in matrix x
c1	number of columns in matrix x
y [r2*c2]	Pointer to input matrix of size r2*c2
r2	number of rows in matrix y
c2	number of columns in matrix y
r [r1*c2]	Pointer to output matrix r of size r1*c2

Description Multiplies matrices x and y and stores the result in 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 mmul(short *x, short r1, short c1, short *y,
short r2, short c2, short *r)
{
   short temp,i,j,k;
   short *yp;

   if((c1==r2)&(c1>0)&(c2>0)&(r1>0))  /*verify
                                /* parameters */
   {
      yp=y;
      for(i=0; i<r1; i++)      /* top to bottom */
      {
         for(j=0; j<c2; j++)   /* left to right */
         {
            yp=y+j;
            temp=0;
            for(k=0; k<c1; k++)   /* multiply and
                                  /* add */
            {
               temp+=(*x)*(*yp);
```

```
                x++;
                yp+=c2;
            }
            x-=c1;
            *(r++)=temp;       /* store sum */
        }
        x+=c1;
    }
  }
}
```

Special Requirements

- ❏ Procedure will exit on invalid matrix dimensions.
- ❏ Elements of matrix assumed to be 16-bit signed integers.
- ❏ r1, c1, r2, and c2 are shorts.
- ❏ In-place processing not allowed.
- ❏ Memory bank hits will occur in different amounts dependent on the dimensions and alignment of the matrices. This will cause roughly a 10% increase in execution time. To eliminate this performance hit, place input matrices in different memory spaces.

Implementation Notes

- ❏ The innermost loop multiplies and adds 4 times. The ADDs are conditional so that the matrix dimensions do not have to be multiples of 4.
- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	15 + ((6 * [c1/4] + 8) * c2 + 6) * r1
Codesize	512 bytes

4.6.2 mat_trans *Matrix Transpose*

void mat_trans (short *x, short rows, short columns, short *r)

Arguments

x[rows*columns]	Pointer to input matrix. In place processing is not allowed.
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.

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

Special Requirements

- ❏ Rows and columns must not be negative numbers.
- ❏ Matrices are assumed to have 16-bit elements.

Implementation Notes

- ❏ The two loops are combined in one. The kernel is unrolled and pipelined using conditional stores to not to write outside of output array.
- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	11 + 6 * [rows * columns / 3]
Codesize	192 bytes

4.7 Miscellaneous

4.7.1 bexp — *Block Exponent Implementation*

short bexp(int *x, short nx)

Arguments

x[nx]	Pointer to input vector of size nx.
nx	Number of elements in input vector. Must be a value between 1 and 24576 inclusive.
return short	Return value is the maximum exponent that may be used in scaling

Description

Computes the exponents (number of extra sign bits) of all values in the input vector 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.

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

Special Requirements

- ❑ $1 \le nx \le 24576$
- ❑ Vector is Q31 format

Implementation Notes

- ❑ This function keeps track of 6 minimum exponent values in the inner loop and finally the 6 exponent values are compared to find the minimum among those 6. Conditional instructions were used so that nx does not have to be a multiple of 6.

- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	nx>6 : 11 + 6 * [nx/6]
	nx36: 15
Codesize	544 bytes

4.7.2 blk_move *Block Move*

void blk_move(short *x, short *r, int nx)

Arguments

x [nx]	Block of data to be moved
r [nx]	Destination of block of data
nx	Number of elements in block

Description This routine moves nx 16-bit elements from one memory location pointed by x to another pointed 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 blk_move(short *x,short *r, int nx)
{
   short *tmpPtr;
   short *tmpPtr2;

   tmpPtr = x;
   tmpPtr2 = r – 1;
   for (tmpPtr = x; tmpPtr < x + nx; tmpPtr++)
      *++tmpPtr2 = *tmpPtr;
}
```

Special Requirements

- ❏ nx ≥ 6
- ❏ nx is a multiple of 2

Implementation Notes

- ❏ Two load words are used to load four 16-bit values at a time
- ❏ The loop is unrolled four times
- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	nx / 2 + 5 (nx multiple of 4)
	nx / 2 + 6 (other values of nx)
Codesize	160 bytes

4.7.3 fltoq15 *Float to Q15 Conversion*

void fltoq15 (float *x, short *r, short nx)

Arguments

x[nx]	Pointer to floating-point input vector of size nx. x should contain the numbers normalized between [–1,1).
r[nx]	Pointer to output data vector of size nx containing the q15 equivalent of vector x.
nx	Length of input and output data vectors

Description

Convert the IEEE floating point numbers stored in vector x into Q15 format numbers stored in vector r. All values that exceed the size limit will be saturated to a Q15 1 or –1 depending on sign. (0x7fff if value is positive, 0x8000 if value is negative) All values too small to be correctly represented will be truncated to 0.

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 fltoq15(float x[ ], short r[ ],short nx)
{
   int i;
   for(i=0;i<nx;i++) {
      r[i] = 0x7fff * x[i];
   }
}
```

Special Requirements

- ❑ nx needs to be a multiple of 2
- ❑ nx >= 2
- ❑ No memory bank hits under any conditions

Implementation Notes

- ❑ Loop unrolled once
- ❑ The assembly implementation assumes little-endian byte ordering.
- ❑ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	7*nx + 12
Codesize	352 bytes

4.7.4 minerror *Minimum Energy Error Search*

int minerror (short *GSP0_TABLE,short *errCoefs, int savePtr_ret)

Arguments

GSP0_TABLE[256]	GSP0 terms array
errCoefs[9]	Array of error coefficients
savePtr_ret	Index of pair of vectors giving max dotprod
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 minerror(short *errCoefs, short *GSP0_TABLE,
int savePtr_ret)
{
   int      val, maxval;
   int      i, j;
   short    *tmpPtr;
   short    *tmpPtr2;
   short    *endPtr;
   short    *endPtr2;
   short    *savePtr;

   #define  GSP0_TERMS 9
   #define  GSP0_NUM 256

   maxval = –50.0;
   tmpPtr = GSP0_TABLE;
   for (endPtr = tmpPtr + GSP0_TERMS*GSP0_NUM;
   tmpPtr < endPtr; ){
     val = 0;
     tmpPtr2 = errCoefs;

     for(endPtr2=tmpPtr2+GSP0_TERMS;tmpPtr2<endPtr2;
     tmpPtr2++){
        val += *tmpPtr * *tmpPtr2;
        tmpPtr++;
     }
     if (val > maxval) {
        maxval = val;
        savePtr = tmpPtr;
     }
   }
   savePtr_ret = (savePtr – GSP0_TABLE)*2;
   return (maxval);
}
```

Special Requirements

- ❑ Number of error coefficients is 9
- ❑ Number of GSP0 terms is 256

Implementation Notes

- ❑ The inner loop is unrolled 2 times.
- ❑ No memory bank hits given errCoefs & GSP0_TABLE are both on even or both on odd word boundaries (4 hits if not)

- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	(256/2) * 9 + 14 = 1166
Codesize	576 bytes

4.7.5 q15tofl *Q15 to Float Conversion*

void q15tofl (short *x, float *r, short nx)

Arguments

x[nx]	Pointer to Q15 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

Description Converts the values stored in vector x in Q15 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 q15tofl(short *x, float *r, short nx)
{
   int i;

   for (i=0;i<nx;i++)
      r[i] = (float) x[i] / 0x7fff;
}
```

Special Requirements nx must be a multiple of 2

Implementation Notes

- ❏ Loop unrolled once
- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	17 * (nx / 2) + 6
Codesize	256 bytes

4.7.6 v_srch — *Codebook Search for VSELP*

int v_srch (short *wiPtr, short *wBasisPtr, int numBasis, short * TABLE, short *R, short *D)

Arguments

numBasis	Number of weighted basis vectors
R[ ]	Array of Rm values, the cross correlations between weighted speech and weighted basis vectors
wiPtr[ ]	Weighted speech vectors
TABLE[ ]	Table of codewords
wbasisPtr[ ]	Weighted basis vectors
D[ ]	Matrix of Dmj values, the cross correlations between the weighted basis vectors
return int	best codeword

Description Performs VSELP vocoder codebook search. This routine performs the entire v_srch.c function as written by Motorola. It involves calculating correlations between weighted basis vectors and weighted speech vector (Rms), C0, and 0.25 * sum of Djj for G0. It then calculates all Dmj and finishes calculating G0. It then initializes the best vector to be code vector zero and performs search by finding the vector that produces the highest C^2/G value.

Algorithm The C source code for this was written by Motorola Systems Research Laboratories and is authorized by Motorola for the use of development of North American digital cellular standards. As such, the C code cannot be shown here.

Special Requirements Vectors wiPtr and wBasisPtr have to be aligned on opposite word boundaries to avoid memory hits.

Implementation Notes

- ❏ The assembly implementation assumes little-endian byte ordering.
- ❏ The assembly implementation of this function disables interrupts for its entire duration and hence non-interruptible.

Benchmarks

Cycles	Loop 1: 342 ; Loop 2: 640 ; Loop 3: 2089 Total: 3071 cycles
Codesize	1504 bytes

Appendix A

Performance/ Fractional Q Formats

This appendix describes performance considerations related to the ’62x DSPLIB and provides information about the Q format used by DSPLIB functions.

Topic **Page**

A.1 Performance Considerations A-2

A.2 Fractional Q Formats A-2

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

Warranty and Support

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

Topic **Page**

B.1 Warranty B-2

B.2 DSPLIB Software Updates B-2

B.3 DSPLIB Customer Support B-2

B.1 Warranty

The ’C62x DSPLIB is distributed free of charge.

BETA RELEASE SPECIAL DISCLAIMER: This DSPLIB software release is preliminary (Beta). It is intended for evaluation only. Testing and characterization has not been fully completed. Production release typically follows the Beta release but there are no explicit guarantees.

B.2 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.3 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.

autocor: Autocorrelation

B

bexp: Block exponent implementation

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*.

bitrev_cplx: Complex bit-reverse.

blk_move: Block move

block: The three least significant bits of the program address. These correspond to the address within a fetch packet of the first instruction being addressed.

board support library (BSL): The BSL is a set of application programming interfaces (APIs) consisting of target side DSP code used to configure and control board level peripherals.

boot: The process of loading a program into program memory.

boot mode: The method of loading a program into program memory. The ’C6x DSP supports booting from external ROM or the host port interface (HPI).

BSL: *See board support library.*

byte: A sequence of eight adjacent bits operated upon as a unit.

C

cache: A fast storage buffer in the central processing unit of a computer.

cache controller: System component that coordinates program accesses between CPU program fetch mechanism, cache, and external memory.

CCS: Code Composer Studio.

central processing unit (CPU): The portion of the processor involved in arithmetic, shifting, and Boolean logic operations, as well as the generation of data- and program-memory addresses. The CPU includes the central arithmetic logic unit (CALU), the multiplier, and the auxiliary register arithmetic unit (ARAU).

chip support library (CSL): The CSL is a set of application programming interfaces (APIs) consisting of target side DSP code used to configure and control all on-chip peripherals.

clock cycle: A periodic or sequence of events based on the input from the external clock.

clock modes: Options used by the clock generator to change the internal CPU clock frequency to a fraction or multiple of the frequency of the input clock signal.

code: A set of instructions written to perform a task; a computer program or part of a program.

coder-decoder or compression/decompression (codec): A device that codes in one direction of transmission and decodes in another direction of transmission.

compiler: A computer program that translates programs in a high-level language into their assembly-language equivalents.

compress and expand (compand): A quantization scheme for audio signals in which the input signal is compressed and then, after processing, is reconstructed at the output by expansion. There are two distinct companding schemes: A-law (used in Europe) and μ-law (used in the United States).

control register: A register that contains bit fields that define the way a device operates.

control register file: A set of control registers.

CSL: See *chip support library*.

D

device ID: Configuration register that identifies each peripheral component interconnect (PCI).

digital signal processor (DSP): A semiconductor that turns analog signals—such as sound or light—into digital signals, which are discrete or discontinuous electrical impulses, so that they can be manipulated.

direct memory access (DMA): A mechanism whereby a device other than the host processor contends for and receives mastery of the memory bus so that data transfers can take place independent of the host.

DMA : See *direct memory access*.

DMA source: The module where the DMA data originates. DMA data is read from the DMA source.

DMA transfer: The process of transferring data from one part of memory to another. Each DMA transfer consists of a read bus cycle (source to DMA holding register) and a write bus cycle (DMA holding register to destination).

dotp_sqr: Vector dot product and square.

dotprod: Vector dot product.

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.*

fir_cplx: Complex FIR filter (radix 2).

firlms2: LMS FIR (radix 2).

fir_r4: FIR filter (radix 4).

fir_r8: FIR filter (radix 8).

fir_gen: FIR filter (general purpose).

fir_sym: Symmetric FIR filter (radix 8).

flag: A binary status indicator whose state indicates whether a particular condition has occurred or is in effect.

fltoq 15: Float to Q15 conversion.

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

iir: IIR with 5 coefficients per biquad.

iir_cas4: IIR with 4 coefficients per biquad.

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

lat_inv: Inverse lattice (radix 2).

lat_fwd: Forward lattice (radix 2).

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.

mat_trans: Matrix transpose.

maxidx: Index of the maximum element of a vector.

maxval: Maximum value of a vector.

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.

minerror: Minimum energy error search.

minval: Minimum value of a vector.

mmul: Matrix multiplication.

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.

mul32: 32-bit vector multiply.

N

neg32: 32-bit vector negate.

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.

Q

q15tofl: Q15 to float conversion.

R

radix2: Complex forward FFT (radix 2)

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

recip16: 16-bit reciprocal.

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 micro-programmed 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.*

r4fft: Complex forward FFT (radix 4)

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.

V

v_srch: Codebook search for VSELP.

vecsumsq: Sum of squares.

W

w_vec: Weighted vector sum.

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

A-law companding, defined, C-1
adaptive filtering, functions, 3-3
adaptive filtering functions, DSPLIB reference, 4-2
address, defined, C-1
API, defined, C-1
application programming interface, defined, C-1
argument conventions, 3-2
arguments, DSPLIB, 2-4
assembler, defined, C-1
assert, defined, C-1
autocor
    defined, C-1
    DSPLIB reference, 4-4

B

bexp, DSPLIB reference, 4-43
big endian, defined, C-1
bit, defined, C-1
bitrev_cplx, defined, C-1
bitrev_cplx, DSPLIB reference, 4-6
blk_move
    defined, C-2
    DSPLIB reference, 4-44
block, defined, C-2
board support library, defined, C-2
boot, defined, C-2
boot mode, defined, C-2
BSL, defined, C-2
byte, defined, C-2

C

cache, defined, C-2
cache controller, defined, C-2
CCS, defined, C-2
central processing unit (CPU), defined, C-2
chip support library, defined, C-2
clock cycle, defined, C-2
clock modes, defined, C-2
code, defined, C-2
coder-decoder, defined, C-3
compiler, defined, C-3
compress and expand (compand), defined, C-3
control register, defined, C-3
control register file, defined, C-3
correlation, functions, 3-3
correlation functions, DSPLIB reference, 4-4
CSL, defined, C-3
customer support, B-2

D

data types, DSPLIB, table, 2-4
device ID, defined, C-3
digital signal processor (DSP), defined, C-3
direct memory access (DMA)
    defined, C-3
    source, defined, C-3
    transfer, defined, C-3
DMA, defined, C-3
datp_sqr, C-3
dotp_sqr, DSPLIB reference, 4-27
dotprod
    defined, C-3
    DSPLIB reference, 4-28

DSPLIB
argument conventions, table, 3-2
arguments, 2-4
arguments and data types, 2-4
calling a function from Assembly, 2-5
calling a function from C, 2-5
Code Composer Studio users, 2-5
customer support, B-2
data types, table, 2-4
features and benefits, 1-4
fractional Q formats, A-2
functional categories, 1-2
functions, 3-3
adaptive filtering, 3-3
correlation, 3-3
FFT (fast Fourier transform), 3-3
filtering and convolution, 3-4
math, 3-4
matrix, 3-5
miscellaneous, 3-5
how DSPLIB deals with overflow and scaling, 2-6
how to install, 2-2
under Code Composer Studio, 2-3
how to rebuild DSPLIB, 2-6
include directory, 2-3
introduction, 1-2
lib directory, 2-2
performance considerations, A-2
Q.3.12 bit fields, A-2
Q.3.12 format, A-2
Q.3.15 bit fields, A-3
Q.3.15 format, A-3
Q.31 format, A-3
Q.31 high-memory location bit fields, A-3
Q.31 low-memory location bit fields, A-3
reference, 4-1
software updates, B-2
testing, how DSPLIB is tested, 2-6
using DSPLIB, 2-4
warranty, B-2

DSPLIB reference
adaptive filtering functions, 4-2
autocor, 4-4
bexp, 4-43
bitrev_cplx, 4-6
blk_move, 4-44
correlation functions, 4-4
dotp_sqr, 4-27
dotprod, 4-28
FFT functions, 4-6

DSPLIB reference (continued)
filtering and convolution functions, 4-14
fir_cplx, 4-14
fir_r4, 4-17
fir_r8, 4-18
fir_sym, 4-20
firlms2, 4-2
fltoq15, 4-45
fir_gen, 4-15
iir, 4-21
iir_cas4, 4-23
lat_fwd, 4-24
lat_inv, 4-25
mat_trans, 4-41
math functions, 4-27
matrix functions, 4-40
maxidx, 4-30
maxval, 4-29
minerror, 4-46
minval, 4-31
miscellaneous functions, 4-43
mmul, 4-40
mul32, 4-32
neg32, 4-34
q15tofl, 4-48
r4fft, 4-11
radix2, 4-9
recip16, 4-35
v_srch, 4-49
vecsumsq, 4-37
w_vec, 4-38

E

evaluation module, defined, C-4
external interrupt, defined, C-4
external memory interface (EMIF), defined, C-4

F

fetch packet, defined, C-4
FFT, defined, C-4
FFT (fast Fourier transform), functions, 3-3
FFT functions, DSPLIB reference, 4-6
filtering and convolution, functions, 3-4
filtering and convolution functions,
DSPLIB reference, 4-14

fir_cplx
  defined, C-4
  DSPLIB reference, 4-14
fir_gen
  defined, C-4
  DSPLIB reference, 4-15
fir_r4
  defined, C-4
  DSPLIB reference, 4-17
fir_r8
  defined, C-4
  DSPLIB reference, 4-18
fir_sym
  defined, C-4
  DSPLIB reference, 4-20
firlms2
  defined, C-4
  DSPLIB reference, 4-2
flag, defined, C-4
fltoq 15, defined, C-4
fltoq15, DSPLIB reference, 4-45
fractional Q formats, A-2
frame, defined, C-4
function
  calling a DSPLIB function from Assembly, 2-5
  calling a DSPLIB function from C, 2-5
    Code Composer Studio users, 2-5
functions, DSPLIB, 3-3

G

GIE bit, defined, C-5

H

HAL, defined, C-5
host, defined, C-5
host port interface (HPI), defined, C-5
HPI, defined, C-5

I

iir
  defined, C-5
  DSPLIB reference, 4-21
iir_cas4
  defined, C-5
  DSPLIB reference, 4-23
include directory, 2-3
index, defined, C-5
indirect addressing, defined, C-5
installing DSPLIB, 2-2
instruction fetch packet, defined, C-5
internal interrupt, defined, C-5
internal peripherals, defined, C-6
interrupt, defined, C-6
interrupt service fetch packet (ISFP), defined, C-6
interrupt service routine (ISR), defined, C-6
interrupt service table (IST), defined, C-6
IST, defined, C-6

L

lat_fwd, DSPLIB reference, 4-24
lat_fwd**Empty**, defined, C-6
lat_inv
  defined, C-6
  DSPLIB reference, 4-25
least significant bit (LSB), defined, C-6
lib directory, 2-2
linker, defined, C-6
little endian, defined, C-6

M

μ-law companding, defined, C-7
maskable interrupt, defined, C-7
mat_trans
  defined, C-7
  DSPLIB reference, 4-41
math, functions, 3-4
math functions, DSPLIB reference, 4-27
matrix, functions, 3-5
matrix functions, DSPLIB reference, 4-40
maxidx
  defined, C-7
  DSPLIB reference, 4-30
maxval
  defined, C-7
  DSPLIB reference, 4-29
memory map, defined, C-7

memory-mapped register, defined, C-7
minerror
  defined, C-7
  DSPLIB reference, 4-46
minval
  defined, C-7
  DSPLIB reference, 4-31
miscellaneous, functions, 3-5
miscellaneous functions, DSPLIB reference, 4-43
mmul
  defined, C-7
  DSPLIB reference, 4-40
most significant bit (MSB), defined, C-7
mul32
  defined, C-7
  DSPLIB reference, 4-32
multichannel buffered serial port (McBSP), defined, C-7
multiplexer, defined, C-7

N

neg32
  defined, C-7
  DSPLIB reference, 4-34
nonmaskable interrupt (NMI), defined, C-7

O

object file, defined, C-8
off chip, defined, C-8
on chip, defined, C-8
overflow and scaling, 2-6

P

performance considerations, A-2
peripheral, defined, C-8
program cache, defined, C-8
program memory, defined, C-8
PWR, defined, C-8
PWR module, defined, C-8

Q

Q.3.12 bit fields, A-2
Q.3.12 format, A-2
Q.3.15 bit fields, A-3
Q.3.15 format, A-3
Q.31 format, A-3
Q.31 high-memory location bit fields, A-3
Q.31 low-memory location bit fields, A-3
q15tofl
  defined, C-8
  DSPLIB reference, 4-48

R

r4fft
  defined, C-9
  DSPLIB reference, 4-11
radix2
  defined, C-8
  DSPLIB reference, 4-9
random-access memory (RAM), defined, C-8
rebuilding DSPLIB, 2-6
recip16
  defined, C-8
  DSPLIB reference, 4-35
reduced-instruction-set computer (RISC), defined, C-9
register, defined, C-9
reset, defined, C-9
routines, DSPLIB functional categories, 1-2
RTOS, defined, C-9

S

service layer, defined, C-9
software updates, B-2
STDINC module, defined, C-9
synchronous dynamic random-access memory (SDRAM), defined, C-9
synchronous-burst static random-access memory (SBSRAM), defined, C-9
syntax, defined, C-9
system software, defined, C-9

T

tag, defined, C-10

testing, how DSPLIB is tested, 2-6

timer, defined, C-10

TIMER module, defined, C-10

U

using DSPLIB, 2-4

V

v_srch
  defined, C-10
  DSPLIB reference, 4-49
vecsumsq
  defined, C-10
  DSPLIB reference, 4-37

W

w_vec
  defined, C-10
  DSPLIB reference, 4-38
warranty, B-2
word, defined, C-10