TMS320C55x DSP Library
Programmer’s Reference

SPRU422A
August 2000
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 acknowledgment, 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.

Customers are responsible for their applications using TI components.

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.
About This Manual

The Texas Instruments TMS20C55x DSPLIB is an optimized DSP Function Library for C programmers on TMS20C55x devices. It includes over 50 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 shorten significantly your DSP application development time.

Related Documentation

- Digital Signal Processing with the TMS320 Family (SPR012)
- TMS320C55x DSP CPU Reference Guide (SPRU371)
- TMS320C55x Optimizing C Compiler User’s Guide (SPRU281)

Trademarks

TMS320, TMS20C55x, and C55x are trademarks of Texas Instruments.

Matlab is a trademark of Mathworks, Inc.
Acknowledgments

DSPLIB includes code contributed by the following people:

David Alter  
Tom Horner  
Jelena Nikolic  
Gunter Schmer  
Douglas Hall  
Adrienne Jaffe  
Chuck Brokish  
Syed Maroof  
Nils Paz  
Stephen Lau  
Mark Adams  
Rekha Radhakrishnan  
Frank Vogler  
David Smalley  
Herve Marechal  
Timo Kuisma  
Karen Baldwin  
Rosemarie Piedra  
Yves Wenzinger  
Cesar Iovescu  
Aaron Aboagye  
Ulrich Schmidt  
Joe George  
Dipa Rao  
Li Yuan
# Contents

1 Introduction ................................................. 1-1  
1.1 DSP Routines ............................................. 1-2  
1.2 Features and Benefits ................................. 1-2  
1.3 DSPLIB: Quality Freeware That You can Build On and Contribute To 1-2  

2 Installing DSPLIB ........................................... 2-1  
2.1 DSPLIB Content ........................................... 2-2  
2.2 How to Install DSPLIB ................................. 2-2  
  2.2.1 Read README.1ST File for Specific Details of Release 2-2  
2.3 How to Rebuild DSPLIB ................................. 2-3  
  2.3.1 For Full Rebuild of 55xdsp.lib ....................... 2-3  
  2.3.2 For Partial Rebuild of 55xdsp.lib .................... 2-3  

3 Using DSPLIB .................................................. 3-1  
3.1 DSPLIB Arguments and Data Types ....................... 3-2  
  3.1.1 DSPLIB Arguments .................................. 3-2  
  3.1.2 DSPLIB Data Types .................................. 3-2  
3.2 Calling a DSPLIB Function from C ....................... 3-3  
3.3 Calling a DSPLIB Function from Assembly Language Source Code 3-4  
3.4 Where to Find Sample Code ............................... 3-4  
3.5 How DSPLIB is Tested – Allowable Error ................. 3-5  
3.6 How DSPLIB Deals with Overflow and Scaling Issues .... 3-5  
3.7 Where DSPLIB Goes from Here ............................ 3-7  

4 Function Descriptions ........................................ 4-1  
4.1 Arguments and Conventions Used ......................... 4-2  
4.2 DSPLIB Functions ......................................... 4-3  

5 DSPLIB Benchmarks and Performance Issues .................. 5-1  
5.1 What DSPLIB Benchmarks are Provided .................... 5-2  
5.2 Performance Considerations ............................... 5-2  

6 Licensing, Warranty, and Support ............................ 6-1  
6.1 Licensing and Warranty ................................... 6-2  
6.2 DSPLIB Software Updates ................................ 6-2  
6.3 DSPLIB Customer Support ................................ 6-2  

A Overview of Fractional Q Formats ........................... A-1  
A.1 Q3.12 Format .............................................. A-2  
A.2 Q.15 Format ............................................... A-2  
A.3 Q.31 Format ............................................... A-2  

B Calculating the Reciprocal of a Q15 Number ................ B-1  

C Texas Instruments License Agreement for DSP Code ........... C-1
Figures

4–1 dbuffer Array in Memory at Time j ................................................................. 4-21
4–2 x Array in Memory ......................................................................................... 4-22
4–3 r Array in Memory ......................................................................................... 4-22
4–4 x Array in Memory ......................................................................................... 4-26
4–5 r Array in Memory ......................................................................................... 4-26
4–6 h Array in Memory ......................................................................................... 4-26
4–7 x Array in Memory ......................................................................................... 4-28
4–8 r Array in Memory ......................................................................................... 4-28
4–9 h Array in Memory ......................................................................................... 4-28
4–10 x Array in Memory ....................................................................................... 4-30
4–11 r Array in Memory ....................................................................................... 4-30
4–12 h Array in Memory ....................................................................................... 4-30
4–13 dbuffer Array in Memory at Time j ............................................................... 4-41
4–14 x Array in Memory ....................................................................................... 4-41
4–15 r Array in Memory ....................................................................................... 4-41
4–16 dbuffer Array in Memory at Time j ............................................................... 4-45
4–17 x Array in Memory ....................................................................................... 4-45
4–18 r Array in Memory ....................................................................................... 4-45
4–19 dbuffer Array in Memory at Time j ............................................................... 4-56
4–20 x Array in Memory ....................................................................................... 4-56
4–21 r Array in Memory ....................................................................................... 4-56
4–22 dbuffer Array in Memory at Time j ............................................................... 4-62
4–23 x Array in Memory ....................................................................................... 4-62
4–24 r Array in Memory ....................................................................................... 4-62
Tables

4–1 Function Descriptions ................................................................. 4–2
4–2 Summary Table ................................................................. 4–3
A–1 Q3.12 Bit Fields ................................................................. A–2
A–2 Q.15 Bit Fields ................................................................. A–2
A–3 Q.31 Low Memory Location Bit Fields ............................................ A–2
A–4 Q.31 High Memory Location Bit Fields ............................................ A–2
The Texas Instruments TMS320C55x™ DSPLIB is an optimized DSP Function Library for C programmers on TMS320C55x devices. It includes over 50 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 considerable faster than equivalent code written in standard ANSI C language. In addition, by providing ready-to-use DSP functions, TI DSPLIB can shorten significantly your DSP application development time.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 DSP Routines</td>
<td>1-2</td>
</tr>
<tr>
<td>1.2 Features and Benefits</td>
<td>1-2</td>
</tr>
<tr>
<td>1.3 DSPLIB: Quality Freeware That You Can Build On and Contribute To</td>
<td>1-2</td>
</tr>
</tbody>
</table>
1.1 DSP Routines

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

The routines included within the library are organized into eight different functional categories:

- Fast-Fourier Transforms (FFT)
- Filtering and convolution
- Adaptive filtering
- Correlation
- Math
- Trigonometric
- Miscellaneous
- Matrix

1.2 Features and Benefits

- Hand-coded assembly optimized routines
- C-callable routines fully compatible with the TI C55x™ compiler
- Fractional Q15-format operand supported
- Complete set of examples on usage provided
- Benchmarks (time and code) provided
- Tested against Matlab™ scripts

1.3 DSPLIB: Quality Freeware That You Can Build On and Contribute To

DSPLIB is a free-of-charge product. You can use, modify, and distribute TI C55x DSPLIB for usage on TI C55x DSPs with no royalty payments. See Licensing and Warranty, section 6.1, and Where DSPLIB Goes from Here, section 3.7, for details.
Chapter 2

Installing DSPLIB

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 DSPLIB Contents</td>
<td>2-2</td>
</tr>
<tr>
<td>2.2 How to Install DSPLIB</td>
<td>2-2</td>
</tr>
<tr>
<td>2.3 How to Rebuild DSPLIB</td>
<td>2-3</td>
</tr>
</tbody>
</table>
2.1 DSPLIB Content

The TI DSPLIB software consists of 4 parts:

1) a header file for C programmers:
   `dsplib.h`

2) One object library:
   `55xdsp.lib`

3) One source library to allow function customization by the end user
   `55xdsp.src`

4) Example programs and linker command files used under the “55x_test” sub-directory.

2.2 How to Install DSPLIB

2.2.1 Read README.1ST File for Specific Details of Release

**Step 1:** De-archive DSPLIB

DSPLIB is distributed in the form of an executable self-extracting ZIP file (55xdsplib.exe). The zip file automatically restores the DSPLIB individual components in the same directory you execute the self extracting file. Following is an example on how to install DSPLIB, just type:

`55xdsplib.exe –d`

The DSPLIB directory structure and content you will find is:

- `55xdsplib (dir)`
  - `55xdsp.lib`: use for standards short-call mode
  - `blt55x.bat`: re-generate `55xdsp.lib` based on `55xdsp.src`
  - `examples(dir)`: contains one subdirectory for each routine included in the library where you can find complete test cases

- `include(dir)`
  - `dsplib.h`: include file with data types and function prototypes
  - `tms320.h`: include file with type definitions to increase TMS320 portability
  - `misc.h`: include file with useful miscellaneous definitions

- `doc(dir)`
  - `dsplib.pdf`: DSPLIB Application Report in PDF format
  - `55x_src (dir)`: contains assembly source files for functions
How to Rebuild DSPLIB

2.3 How to Rebuild DSPLIB

2.3.1 For Full Rebuild of 55xdsp.lib

To rebuild 55xdsp.lib, execute the blt55x.bat. This will overwrite any existing 55xdsp.lib.

2.3.2 For Partial Rebuild of 55xdsp.lib

(modification of a specific DSPLIB function, for example fir.asm)

1) Extract the source for the selected function from the source archive:

   \texttt{ar55 x 55xdsp.src fir.asm}

2) Re-assemble your new fir.asm assembly source file:

   \texttt{asm55 -g fir.asm}

3) Replace the object, fir.obj, in the dsplib.lib object library with the newly formed object:

   \texttt{ar55 r 55xdsp.lib fir.obj}

Step 2: Relocate library file

Copy the C55x DSPLIB object library file, 55xdsp.lib, to your C5500 runtime support library folder.

For example, if your TI C5500 tools are located in \texttt{c:\ti\c5500\cgtools\bin} and c runtime support libraries (rts55.lib etc.) in \texttt{c:\ti\c5500\cgtools\lib}, copy 55xdsp.lib to this folder. This allows the C55x compiler/linker to find 55xdsp.lib.
### Chapter 3

**Using DSPLIB**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 DSP Arguments and Data Types</td>
<td>3-2</td>
</tr>
<tr>
<td>3.2 Calling a DSPLIB Function from C</td>
<td>3-3</td>
</tr>
<tr>
<td>3.3 Calling a DSPLIB Function from Assembly Language</td>
<td></td>
</tr>
<tr>
<td>Source Code</td>
<td>3-4</td>
</tr>
<tr>
<td>3.4 Where to Find Sample Code</td>
<td>3-4</td>
</tr>
<tr>
<td>3.5 How DSPLIB is Tested — Allowable Error</td>
<td>3-5</td>
</tr>
<tr>
<td>3.6 How DSPLIB Deals with Overflow and Scaling Issues</td>
<td>3-5</td>
</tr>
<tr>
<td>3.7 Where DSPLIB Goes from Here</td>
<td>3-7</td>
</tr>
</tbody>
</table>
3.1 DSPLIB Arguments and Data Types

3.1.1 DSPLIB Arguments

DSPLIB functions typically operate over vector operands for greater efficiency. Though these routines can be used to process short arrays or scalars (unless a minimum size requirement is noted), the execution times will be longer in those cases.

- **Vector stride is always equal 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 a Re-Im format.
- **In-place computation is allowed (unless specifically noted)**: Source operand can be equal to destination operand to conserve memory.

3.1.2 DSPLIB Data Types

DSPLIB handles the following fractional data types:

- **Q.15 (DATA)**: A Q.15 operand is represented by a *short* data type (16 bit) that is predefined as `DATA`, in the `dsplib.h` header file.

- **Q.31 (LDATA)**: A Q.31 operand is represented by a *long* data type (32 bit) that is predefined as `LDATA`, in the `dsplib.h` header file.

- **Q.3.12**: Contains 3 integer bits and 12 fractional bits.

Unless specifically noted, DSPLIB operates on Q15-fractional data type elements. Appendix A presents an overview of Fractional Q formats.
3.2 Calling a DSPLIB Function from C

In addition to installing the DSPLIB software, to include a DSPLIB function in your code you have to:

- Include the dsplib.h include file
- Link your code with the DSPLIB object code library, 55xdsp.lib.
- Use a correct linker command file describing the memory configuration available in your C55x board.

For example, the following code contains a call to the recip16 and q15tofl routines in DSPLIB:

```c
#include "dsplib.h"

DATA x[3] = { 12398, 23167, 564};

DATA r[NX];
DATA rexp[NX];
float rf1[NX];
float rf2[NX];

void main()
{
    short i;

    for (i=0; i<NX; i++)
    {
        r[i] = 0;
        rexp[i] = 0;
    }

    recip16(x, r, rexp, NX);
    q15tofl(r, rf1, NX);

    for (i=0; i<NX; i++)
    {
        rf2[i] = (float)rexp[i] * rf1[i];
    }

    return;
}
```
In this example, the q15tofl DSPLIB function is used to convert Q15 fractional values to floating point fractional values. However, in many applications, your data is always maintained in Q15 format so that the conversion between float and Q15 is not required.

The above code, ug.c, is available to you in the /doc/code subdirectory. To compile and link this code with 55xdsp.lib, issue the following command:

```
c155 –pk –g –o3 –i. ug.c –z –v0 ld3.cmd 55xdsp.lib –m ug.map –oug.out
```

The examples presented in this document have been tested using the Texas Instruments C55x Simulator. Customization may be required to use it with a different simulator or development board.

Refer to the TMS320C55x Optimizing C Compiler User’s Guide (SPRU281).

### 3.3 Calling a DSPLIB Function from Assembly Language Source Code

The TMS320C55x 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 with the Texas Instruments C55x C compiler calling conventions. Refer to the TMS320C55x Optimizing C Compiler User’s Guide, if a more in-depth explanation is required.

Realize that the TI DSPLIB is not an optimal solution for assembly-only programmers. Even though DSPLIB functions can be invoked from an assembly program, the result may not be optimal due to unnecessary C-calling overhead.

### 3.4 Where to Find Sample Code

You can find examples on how to use every single function in DSPLIB, in the examples subdirectory. This subdirectory contains one subdirectory for each function. For example, the examples/araw subdirectory contains the following files:

- **araw_t.c**: main driver for testing the DSPLIB acorr (raw) function.
- **test.h**: contains input data(a) and expected output data(yraw) for the acorr (raw) function as. This test.h file is generated by using Matlab scripts.
- **test.c**: contains function used to compare the output of araw function with the expected output data.
- **ftest.c**: contains function used to compare two arrays of float data types.
- **ltest.c**: contains function used to compare two arrays of long data types.
- **ld3.cmd**: an example of a linker command you can use for this function.
3.5 How DSPLIB is Tested – Allowable Error

Version 1.0 of DSPLIB is tested against Matlab scripts. Expected data output has been generated from Matlab that uses double-precision (64-bit) floating-point operations (default precision in Matlab). Test utilities have been added to our test main drivers to automate this checking process. Note that a maximum absolute error value (MAXERROR) is passed to the test function, to set the trigger point to flag a functional error.

We consider this testing methodology a good first pass approximation. Further characterization of the quantization error ranges for each function (under random input) as well as testing against a set of fixed-point C models is planned for future releases. We welcome any suggestions you may have on this respect.

3.6 How DSPLIB Deals with Overflow and Scaling Issues

One of the inherent difficulties of programming for fixed-point processors is determining how to deal with overflow issues. Overflow occurs as a result of addition and subtraction operations when the dynamic range of the resulting data is larger than what the intermediate and final data types can contain.

The methodology used to deal with overflow should depend on the specifics of your signal, the type of operation in your functions, and the DSP architecture used. In general, overflow handling methodologies can be classified in five categories: saturation, input scaling, fixed scaling, dynamic scaling, and system design considerations.

It's important to note that a TMS320C55x architectural feature that makes overflow easier to deal with is the presence of guard bits in all four accumulators. The 40-bit accumulators provide eight guard bits that allow up to 256 consecutive multiply-and-accumulate (MAC) operations before an accumulator overrun – a very useful feature when implementing, for example, FIR filters.

There are 4 specific ways DSPLIB deals with overflow, as reflected in each function description:

- **Scaling implemented for overflow prevention**: In this type of function, DSPLIB scales the intermediate results to prevent overflow. Overflow should not occur as a result. Precision is affected but not significantly. This is the case of the FFT functions, in which scaling is used after each FFT stage.
How DSPLIB Deals with Overflow and Scaling Issues

- **No scaling implemented for overflow prevention**: In this type of function, DSPLIB does not scale to prevent overflow due to the potentially strong effect in data output precision or in the number of cycles required. This is the case, for example, of the MAC-based operations like filtering, correlation, or convolutions. The best solution on those cases is to design your system, for example your filter coefficients with a gain less than 1 to prevent overflow. In this case, overflow could happen unless you input scale or you design for no overflow.

- **Saturation implemented for overflow handling**: In this type of function, DSPLIB has enabled the TMS320C55x 32-bit saturation mode (SATD bit = 1). This is the case of certain basic math functions that require the saturation mode to be enabled.

- **Not applicable**: In this type of function, due to the nature of the function operations, there is no overflow.

- **DSPLIB reporting of overflow conditions (overflow flag)**: Due to the sometimes unpredictable overflow risk, most DSPLIB functions have been written to return an overflow flag (oflag) as an indication of a potentially dangerous 32-bit overflow. However, because of the guard-bits, the C55x is capable of handling intermediate 32-bit overflows and still produce the correct final result. Therefore, the oflag parameter should be taken in the context of a warning but not a definitive error.

As a final note, DSPLIB is provided also in source format to allow customization of DSPLIB functions to your specific system needs.
3.7 Where DSPLIB Goes from Here

We anticipate DSPLIB to improve in future releases in the following areas:

- **Increased number of functions**: We anticipate the number of functions in DSPLIB will increase. We welcome user-contributed code. If during the process of developing your application you develop a DSP routine that seems like a good fit to DSPLIB, let us know. We will review and test your routine and possibly include it in the next DSPLIB software release. Your contribution will be acknowledged and recognized by TI in the Acknowledgments section. Use this opportunity to make your name known by your DSP industry peers. Simply email your contribution To Whom It May Concern: dsph@ti.com and we will contact you.

- **Increased Code portability**: DSPLIB looks to enhance code portability across different TMS320-based platforms. It is our goal to provide similar DSP libraries for other TMS320™ devices, working in conjunction with C55x compiler intrinsics to make C-developing easier for fixed-point devices. However, it's anticipated that a 100% portable library across TMS320™ devices may not be possible due to normal device architectural differences. TI will continue monitoring DSP industry standardization activities in terms of DSP function libraries.
## Function Descriptions

### Topic  Page

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Arguments and Conventions Used</td>
<td>4-2</td>
</tr>
<tr>
<td>4.2 DSPLIB Functions</td>
<td>4-3</td>
</tr>
</tbody>
</table>
4.1 Arguments and Conventions Used

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

Table 4–1. Function Descriptions

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x, y)</td>
<td>argument reflecting input data vector</td>
</tr>
<tr>
<td>(r)</td>
<td>argument reflecting output data vector</td>
</tr>
<tr>
<td>(nx, ny, nr)</td>
<td>arguments reflecting the size of vectors (x, y,) and (r) respectively. In functions where (nx = nr = nr), only (nx) has been used.</td>
</tr>
<tr>
<td>(h)</td>
<td>Argument reflecting filter coefficient vector (filter routines only)</td>
</tr>
<tr>
<td>(nh)</td>
<td>Argument reflecting the size of vector (h)</td>
</tr>
<tr>
<td>(DATA)</td>
<td>data type definition equating a short, a 16-bit value representing a Q15 number. Usage of (DATA) instead of short is recommended to increase future portability across devices.</td>
</tr>
<tr>
<td>(LDATA)</td>
<td>data type definition equating a long, a 32-bit value representing a Q31 number. Usage of (LDATA) instead of long is recommended to increase future portability across devices.</td>
</tr>
<tr>
<td>(ushort)</td>
<td>Unsigned short (16 bit). You can use this data type directly, because it has been defined in (dsplib.h)</td>
</tr>
</tbody>
</table>
4.2 DSPLIB Functions

The routines included within the library are organized into 8 different functional categories:

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

Table 4–2 lists the functions by these 8 functional categories.

Table 4–2. Summary Table

(a) FFT

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void cfft (DATA *x, ushort nx, ushort scale)</td>
<td>Radix-2 complex forward FFT – MACRO</td>
</tr>
<tr>
<td>void cifft (DATA *x, ushort nx, ushort scale)</td>
<td>Radix-2 complex inverse FFT – MACRO</td>
</tr>
<tr>
<td>void cbrev (DATA *x, DATA *r, ushort n)</td>
<td>Complex bit-reverse function</td>
</tr>
<tr>
<td>void rfft (DATA *x, ushort nx, ushort scale)</td>
<td>Radix-2 real forward FFT – MACRO</td>
</tr>
<tr>
<td>void rifft (DATA *x, ushort nx, ushort scale)</td>
<td>Radix-2 real inverse FFT – MACRO</td>
</tr>
</tbody>
</table>

(b) Filtering and Convolution

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ushort fir (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh)</td>
<td>FIR direct form</td>
</tr>
<tr>
<td>ushort fir2 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh)</td>
<td>FIR direct form (Optimized to use DUAL–MAC)</td>
</tr>
<tr>
<td>ushort firs (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh2)</td>
<td>Symmetric FIR direct form (generic routine)</td>
</tr>
<tr>
<td>ushort cfir (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh)</td>
<td>Complex FIR direct form</td>
</tr>
<tr>
<td>ushort convol (DATA *x, DATA *h, DATA *r, ushort nr, ushort nh)</td>
<td>Convolution</td>
</tr>
<tr>
<td>ushort convol1 (DATA *x, DATA *h, DATA *r, ushort nr, ushort nh)</td>
<td>Convolution (Optimized to use DUAL–MAC)</td>
</tr>
</tbody>
</table>
### DSPLIB Functions

**Table 4–2. Summary Table (Continued)**

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ushort convol2 (DATA *x, DATA *h, DATA *r, ushort nr, ushort nh)</td>
<td>Convolution (Optimized to use DUAL–MAC)</td>
</tr>
<tr>
<td>ushort iircas4 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nbiq, ushort nx)</td>
<td>IIR cascade direct form II. 4 coefficients per biquad.</td>
</tr>
<tr>
<td>ushort iircas5 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nbiq, ushort nx)</td>
<td>IIR cascade direct form II. 5 coefficients per biquad</td>
</tr>
<tr>
<td>ushort iircas51 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nbiq, ushort nx)</td>
<td>IIR cascade direct form I. 5 coefficients per biquad</td>
</tr>
<tr>
<td>ushort iirlat (DATA *x, DATA *h, DATA *r, DATA *pbuffer, int nx, int nh)</td>
<td>Lattice inverse IIR filter</td>
</tr>
<tr>
<td>ushort firlat (DATA *x, DATA *h, DATA *r, DATA *pbuffer, int nx, int nh)</td>
<td>Lattice forward FIR filter</td>
</tr>
<tr>
<td>ushort firdec (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nh, ushort nx, ushort D)</td>
<td>Decimating FIR filter</td>
</tr>
<tr>
<td>ushort firinterp (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nh, ushort nx,ushort l)</td>
<td>Interpolating FIR filter</td>
</tr>
<tr>
<td>ushort hilb16 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh)</td>
<td>FIR Hilbert Transformer</td>
</tr>
<tr>
<td>ushort iir32 (DATA *x, LDATA *h, DATA *r, LDATA *dbuffer, ushort nbiq, ushort nr)</td>
<td>Double-precision IIR filter</td>
</tr>
</tbody>
</table>

#### (c) Adaptive filtering

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ushort dlms (DATA *x, DATA *h, DATA *r, DATA *des, DATA *dbuffer, DATA step, ushort nh, ushort nx)</td>
<td>LMS FIR (delayed version)</td>
</tr>
</tbody>
</table>

#### (d) Correlation

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ushort acorr (DATA *x, DATA *r, ushort nx, ushort nr, type)</td>
<td>Autocorrelation (positive side only) – MACRO</td>
</tr>
<tr>
<td>ushort corr (DATA *x, DATA *y, DATA *r, ushort nx, ushort ny, type)</td>
<td>Correlation (full-length)</td>
</tr>
</tbody>
</table>
### Table 4–2. Summary Table (Continued)

#### (e) Trigonometric

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ushort sine (DATA *x, DATA *r, ushort nx)</td>
<td>Sine of a vector</td>
</tr>
<tr>
<td>ushort atan2_16 (DATA *i, DATA *q, DATA *r, ushort nx)</td>
<td>Four quadrant inverse tangent of a vector</td>
</tr>
<tr>
<td>ushort atan16 (DATA *x, DATA *r, ushort nx)</td>
<td>Arctan of a vector</td>
</tr>
</tbody>
</table>

#### (f) Math

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ushort add (DATA *x, DATA *y, DATA *r, ushort nx, ushort scale)</td>
<td>Optimized vector addition</td>
</tr>
<tr>
<td>ushort expn (DATA *x, DATA *r, ushort nx)</td>
<td>Exponent of a vector</td>
</tr>
<tr>
<td>short bexp (DATA *x, ushort nx)</td>
<td>Exponent of all values in a vector</td>
</tr>
<tr>
<td>ushort logn (DATA *x, LDATA *r, ushort nx)</td>
<td>Natural log of a vector</td>
</tr>
<tr>
<td>ushort log_2 (DATA *x, LDATA *r, ushort nx)</td>
<td>Log base 2 of a vector</td>
</tr>
<tr>
<td>ushort log_10 (DATA *x, LDATA *r, ushort nx)</td>
<td>Log base 10 of a vector</td>
</tr>
<tr>
<td>short maxidx (DATA *x, ushort nx)</td>
<td>Index for maximum magnitude in a vector</td>
</tr>
<tr>
<td>short maxval (DATA *x, ushort nx)</td>
<td>Maximum magnitude in a vector</td>
</tr>
<tr>
<td>void maxvec (DATA *x, ushort nx, DATA *r_val, DATA *r_idx)</td>
<td>Index and value of the maximum element of a vector</td>
</tr>
<tr>
<td>short minidx (DATA *x, ushort nx)</td>
<td>Index for minimum magnitude in a vector</td>
</tr>
<tr>
<td>short minval (DATA *x, ushort nx)</td>
<td>Minimum element in a vector</td>
</tr>
<tr>
<td>void minvec (DATA *x, ushort nx, DATA *r_val, DATA *r_idx)</td>
<td>Index and value of the minimum element of a vector</td>
</tr>
<tr>
<td>ushort mul32 (LDATA *x, LDATA *y, LDATA *r, ushort nx)</td>
<td>32-bit vector multiply</td>
</tr>
<tr>
<td>short neg (DATA *x, DATA *r, ushort nx)</td>
<td>16-bit vector negate</td>
</tr>
<tr>
<td>short neg32 (LDATA *x, LDATA *r, ushort nx)</td>
<td>32-bit vector negate</td>
</tr>
<tr>
<td>short power (DATA *x, LDATA *r, ushort nx)</td>
<td>Sum of squares of a vector (power)</td>
</tr>
<tr>
<td>void recip16 (DATA *x, DATA *r, DATA *rexp, ushort nx)</td>
<td>Vector reciprocal</td>
</tr>
<tr>
<td>void ldiv16 (LDATA *x, DATA *y, DATA *r, DATA *rexp, ushort nx)</td>
<td>32-bit by 16-bit long division</td>
</tr>
</tbody>
</table>
Table 4–2. Summary Table (Continued)

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ushort <code>sqrt_16</code> (DATA *x, DATA *r, short nx)</td>
<td>Square root of a vector</td>
</tr>
<tr>
<td>short <code>sub</code> (DATA *x, DATA *y, DATA *r, ushort nx, ushort scale)</td>
<td>Vector subtraction</td>
</tr>
</tbody>
</table>

(g) Matrix

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ushort <code>mmul</code> (DATA *x1, short row1, short col1, DATA *x2, short row2, short col2, DATA *r)</td>
<td>matrix multiply</td>
</tr>
<tr>
<td>ushort <code>mtrans</code> (DATA *x, short row, short col, DATA *r)</td>
<td>matrix transpose</td>
</tr>
</tbody>
</table>

(h) Miscellaneous

<table>
<thead>
<tr>
<th>Functions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ushort <code>fltoq15</code> (float *x, DATA *r, ushort nx)</td>
<td>Floating-point to Q15 conversion</td>
</tr>
<tr>
<td>ushort <code>q15tofl</code> (DATA *x, float *r, ushort nx)</td>
<td>Q15 to floating-point conversion</td>
</tr>
<tr>
<td>ushort <code>rand16</code> (DATA *r, ushort nr)</td>
<td>Random number generation</td>
</tr>
<tr>
<td>void <code>rand16init(void)</code></td>
<td>Random number generation initialization</td>
</tr>
</tbody>
</table>
**Autocorrelation**

```c
ushort oflag = acorr (DATA *x, DATA *r, ushort nx, ushort nr, type)
```

(defined in araw.asm, abias.asm, aubias.asm)

**Arguments**

- **x [nx]** Pointer to real input vector of nx real elements. nx \( \geq \) nr
- **r [nr]** Pointer to real output vector containing the first nr elements of the positive side of the autocorrelation function of vector a. r must be different than a (in-place computation is not allowed).
- **nx** Number of real elements in vector x
- **nr** Number of real elements in vector r
- **type** Autocorrelation type selector. Types supported:
  - If type = raw, r contains the raw autocorrelation of x
  - If type = bias, r contains the biased autocorrelation of x
  - If type = unbias, r contains the unbiased autocorrelation of x
- **oflag** Overflow flag.
  - If oflag = 1, a 32-bit overflow has occurred
  - If oflag = 0, a 32-bit overflow has not occurred

**Description**

Computes the first nr points of the positive side of the autocorrelation of the real vector x and stores the results in real output vector r. The full-length autocorrelation of vector x will have \(2 \times nx-1\) points with even symmetry around the lag 0 point (r[0]). This routine provides only the positive half of this for memory and computational savings.

**Algorithm**

**Raw Autocorrelation**

\[
r[j] = \sum_{k=0}^{nx-1-j} x[j + k] x[k] \quad 0 \leq j \leq nr
\]

**Biased Autocorrelation**

\[
r[j] = \frac{1}{nx} \sum_{k=0}^{nx-1-j} x[j + k] x[k] \quad 0 \leq j \leq nr
\]
Unbiased Autocorrelation

\[ r[j] = \frac{1}{(nx - \text{abs}(j))} \sum_{k=0}^{nx-1-j} x[j+k] x[k] \] \hspace{1cm} 0 \leq j \leq nr

Overflow Handling Methodology
No scaling implemented for overflow prevention

Special Requirements
None

Implementation Notes

- Special debugging consideration: This function is implemented as a macro that invokes different autocorrelation routines according to the type selected. As a consequence the acorr symbol is not defined. Instead the acorr_raw, acorr_bias, acorr_unbias symbols are defined.

- Autocorrelation is implemented using time-domain techniques

Example
See examples/abias, examples/aubias, examples/araw subdirectories

Benchmarks (preliminary)

<table>
<thead>
<tr>
<th></th>
<th>Abias:</th>
<th>Araw:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cycles†</td>
<td></td>
</tr>
<tr>
<td>Abias:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nr even:</td>
<td>((4 \times nx - nr \times (nr + 2) + 20) / 8 \times nr)</td>
<td>((4 \times nx - nr \times (nr + 2) + 28) / 8 \times nr)</td>
</tr>
<tr>
<td>nr odd:</td>
<td>((4 \times nx - (nr - 1) \times (nr + 1) + 20) / 8 \times (nr - 1) + 10)</td>
<td>((4 \times nx - (nr - 1) \times (nr + 1) + 28) / 8 \times (nr - 1) + 13)</td>
</tr>
<tr>
<td>nr = 1:</td>
<td>((nx + 2))</td>
<td>((nx + 1))</td>
</tr>
<tr>
<td>Overhead:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>nr even:</td>
<td>90</td>
<td>34</td>
</tr>
<tr>
<td>nr odd:</td>
<td>83</td>
<td>35</td>
</tr>
<tr>
<td>nr = 1:</td>
<td>59</td>
<td>30</td>
</tr>
</tbody>
</table>

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
### Cycles†

<table>
<thead>
<tr>
<th></th>
<th><strong>Aubias:</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Core:</td>
<td></td>
</tr>
</tbody>
</table>
| nreven: | \[
\frac{(8 \cdot \text{n}_x - 3 \cdot \text{n}_r \cdot (\text{n}_r + 2) + 68)}{8} \cdot \text{n}_r
\] |
| nr odd: | \[
\frac{(8 \cdot \text{n}_x - 3 \cdot (\text{n}_r-1) \cdot (\text{n}_r+1) + 68)/8} \cdot (\text{n}_r - 1) + 33
\] |
| nr = 1: | \(\text{n}_x + 26\) |
| Overhead: |             |
| nr even: | 64         |
| nr odd:  | 55         |
| nr = 1:  | 47         |

<table>
<thead>
<tr>
<th>Code size (in bytes)</th>
<th><strong>Abias:</strong> 251</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Araw:</strong> 178</td>
<td></td>
</tr>
<tr>
<td><strong>Aubias:</strong> 308</td>
<td></td>
</tr>
</tbody>
</table>

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
**add**  

**Vector Add**

```c
ushort oflag = add (DATA *x, DATA *y, DATA *r, ushort nx, ushort scale)
```

(defined in add.asm)

**Arguments**

- **x[nx]** Pointer to input data vector 1 of size nx. In-place processing allowed (r can be = x = y)
- **y[nx]** Pointer to input data vector 2 of size nx
- **r[nx]** Pointer to output data vector of size nx containing
  - (x+y) if scale = 0
  - (x+y) / 2 if scale = 1
- **nx** Number of elements of input and output vectors. nx ≥ 4
- **scale** Scale selection
  - If scale = 1, divide the result by 2 to prevent overflow
  - If scale = 0, do not divide by 2
- **oflag** Overflow flag.
  - If oflag = 1, a 32-bit overflow has occurred
  - If oflag = 0, a 32-bit overflow has not occurred

**Description**

This function adds two vectors, element by element.

**Algorithm**

```
for (i = 0; i < nx; i + +)  z(i) = x(i) + y(i)
```

**Overflow Handling Methodology** Scaling implemented for overflow prevention (User selectable)

**Special Requirements** None

**Implementation Notes** None

**Example** See examples/add subdirectory

**Benchmarks (preliminary)**

- Cycles† Core: 3 * nx
  - Overhead: 23
- Code size 60

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
Arctangent 2 Implementation

atan2_16

ushort oflag = atan2_16 (DATA *i, DATA *q, DATA *r, ushort nx)
(defined in arct2.asm)

Arguments:

q[nx] Pointer to quadrature input vector of size nx.
i[nx] Pointer to in-phase input vector of size nx
r[nx] Pointer to output data vector (in Q15 format) number representation of size nx containing. In-place processing allowed (r can be equal to x) on output, r contains the arctangent of (q/I) /PI

nx Number of elements of input and output vectors.
oflag Overflow flag.
  □ If oflag = 1, a 32-bit overflow has occurred
  □ If oflag = 0, a 32-bit overflow has not occurred

Description
This function calculates the arctangent of the ratio q/I, where –1 <= atan2_16(Q/I) <= 1 representing an actual range of –PI < atan2_16(Q/I) < PI. The result is placed in the resultant vector r. Output scale factor correction = PI. For example, if:
y = [0x1999, 0x1999, 0x0, 0xe667, 0x1999] (equivalent to [0.2, 0.2, 0, –0.2, 0.2] float)
x = [0x1999, 0x3dcc, 0x7 ffff, 0x3dcc c234] (equivalent to [0.2, 0.4828, 1, 0.4828, –0.4828] float)
atan2_16(y, x, r,4) should give:
r = [0x2000, 0x1000, 0x0, 0x0000, 0x7000] equivalent to [0.25, 0.125, 0, –0.125, 0.875]*pi

Algorithm
for (j = 0; j < nx; j + +)  r(j) = atan2(q(j), i(j))

Overflow Handling Methodology Not applicable

Special Requirements Linker command file: you must allocate .data section (for polynomial coefficients)

Implementation Notes None

Example See examples/arct2 subdirectory
Benchmarks (preliminary)

Cycles† \[18 + 62 \times nx\]

Code size

170 program; 10 data; 4 stack
(in bytes)

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
atan16

Arctangent Implementation

ushort oflag = atan16 (DATA *x, DATA *r, ushort nx)  
(defined in atant.asm)

Arguments

x[nx]  Pointer to input data vector of size nx. x contains the tangent of r, where |x| < 1.

r[nx]  Pointer to output data vector of size nx containing the arctangent of x in the range [-pi/4, pi/4] radians. In-place processing allowed (r can be equal to x)  
atan(1.0) = 0.7854 or 6478h

nx  Number of elements of input and output vectors.

oflag  Overflow flag.

- If oflag = 1, a 32-bit overflow has occurred
- If oflag = 0, a 32-bit overflow has not occurred

Description

This function calculates the arc tangent of each of the elements of vector x. The result is placed in the resultant vector r and is in the range [-pi/2 to pi/2] radians. For example,
if x = [0x7fff, 0x3505, 0x1976, 0x0] (equivalent to tan(PI/4), tan(PI/8), tan(PI/16), 0 in float):
atan16(x,r,4) should give
r = [0x6478, 0x3243, 0x1921, 0x0] equivalent to [PI/4, PI/8, PI/16, 0]

Algorithm

for (i = 0; i < nx; i++)  r(i) = atan(x(i))

Overflow Handling Methodology  Not applicable

Special Requirements  Linker command file: you must allocate .data section (for polynomial coefficients)

Implementation Notes

- atan(x), with 0 <= x <= 1, output scaling factor = PI.
- Uses a polynomial to compute the arctan (x) for |x| <1. For |x| > 1, you can express the number x as a ratio of 2 fractional numbers and use the atan2_16 function.

Example

See examples/atant subdirectory
Benchmarks (preliminary)

Cycles†  $14 + 8 \times nx$

Code size (in bytes)  43 program; 6 data

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
**bexp**  

*Block Exponent Implementation*

short \( r = \text{bexp} \left( \text{DATA} \times x, \text{ushort} \ nx \right) \)

(defined in bexp.asm)

**Arguments**

- \( x \ [nx] \): Pointer to input vector of size \( nx \)
- \( r \): Return value. Maximum exponent that may be used in scaling.
- \( nx \): Length of input data vector

**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**

Not applicable

**Overflow Handling Methodology**

Not applicable

**Special Requirements**

None

**Implementation Notes**

None

**Example**

See examples/bexp subdirectory

**Benchmarks (preliminary)***

<table>
<thead>
<tr>
<th></th>
<th>Core: 3 * nx</th>
<th>Overhead: 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Code size (in bytes)</td>
<td>19</td>
<td></td>
</tr>
</tbody>
</table>
cbrev

Complex Bit Reverse

```c
void cbrev (DATA *a, DATA *r, ushort n)
(defined in cbrev.asm)
```

**Arguments**

- `x[2*nx]` Pointer to complex input vector `x`
- `r[2*nx]` Pointer to complex output vector `r`
- `nx` Number of complex elements of vectors `x` and `r`.
  - To bit-reverse the input of a complex FFT, `nx` should be the complex FFT size.
  - To bit-reverse the input of a real FFT, `nx` should be half the real FFT size.

**Description**

This function bit-reverses the position of elements in complex vector `x` into output vector `r`. In-place bit-reversing is allowed. Use this function in conjunction with FFT routines to provide the correct format for the FFT input or output data. If you bit-reverse a linear-order array, you obtain a bit-reversed order array. If you bit-reverse a bit-reversed order array, you obtain a linear-order array.

**Algorithm**

Not applicable

**Overflow Handling Methodology**

Not applicable

**Special Requirements**

None

**Implementation Notes**

- `x` is read with bit-reversed addressing and `r` is written in normal linear addressing.
- Off-place bit-reversing (`x = r`) requires half as many cycles as in-place bit-reversing (`x <> r`). However, this is at the expense of doubling the data memory requirements.

**Example**

See examples/cfft and examples/rfft subdirectories

**Benchmarks (preliminary)**

- **Cycles**†
  - Core:
    - `2 * nx` (off-place)
    - `4 * nx + 6` (in-place)
  - Overhead: 17
- **Code size**
  - `81` (includes support for both in-place and off-place bit-reverse)

† Assesses all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
cfft  

Forward Complex FFT

void cfft (DATA *x, ushort nx, ushort scale);

(defined in cfft.asm)

Arguments

x [2*nx]  Pointer to input vector containing nx complex elements
          (2*nx real elements) in bit-reversed order. On output, vector
          a contains the nx complex elements of the FFT(x). Complex
          numbers are stored in interleaved Re-Im format.

nx  Number of complex elements in vector x. Must be between
     8 and 1024.

scale  Flag to indicate whether or not scaling should be
       implemented during computation.
       if (scale == 1)
           scale factor = nx;
       else
           scale factor = 1;
       end

Description

Computes a complex nx-point FFT on vector x, which is in bit-reversed order.
The original content of vector x is destroyed in the process. The nx complex
elements of the result are stored in vector x in normal-order.

Algorithm

\[ y[k] = \frac{1}{(scale \ factor)} \sum_{i=0}^{nx-1} x[i] \cdot \left( \cos \left( \frac{2 \cdot \pi \cdot i \cdot k}{nx} \right) + j \sin \left( \frac{2 \cdot \pi \cdot i \cdot k}{nx} \right) \right) \]

Overflow Handling Methodology

If scale==1, scaling before each stage is implemented for overflow prevention

Special Requirements

- Special linker command file sections required.
- This function requires the inclusion of file twiddle1024br.h that contains
  the twiddle table (automatically included).
cfft  *Forward Complex FFT*

**Implementation Notes**

- Radix-2 DIF version of the FFT algorithm is implemented. The implementation is optimized for MIPS, not for code size. The first stage is unrolled and the last two stages are implemented in radix-4, for MIPS optimization.

- This routine prevents overflow by scaling by 2 before each FFT stage, assuming that the parameter scale is set to 1.

**Example**

See examples/cfft subdirectory

**Benchmarks (preliminary)**

- 5 cycles (radix-2 butterfly)
- 10 cycles (radix-4 butterfly)
- 1 cycle/complex value (scaling)

Cycles†  FFT Core:
5 * (nx/2) + 5 * (nx/2) * (log2(nx) – 3) * 1.15 + 10 * (nx/4)
Scaling‡:  2 * nx * log2(nx)
Overhead:  71

**Code size** (in bytes)  510

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
‡ When scaling is used, this version of FFT has a separate scaling stage for each FFT stage. The next release of this code will embed scaling as part of the butterfly core without adding extra cycle penalty.
Complex FIR Filter

\texttt{cfir}

\begin{verbatim}
ushort oflag = cfir (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh)
\end{verbatim}

**Arguments**

- **x[2*nx]** Pointer to input vector of nx complex elements.
- **h[2*nh]** Pointer to complex coefficient vector of size nh in normal order. For example, if nh=6, then \( h[nh] = \{h0r, h0i, h1r, h1i, h2r, h2i, h3r, h3i, h4r, h4i, h5r, h5i\} \) where h0 resides at the lowest memory address in the array.
- **r[2*nx]** Pointer to output vector of nx complex elements. In-place computation \((r = x)\) is allowed.
- **dbuffer[2*nh + 2]** Pointer to delay buffer of length \( nh = 2 * nh + 2 \)
  - In the case of multiple-buffering schemes, this array should be initialized to 0 for the first filter block only. Between consecutive blocks, the delay buffer preserves the previous r output elements needed.
  - The first element in this array is present for alignment purposes, the second element is special in that it contains the array index–1 of the oldest input entry in the delay buffer. This is needed for multiple-buffering schemes, and should be initialized to 0 (like all the other array entries) for the first block only.
- **nx** Number of complex input samples
- **nh** The number of complex coefficients of the filter. For example, if the filter coefficients are \( \{h0, h1, h2, h3, h4, h5\} \), then \( nh = 6 \). Must be a minimum value of 3. For smaller filters, zero pad the coefficients to meet the minimum value.
- **oflag** Overflow error flag (returned value)
  - If oflag = 1, a 32-bit data overflow has occurred in an intermediate or final result.
  - If oflag = 0, a 32-bit overflow has not occurred.
Description
Computes a complex FIR filter (direct-form) using the coefficients stored in vector h. The complex input data is stored in vector x. The filter output result is stored in vector r. This function maintains the array dbuffer containing the previous delayed input values to allow consecutive processing of input data blocks. This function can be used for both block-by-block \((nx \geq 2)\) and sample-by-sample filtering \((nx = 1)\). In-place computation \((r = x)\) is allowed.

Algorithm
\[
r[j] = \sum_{k=0}^{nh-1} h[k] \times [j - k] \quad \text{where} \quad 0 \leq j \leq nx
\]

Overflow Handling Methodology
No scaling implemented for overflow prevention.

Special Requirements
nh must be a minimum value of 3. For smaller filters, zero pad the h[] array.

Implementation Notes
The first element in the dbuffer array is present only for alignment purposes. The second element in this array \((index=0)\) is the entry index for the input history. It is treated as an unsigned 16-bit value by the function even though it has been declared as signed in C. The value of the entry index is equal to the index – 1 of the oldest input entry in the array. The remaining elements make up the input history. Figure 4–1 shows the array in memory with an entry index of 2. The newest entry in the dbuffer is denoted by x[\(j-0\)], which in this case would occupy index = 3 in the array. The next newest entry is x[\(j-1\)], and so on. It is assumed that all x() entries were placed into the array by the previous invocation of the function in a multiple-buffering scheme.

Figure 4–1, Figure 4–2, and Figure 4–3 show the dbuffer, x, and r arrays as they appear in memory.
Figure 4–1. dbuffer Array in Memory at Time j

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>dummy value</td>
<td></td>
<td>lowest memory address</td>
</tr>
<tr>
<td>entry index = 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(x_r(j-nh-2))</td>
<td>(x_i(j-nh-2))</td>
<td></td>
</tr>
<tr>
<td>(x_r(j-nh-1))</td>
<td>(x_i(j-nh-1))</td>
<td></td>
</tr>
<tr>
<td>(x_r(j-nh))</td>
<td>(x_i(j-nh))</td>
<td></td>
</tr>
<tr>
<td>(x_r(j-0))</td>
<td>(x_i(j-0))</td>
<td></td>
</tr>
<tr>
<td>(x_r(j-1))</td>
<td>(x_i(j-1))</td>
<td></td>
</tr>
<tr>
<td>(x_r(j-2))</td>
<td>(x_i(j-2))</td>
<td></td>
</tr>
<tr>
<td>(x_r(j-nh-3))</td>
<td>(x_i(j-nh-3))</td>
<td></td>
</tr>
<tr>
<td>(x_r(j-nh-4))</td>
<td>(x_i(j-nh-4))</td>
<td></td>
</tr>
<tr>
<td>(x_r(j-nh-3))</td>
<td>(x_i(j-nh-3))</td>
<td>highest memory address</td>
</tr>
</tbody>
</table>

oldest \(x()\) entry

newest \(x()\) entry
Figure 4–2. x Array in Memory

- Oldest x( ) entry: xr(0), highest memory address: xi(1), lowest memory address: xr(nx–1)
- Newest x( ) entry: xr(nx–2), highest memory address: xi(nx–1)

Figure 4–3. r Array in Memory

- Oldest r( ) entry: rr(0), highest memory address: ri(1), lowest memory address: rr(nx–1)
- Newest r( ) entry: rr(nx–2), highest memory address: nr(nx–1)

Example

See examples/cfir subdirectory

Benchmarks (preliminary)

<table>
<thead>
<tr>
<th>Cycles†</th>
<th>Core: nx * (2 + 2 * nh)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead:</td>
<td>51</td>
</tr>
<tr>
<td>Code size (in bytes)</td>
<td>136</td>
</tr>
</tbody>
</table>

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
void cifft (DATA *x, ushort nx, ushort scale);

(defined in cifft.asm)

Arguments

x [2*nx] Pointer to input vector containing nx complex elements (2*nx real elements) in bit-reversed order. On output, vector a contains the nx complex elements of the IFFT(x). Complex numbers are stored in interleaved Re-Im format.

nx Number of complex elements in vector x. Must be between 8 and 1024.

scale Flag to indicate whether or not scaling should be implemented during computation.

if (scale == 1)
    scale factor = nx;
else
    scale factor = 1;
end

Description

Computes a complex nx-point IFFT on vector x, which is in bit-reversed order. The original content of vector x is destroyed in the process. The nx complex elements of the result are stored in vector x in normal-order.

Algorithm

(IDFT)

\[ y[k] = \frac{1}{(scale \text{ factor})} \sum_{i=0}^{nx-1} x[w] \begin{pmatrix} \cos \left( \frac{2 \pi i^* k}{nx} \right) + j \sin \left( \frac{2 \pi i^* k}{nx} \right) \end{pmatrix} \]

Overflow Handling Methodology

If scale==1, scaling before each stage is implemented for overflow prevention

Special Requirements

- Special linker command file sections required.
- This function requires the inclusion of file twiddle1024br.h that contains the twiddle table (automatically included).
Inverse Complex FFT

Implementation Notes

- Radix-2 DIF version of the IFFT algorithm is implemented. The implementation is optimized for MIPS, not for code size. The first stage is unrolled and the last two stages are implemented in radix-4, for MIPS optimization.

- This routine prevents overflow by scaling by 2 before each IFFT stage, assuming that the parameter scale is set to 1.

- IFFT implementation is directly derived from FFT implementation by changing signs (or replacing adds with subtractions or vice versa) where appropriate.

Example

See examples/cifft subdirectory

Benchmarks (preliminary)

- 5 cycles (radix-2 butterfly)
- 10 cycles (radix-4 butterfly)
- 1 cycle/complex value (scaling)

Cycles†     IFFT Core(approx):
            5 * (nx/2) + 5 * (nx/2) * (log2(nx) – 3) * 1.15 + 10 * (nx/4)
            Scaling:    2*nx*log2(nx)
            Overhead:  71

Code size   510
            (in bytes)

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
**Convolution**

```c
ushort oflag = convol (DATA *x, DATA *h, DATA *r, ushort nr, ushort nh)
```

**Arguments**

- `x[nr+nh–1]` Pointer to input vector of `nr + nh – 1` real elements.
- `h[nh]` Pointer to input vector of `nh` real elements.
- `r[nr]` Pointer to output vector of `nr` real elements.
- `nr` Number of elements in vector `r`. In-place computation (`r = x`) is allowed (see Description section for comment).
- `nh` Number of elements in vector `h`.
- `oflag` Overflow error flag (returned value)
  - If `oflag = 1`, a 32-bit data overflow occurred in an intermediate or final result.
  - If `oflag = 0`, a 32-bit overflow has not occurred.

**Description**

Computes the real convolution of two real vectors `x` and `h`, and places the results in vector `r`. Typically used for block FIR filter computation when there is no need to retain an input delay buffer. This function can also be used to implement single-sample FIR filters (`nr = 1`) provided the input delay history for the filter is maintained external to this function. In-place computation (`r = x`) is allowed, but be aware that the `r` output vector is shorter in length than the `x` input vector; therefore, `r` will only overwrite the first `nr` elements of the `x`.

**Algorithm**

```latex
r[j] = \sum_{k=0}^{nh-1} h[k] x[j - k] \text{ where } 0 \leq j \leq nr
```

**Overflow Handling Methodology**
No scaling implemented for overflow prevention.

**Special Requirements** none

**Implementation Notes**
Figure 4–4, Figure 4–5, and Figure 4–6 show the `x`, `r`, and `h` arrays as they appear in memory.
Convolution

Figure 4–4. x Array in Memory

\[
\begin{array}{c}
  x(0) \\
  x(1) \\
  \vdots \\
  x(nr+nh-2) \\
  x(nr+nh-1)
\end{array}
\]

lowest memory address

Figure 4–5. r Array in Memory

\[
\begin{array}{c}
  r(0) \\
  r(1) \\
  \vdots \\
  r(nr-2) \\
  r(nr-1)
\end{array}
\]

lowest memory address

Figure 4–6. h Array in Memory

\[
\begin{array}{c}
  h(0) \\
  h(1) \\
  \vdots \\
  h(nh-2) \\
  h(nh-1)
\end{array}
\]

lowest memory address

Example
See examples/convol subdirectory

Benchmarks (preliminary)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles†</td>
<td>Core: ( nr \times (1 + nh) )</td>
</tr>
<tr>
<td>Overhead:</td>
<td>21</td>
</tr>
<tr>
<td>Code size (in bytes)</td>
<td>88</td>
</tr>
</tbody>
</table>

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
**Convolution (fast)**

*convol1*

`ushort oflag = convol1 (DATA *x, DATA *h, DATA *r, ushort nr, ushort nh)`

**Arguments**

- `x[nr+nh–1]` Pointer to input vector of `nr+nh–1` real elements.
- `h[nh]` Pointer to input vector of `nh` real elements.
- `r[nr]` Pointer to output vector of `nr` real elements. In-place computation (`r = x`) is allowed (see Description section for comment).
- `nr` Number of elements in vector `r`. Must be an even number.
- `nh` Number of elements in vector `h`.
- `oflag` Overflow error flag (returned value)
  - If `oflag = 1`, a 32-bit data overflow occurred in an intermediate or final result.
  - If `oflag = 0`, a 32-bit overflow has not occurred.

**Description**

Computes the real convolution of two real vectors `x` and `h`, and places the results in vector `r`. This function utilizes the dual-MAC capability of the C55x to process in parallel two output samples for each iteration of the inner function loop. It is, therefore, roughly twice as fast as CONVOL, which implements only a single-MAC approach. However, the number of output samples (`nr`) must be even. Typically used for block FIR filter computation when there is no need to retain an input delay buffer. This function can also be used to implement single-sample FIR filters (`nr = 1`) provided the input delay history for the filter is maintained external to this function. In-place computation (`r = x`) is allowed, but be aware that the `r` output vector is shorter in length than the `x` input vector; therefore, `r` will only overwrite the first `nr` elements of the `x`.

**Algorithm**

\[ r[j] = \sum_{k=0}^{nh-1} h[k] x[j - k] \text{ where } 0 \leq j \leq nr \]

**Overflow Handling Methodology**

No scaling implemented for overflow prevention.

**Special Requirements**

- `nr` must be an even value.
- The vector `h[nh]` must be located in internal memory since it is accessed using the C55x coefficient bus, and that bus does not have access to external memory.
**Implementation Notes** Figure 4–7, Figure 4–8, and Figure 4–9 show the x, r, and h arrays as they appear in memory.

**Figure 4–7. x Array in Memory**

```
x(0)
x(1)
```

lowest memory address

```
x(nr+nh–2)
x(nr+nh–1)
```

highest memory address

**Figure 4–8. r Array in Memory**

```
r(0)
r(1)
```

lowest memory address

```
r(nr–2)
r(nr–1)
```

highest memory address

**Figure 4–9. h Array in Memory**

```
h(0)
h(1)
```

lowest memory address

```
h(nh–2)
h(nh–1)
```

highest memory address

**Example** See examples/convol1 subdirectory

**Benchmarks (preliminary)**

<table>
<thead>
<tr>
<th>Cycles†</th>
<th>Core: nr/2 * (2 + nh)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead: 24</td>
<td></td>
</tr>
</tbody>
</table>

| Code size (in bytes) | 101 |

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
**Arguments**

- `x[nr+nh-1]` Pointer to input vector of `nr + nh – 1` real elements.
- `h[nh]` Pointer to input vector of `nh` real elements.
- `r[nr]` Pointer to output vector of `nr` real elements. In-place computation (`r = x`) is allowed (see Description section for comment). This array must be aligned on a 32-bit boundary in memory.
- `nr` Number of elements in vector `r`. Must be an even number.
- `nh` Number of elements in vector `h`.
- `oflag` Overflow error flag (returned value)
  - If `oflag = 1`, a 32-bit data overflow has occurred in an intermediate or final result.
  - If `oflag = 0`, a 32-bit overflow has not occurred.

**Description**

Computes the real convolution of two real vectors `x` and `h`, and places the results in vector `r`. This function utilizes the dual-MAC capability of the C55x to process in parallel two output samples for each iteration of the inner function loop. It is, therefore, roughly twice as fast as CONVOL, which implements only a single-MAC approach. However, the number of output samples (`nr`) must be even. In addition, this function offers a small performance improvement over CONVOL1 at the expense of requiring the `r` array to be 32-bit aligned in memory. Typically used for block FIR filter computation when there is no need to retain an input delay buffer. This function can also be used to implement single-sample FIR filters (`nr = 1`) provided the input delay history for the filter is maintained external to this function. In-place computation (`r = x`) is allowed, but be aware that the `r` output vector is shorter in length than the `x` input vector; therefore, `r` will only overwrite the first `nr` elements of the `x`.

**Algorithm**

\[ r[j] = \sum_{k=0}^{nh-1} h[k] \times (j - k) \text{ where } 0 \leq j \leq nr \]

**Overflow Handling Methodology**

No scaling implemented for overflow prevention.
Special Requirements

- nr must be an even value.
- The vector $h[nh]$ must be located in internal memory since it is accessed using the C55x coefficient bus, and that bus does not have access to external memory.
- The vector $r[nr]$ must be 32-bit aligned in memory.

Implementation Notes  Figure 4–10, Figure 4–11, and Figure 4–12 show the $x$, $r$, and $h$ arrays as they appear in memory.

**Figure 4–10. $x$ Array in Memory**

```
<table>
<thead>
<tr>
<th>x(0)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x(1)</td>
<td></td>
</tr>
</tbody>
</table>
|      | ...
|      | ...
| x(nr+nh–2) |  |
| x(nr+nh–1) |  |
```

lowest memory address

highest memory address

**Figure 4–11. $r$ Array in Memory**

```
<table>
<thead>
<tr>
<th>r(0)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>r(1)</td>
<td></td>
</tr>
</tbody>
</table>
|      | ...
|      | ...
| r(nr–2) |  |
| r(nr–1) |  |
```

lowest memory address

highest memory address

**Figure 4–12. $h$ Array in Memory**

```
<table>
<thead>
<tr>
<th>h(0)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>h(1)</td>
<td></td>
</tr>
</tbody>
</table>
|      | ...
|      | ...
| h(nh–2) |  |
| h(nh–1) |  |
```

lowest memory address

highest memory address
Example

See examples/convol2 subdirectory

Benchmarks (preliminary)

<table>
<thead>
<tr>
<th>Cycles†</th>
<th>Core: nr/2 * (1 + nh)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead:</td>
<td>24</td>
</tr>
</tbody>
</table>

Code size 100
(in bytes)

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
Correlation, full-length

ushort oflag = corr (DATA *x, DATA *y, DATA *r, ushort nx, ushort ny, type)

Arguments

x [nx] Pointer to real input vector of nx real elements.

x [ny] Pointer to real input vector of ny real elements.

r[nx+ny–1] Pointer to real output vector containing the full-length correlation (nx + ny – 1 elements) of vector x with y. r must be different than both x and y (in-place computation is not allowed).

nx Number of real elements in vector x

ny Number of real elements in vector y

type Correlation type selector. Types supported:

[] If type = raw, r contains the raw correlation

[] If type = bias, r contains the biased-correlation

[] If type = unbias, r contains the unbiased-correlation

oflag Overflow flag

[] If oflag = 1, a 32-bit overflow has occurred

[] If oflag = 0, a 32-bit overflow has not occurred

Description

Computes the full-length correlation of vectors x and y and stores the result in vector r. using time-domain techniques.

Algorithm

Raw correlation

\[ r[j] = \sum_{k=0}^{nr-j-1} x[j + k] * y[k] \quad 0 \leq j \leq nr = nx + ny - 1 \]

Biased correlation

\[ r[j] = 1/nr \sum_{k=0}^{nr-j-1} x[j + k] * y[k] \quad 0 \leq j \leq nr = nx + ny - 1 \]

Unbiased correlation

\[ r[j] = 1/(nx - abs(j)) \sum_{k=0}^{nr-j-1} x[j + k] * y[k] \quad 0 \leq j \leq nr = nx + ny - 1 \]
Overflow Handling Methodology  No scaling implemented for overflow prevention

Special Requirements  None

Implementation Notes

- Special debugging consideration: This function is implemented as a macro that invokes different correlation routines according to the type selected. As a consequence the corr symbol is not defined. Instead the corr_raw, corr_bias, corr_unbias symbols are defined.
- Correlation is implemented using time-domain techniques

Benchmarks (preliminary)

<table>
<thead>
<tr>
<th></th>
<th>Raw:</th>
<th>Unbias:</th>
<th>Bias:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles</td>
<td>2 times faster than C54x</td>
<td>2.14 times faster than C54x</td>
<td>2.1 times faster than C54x</td>
</tr>
</tbody>
</table>
Adaptive Delayed LMS Filter

```c
ushort oflag = dlms (DATA *x, DATA *h, DATA *r, DATA *des, DATA *dbuffer,
DATA step, ushort nh, ushort nx)
```

(defined in dlms.asm)

**Arguments**

- **x[nx]** Pointer to input vector of size nx
- **h[nh]** Pointer to filter coefficient vector of size nh.
  - h is stored in reversed order: h(n–1), ... h(0) where h[n] is at the lowest memory address.
  - Memory alignment: h is a circular buffer and must start in a k-bit boundary (that is, the k LSBs of the starting address must be zeros) where k = log2(nh)
- **r[nx]** Pointer to output data vector of size nx. r can be equal to x.
- **des[nx]** Pointer to expected output array
- **dbuffer[nh+2]** Pointer to the delay buffer structure.
  - The delay buffer is a structure comprised of an index register and a circular buffer of length nh + 1. The index register is the index into the circular buffer of the oldest data sample.
- **nh** Number of filter coefficients. Filter order = nh – 1.
  - nh ≥ 3
- **nx** Length of input and output data vectors
- **oflag** Overflow flag.
  - If oflag = 1, a 32-bit overflow has occurred
  - If oflag = 0, a 32-bit overflow has not occurred

**Description**

Adaptive delayed least-mean-square (LMS) FIR filter using coefficients stored in vector h. Coefficients are updated after each sample based on the LMS algorithm and using a constant step = 2*mu. The real data input is stored in vector dbuffer. The filter output result is stored in vector r.

LMS algorithm uses the previous error and the previous sample (delayed) to take advantage of the C55x LMS instruction.
The delay buffer used is the same delay buffer used for other functions in the C55x DSP Library. There is one more data location in the circular delay buffer than there are coefficients. Other C55x DSP Library functions use this delay buffer to accommodate use of the dual-MAC architecture. In the DLMS function, we make use of the additional delay slot to allow coefficient updating as well as FIR calculation without a need to update the circular buffer in the interim operations.

The FIR output calculation is based on \(x(i)\) through \(x(i-\text{nh}+1)\). The coefficient update for a delayed LMS is based on \(x(i-1)\) through \(x(i-\text{nh})\). Therefore, by having a delay buffer of \(\text{nh}+1\), we can perform all calculations with the given delay buffer containing delay values of \(x(i)\) through \(x(i-\text{nh})\). If the delay buffer was of length \(\text{nh}\), the oldest data sample, \(x(i-\text{nh})\), would need to be updated with the newest data sample, \(x(i)\), sometime after the calculation of the first coefficient update term, but before the calculation of the last FIR term.

**Algorithm**

\[
\text{FIR portion}\\ r[j] = \sum_{k=0}^{\text{nh}-1} h[k] \cdot x[i-k] \quad 0 \leq i \leq nx - 1
\]

Adaptation using the previous error and the previous sample:

\[
e(i) = \text{des}(i-1) - r(i-1)\\ h_n(i + 1) = h_n(i) + 2 \cdot \mu \cdot e(i-1) \cdot x(i-k-1)
\]

**Overflow Handling Methodology** No scaling implemented for overflow prevention.

**Special Requirements** Minimum of 2 input and desired data samples. Minimum of 2 coefficients

**Implementation Notes**

- Delayed version implemented to take advantage of the C55x LMS instruction.
- Effect of using delayed error signal on convergence minimum:
  For reference, following is the algorithm for the regular LMS (non-delayed):

\[
\text{FIR portion}\\ r[j] = \sum_{k=0}^{\text{nh}-1} h[k] \cdot x[i-k] \quad 0 \leq i \leq nx - 1
\]

Adaptation using the current error and the current sample:

\[
e(i) = \text{des}(i) - r(i)\\ h_n(i + 1) = h_n(i) + 2 \cdot \mu \cdot e(i) \cdot x(i-k)
\]
Example  
See examples/dlms subdirectory

Benchmarks (preliminary)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles†</td>
<td>Core: nx * (7 + 2*(nh – 1)) = nx * (5 + 2 * nh)</td>
</tr>
<tr>
<td></td>
<td>Overhead: 26</td>
</tr>
<tr>
<td>Code size</td>
<td>122</td>
</tr>
<tr>
<td>(in bytes)</td>
<td></td>
</tr>
</tbody>
</table>

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
**Exponential Base e**

**expn**

```c
ushort oflag = expn (DATA *x, DATA *r, ushort nx)
```

(defined in expn.asm)

**Arguments**

- **x[nx]**: Pointer to input vector of size nx. x contains the numbers normalized between (-1,1) in q15 format.
- **r[nx]**: Pointer to output data vector (Q3.12 format) of size nx. r can be equal to x.
- **nx**: Length of input and output data vectors.
- **oflag**: Overflow flag.
  - If `oflag = 1`, a 32-bit overflow has occurred
  - If `oflag = 0`, a 32-bit overflow has not occurred

**Description**
Computes the exponent of elements of vector x using Taylor series.

**Algorithm**
```
for (i = 0; i < nx; i++)   y(i) = e^{x(i)} where -1 < x(i) < 1
```

**Overflow Handling Methodology** Not applicable

**Special Requirements** Linker command file: you must allocate .data section (for polynomial coefficients)

**Implementation Notes**
Computes the exponent of elements of vector x. It uses the following Taylor series:
```
exp(x) = c0 + (c1 * x) + (c2 * x^2) + (c3 * x^3) + (c4 * x^4) + (c5 * x^5)
```
where
- c0 = 1.0000
- c1 = 1.0001
- c2 = 0.4990
- c3 = 0.1705
- c4 = 0.0348
- c5 = 0.0139
**expn  Exponential Base e**

**Example**

See examples/expn subdirectory

**Benchmarks (preliminary)**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles†</td>
<td>Core: 11 * nx</td>
</tr>
<tr>
<td></td>
<td>Overhead: 18</td>
</tr>
<tr>
<td>Code size</td>
<td>57</td>
</tr>
<tr>
<td>(in bytes)</td>
<td></td>
</tr>
</tbody>
</table>

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
FIR Filter

```
ushort oflag = fir (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh)
```

Arguments

- **x[nx]** Pointer to input vector of nx real elements.
- **h[nh]** Pointer to coefficient vector of size nh in normal order. For example, if nh=6, then h[nh] = {h0, h1, h2, h3, h4, h5} where h0 resides at the lowest memory address in the array.
- **r[nx]** Pointer to output vector of nx real elements. In-place computation (r = x) is allowed.
- **dbuffer[nh+2]** Pointer to delay buffer of length nh = nh + 2
  - In the case of multiple-buffering schemes, this array should be initialized to 0 for the first filter block only. Between consecutive blocks, the delay buffer preserves the previous r output elements needed.
  - The first element in this array is special in that it contains the array index–1 of the oldest input entry in the delay buffer. This is needed for multiple-buffering schemes, and should be initialized to 0 (like all the other array entries) for the first block only.
- **nx** Number of input samples
- **nh** The number of coefficients of the filter. For example, if the filter coefficients are {h0, h1, h2, h3, h4, h5}, then nh = 6. Must be a minimum value of 3. For smaller filters, zero pad the coefficients to meet the minimum value.
- **oflag** Overflow error flag (returned value)
  - If oflag = 1, a 32-bit data overflow occurred in an intermediate or final result.
  - If oflag = 0, a 32-bit overflow has not occurred.
fir  FIR Filter

Description Computes a real FIR filter (direct-form) using the coefficients stored in vector h. The real input data is stored in vector x. The filter output result is stored in vector r. This function maintains the array dbuffer containing the previous delayed input values to allow consecutive processing of input data blocks. This function can be used for both block-by-block (nx ≥ 2) and sample-by-sample filtering (nx = 1). In place computation (r = x) is allowed.

Algorithm
\[ r[j] = \sum_{k=0}^{nh-1} h[k] x[j-k] \text{ where } 0 \leq j \leq nx \]

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements nh must be a minimum value of 3. For smaller filters, zero pad the h[] array.

Implementation Notes The first element in the dbuffer array (index = 0) is the entry index for the input history. It is treated as an unsigned 16-bit value by the function even though it has been declared as signed in C. The value of the entry index is equal to the index – 1 of the oldest input entry in the array. The remaining elements make up the input history. Figure 4–13 shows the array in memory with an entry index of 2. The newest entry in the dbuffer is denoted by x(j−0), which in this case would occupy index = 3 in the array. The next newest entry is x(j−1), and so on. It is assumed that all x() entries were placed into the array by the previous invocation of the function in a multiple-buffering scheme.

The dbuffer array actually contains one more history value than is needed to implement this filter. The value x(j−nh) does not enter into the calculations for the output r(j). However, this value is required in other DSPLIB filter functions that utilize the dual-MAC units on the C55x, such as FIR2. Including this extra location ensures compatibility across all filter functions in the C55x DSPLIB.

Figure 4–13, Figure 4–14, and Figure 4–15 show the dbuffer, x, and r arrays as they appear in memory.
**Figure 4–13. dbuffer Array in Memory at Time j**

- Entry index = 2
- x(j–nh–2)
- x(j–nh–1)
- x(j–nh–1)
- x(j–nh)
- x(j–0)
- x(j–1)
- x(j–2)

- Oldest x( ) entry
- Newest x( ) entry

- Highest memory address

**Figure 4–14. x Array in Memory**

- Oldest x( ) entry
- Newest x( ) entry

- Lowest memory address

**Figure 4–15. r Array in Memory**

- Oldest x( ) entry
- Newest x( ) entry

- Lowest memory address

*Function Descriptions* 4-41
Example

See examples/fir subdirectory

Benchmarks (preliminary)

Cycles†  Core:  nx * (2 + nh)
Overhead:  25

Code size  107
(in bytes)

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
Block FIR Filter (fast) \texttt{fir2}

\texttt{ushort oflag = fir2 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh)}

Arguments

- \texttt{x[nx]} Pointer to input vector of \texttt{nx} real elements.
- \texttt{r[nx]} Pointer to output vector of \texttt{nx} real elements. In-place computation \((r = x)\) is allowed.
- \texttt{h[nh]} Pointer to coefficient vector of size \texttt{nh} in normal order. For example, if \texttt{nh=6}, then \texttt{h[nh]} = \{h0, h1, h2, h3, h4, h5\} where \texttt{h0} resides at the lowest memory address in the array.
- \texttt{This array must be located in internal memory since it is accessed by the C55x coefficient bus.}
- \texttt{dbuf[er[nh + 2]]} Pointer to delay buffer of length \texttt{nh = nh + 2}
  - In the case of multiple-buffering schemes, this array should be initialized to 0 for the first filter block only. Between consecutive blocks, the delay buffer preserves the previous \texttt{r} output elements needed.
  - The first element in this array is special in that it contains the array index−1 of the oldest input entry in the delay buffer. This is needed for multiple-buffering schemes, and should be initialized to 0 (like all the other array entries) for the first block only.
- \texttt{nx} Number of input samples. Must be an even number.
- \texttt{nh} The number of coefficients of the filter. For example, if the filter coefficients are \{h0, h1, h2, h3, h4, h5\}, then \texttt{nh = 6}. Must be a minimum value of 3. For smaller filters, zero pad the coefficients to meet the minimum value.
- \texttt{oflag} Overflow error flag (returned value)
  - \texttt{If oflag = 1}, a 32-bit data overflow occurred in an intermediate or final result.
  - \texttt{If oflag = 0}, a 32-bit overflow has not occurred.
fir2  Block FIR Filter (fast)

Description
Computes a real block FIR filter (direct-form) using the coefficients stored in vector h. This function utilizes the dual-MAC capability of the C55x to process in parallel two output samples for each iteration of the inner function loop. It is, therefore, roughly twice as fast as FIR, which implements only a single-MAC approach. However, the number of input samples (nx) must be even. The real input data is stored in vector x. The filter output result is stored in vector r. This function maintains the array dbuffer containing the previous delayed input values to allow consecutive processing of input data blocks. This function can be used for block-by-block filtering only (nx ≥ 2). It cannot be used for sample-by-sample filtering (nx = 1). In-place computation (r = x) is allowed.

Algorithm

\[ r[j] = \sum_{k=0}^{nh-1} h[k] x[j-k] \text{ where } 0 \leq j \leq nx \]

Overflow Handling Methodology
No scaling implemented for overflow prevention.

Special Requirements
- nh must be a minimum value of 3. For smaller filters, zero pad the h[] array.
- nx must be an even value.
- Coefficient array h[nh] must be located in internal memory since it is accessed using the C55x coefficient bus, and that bus does not have access to external memory.

Implementation Notes
The first element in the dbuffer array (index = 0) is the entry index for the input history. It is treated as an unsigned 16-bit value by the function even though it has been declared as signed in C. The value of the entry index is equal to the index – 1 of the oldest input entry in the array. The remaining elements make up the input history. Figure 4–16 shows the array in memory with an entry index of 2. The newest entry in the dbuffer is denoted by x(j–0), which in this case would occupy index = 3 in the array. The next newest entry is x(j–1), and so on. It is assumed that all x() entries were placed into the array by the previous invocation of the function in a multiple-buffering scheme.

Figure 4–16, Figure 4–17, and Figure 4–18 show the dbuffer, x, and r arrays as they appear in memory.
Figure 4–16. dbuffer Array in Memory at Time j

- entry index = 2
- x(j–2*nh2–2)
- x(j–2*nh2–1)
- x(j–2*nh2)
- x(j–0)
- x(j–1)
- x(j–2)

Oldest x( ) entry
Newest x( ) entry

Figure 4–17. x Array in Memory

- x(0)
- x(1)

Oldest x( ) entry
Newest x( ) entry

Figure 4–18. r Array in Memory

- r(0)
- r(1)

Oldest x( ) entry
Newest x( ) entry
**fir2**  Block FIR Filter (fast)

**Example**
See examples/fir2 subdirectory

**Benchmarks (preliminary)**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles†</td>
<td>Core: (nx/2) * (4 + nh)</td>
</tr>
<tr>
<td></td>
<td>Overhead: 32</td>
</tr>
<tr>
<td>Code size</td>
<td>134 (in bytes)</td>
</tr>
</tbody>
</table>

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
Decimating FIR Filter

ushort oflag = firdec (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nh, ushort nx, ushort D)

(defined in decimate.asm)

Arguments

x [nx] Pointer to real input vector of nx real elements.
h[nh] Pointer to coefficient vector of size nh in normal order:
H = b0 b1 b2 b3 …
r[nx/D] Pointer to real input vector of nx/D real elements.
In-place computation (r = x) is allowed
dbuffer[nh+1] Delay buffer

- In the case of multiple-buffering schemes, this array should be initialized to 0 for the first block only. Between consecutive blocks, the delay buffer preserves previous delayed input samples. It also preserves a ptr to the next new entry into the dbuffer. This ptr is preserved across function calls in dbuffer[0].
- Memory alignment: this is a circular buffer and must start in a k-bit boundary (that is, the k LSBs of the starting address must be zeros) where k = log2 (nh).

nx Number of real elements in vector x
nh Number of coefficients
D Decimation factor. For example a D = 2 means you drop every other sample. Ideally, nx should be a multiple of D. If not, the trailing samples will be lost in the process.
oflag Overflow error flag

- If oflag = 1, a 32-bit data overflow occurred in an intermediate or final result.
- If oflag = 0, a 32-bit overflow has not occurred.

Description

Computes a decimating real FIR filter (direct-form) using coefficient stored in vector h. The real data input is stored in vector x. The filter output result is stored in vector r. This function retains the address of the delay filter memory d containing the previous delayed values to allow consecutive processing of blocks. This function can be used for both block-by-block and sample-by-sample filtering (nx = 1).
**firdec  Decimating FIR Filter**

**Algorithm**

\[ r[j] = \sum_{k=0}^{N} h[k]x[j * D - k] \quad 0 \leq j \leq nx \]

**Overflow Handling Methodology** No scaling implemented for overflow prevention.

**Special Requirements** None

**Implementation Notes** None

**Example** See examples/decim subdirectory

**Benchmarks (preliminary)**

<table>
<thead>
<tr>
<th>Cycles</th>
<th>Core: ((nx/D) \times (10 + nh + (D-1)))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead</td>
<td>67</td>
</tr>
<tr>
<td>Code size</td>
<td>144 (in bytes)</td>
</tr>
</tbody>
</table>
**Interpolating FIR Filter**

```c
ushort oflag = firinterp (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nh, ushort nx, ushort I)
```

(defined in interp.asm)

**Arguments**

- **x [nx]** Pointer to real input vector of nx real elements.
- **h[nh]** Pointer to coefficient vector of size nh in normal order:
  
  \[H = b_0 b_1 b_2 b_3 \ldots\]

- **r[nx*I]** Pointer to real output vector of nx real elements. In-place computation \((r = x)\) is allowed
- **dbuffer[(nh/I)+1]** Delay buffer of \((nh/I)+1\) elements
  
  - In the case of multiple-buffering schemes, this array should be initialized to 0 for the first block only. Between consecutive blocks, the delay buffer preserves delayed input samples in `dbuffer[1...(nh/I)+1]`. It also preserves a ptr to the next new entry into the `dbuffer`. This ptr is preserved across function calls in `dbuffer[0]`.
  
  - The delay buffer is only \(nh/I\) elements and holds only delayed \(x\) inputs. No zero-samples are inserted into `dbuffer` (since only non-zero products contribute to the filter output).

- **nx** Number of real elements in vector \(x\) and \(r\)
- **nh** Number of coefficients, with \((nh/I) \geq 3\)
- **I** Interpolation factor. \(I\) is effectively the number of output samples for every input sample. This routine can be used with \(I=1\).
- **oflag** Overflow error flag
  
  - If `oflag = 1`, a 32-bit data overflow occurred in an intermediate or final result.
  
  - If `oflag = 0`, a 32-bit overflow has not occurred.
**Description**
Computes an interpolating real FIR filter (direct-form) using coefficient stored in vector h. The real data input is stored in vector x. The filter output result is stored in vector r. This function retains the address of the delay filter memory d containing the previous delayed values to allow consecutive processing of blocks. This function can be used for both block-by-block and sample-by-sample filtering (nx = 1).

**Algorithm**

\[
    r[t] = \sum_{k=0}^{nh} h[k]x[t/I - k] \quad 0 \leq j \leq nr
\]

**Overflow Handling Methodology**
No scaling implemented for overflow prevention.

**Special Requirements**
None

**Implementation Notes**
None

**Example**
See examples/decimate subdirectory

**Benchmarks (preliminary)**

<table>
<thead>
<tr>
<th>Cycles</th>
<th>Core:</th>
</tr>
</thead>
<tbody>
<tr>
<td>If I &gt; 1</td>
<td></td>
</tr>
<tr>
<td>nx*(2+I*(1+(nh/I)))</td>
<td></td>
</tr>
<tr>
<td>If I=1:</td>
<td></td>
</tr>
<tr>
<td>nx*(2+nh)</td>
<td></td>
</tr>
</tbody>
</table>

| Overhead | 72 |

| Code size | 164 |
Lattice Forward (FIR) Filter

`ushort oflag = firlat (DATA *x, DATA *h, DATA *r, DATA *pbuffer, int nx, int nh)`

**Arguments**

- `x [nx]`- Pointer to real input vector of nx real elements in normal order:
  - `x[0]`
  - `x[1]`
  - ...
  - `x[nx–2]`
  - `x[nx–1]`

- `h[nh]`- Pointer to lattice coefficient vector of size nh in normal order:
  - `h[0]`
  - `h[1]`
  - ...
  - `h[nh–2]`
  - `h[nh–1]`

- `r[nx]`- Pointer to output vector of nx real elements. In-place computation (`r = x`) is allowed.
  - `r[0]`
  - `r[1]`
  - ...
  - `r[nx–2]`
  - `r[nx–1]`
firlat  Lattice Forward (FIR) Filter

pbuffer [nh]  Delay buffer

- In the case of multiple-buffering schemes, this array should be initialized to 0 for the first block only. Between consecutive blocks, the delay buffer preserves the previous r output elements needed.
- pbuffer: procession buffer of nh length in order:
  - e’0[n–1]
  - e’1[n–1]
  - ... 
  - e’nh–2[n–1]
  - e’nh–1[n–1]

nx  Number of real elements in vector x (input samples)

nh  Number of coefficients

oflag  Overflow error flag

- If oflag = 1, a 32-bit data overflow has occurred in an intermediate or final result
- If oflag = 0, a 32-bit overflow has not occurred.

Description  Computes a real lattice FIR filter implementation using coefficient stored in vector h. The real data input is stored in vector x. The filter output result is stored in vector r. This function retains the address of the delay filter memory containing the previous delayed values to allow consecutive processing of blocks. This function can be used for both block-by-block and sample-by-sample filtering (nx=1)

Algorithm

\[
e_0[n] = e'_0[n] = x[n],
\]
\[
e[i][n] = e[i-1][n] - h[i] e'[i-1][n-1], \quad i = 1, 2, ..., N
\]
\[
e'[i][n] = h[i] e'[i-1][n] + e'[i-1][n-1], \quad i = 1, 2, ..., N
\]
\[
y[n] = e_0[n]
\]

Overflow Handling Methodology  No scaling implemented for overflow prevention.

Special Requirements  None

Implementation Notes  None
Example

See examples/firlat subdirectory

Benchmarks (preliminary)

Cycles†

Core: $4 \times nh \times nx + 1$

Overhead: 23

Code size

53

(in bytes)

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
Symmetric FIR Filter

```c
ushort oflag = firs (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh2)
```

**Arguments**

- **x[nx]** Pointer to input vector of nx real elements.
- **r[nx]** Pointer to output vector of nx real elements. In-place computation \((r = x)\) is allowed.
- **h[nh2]** Pointer to coefficient vector containing the first half of the symmetric filter coefficients. For example, if the filter coefficients are \(\{h_0, h_1, h_2, h_2, h_1, h_0\}\), then \(h[nh2] = \{h_0, h_1, h_2\}\) where \(h_0\) resides at the lowest memory address in the array.
  - This array must be located in internal memory since it is accessed by the C55x coefficient bus.
- **dbuffer[2*nh2 + 2]** Pointer to delay buffer of length \(nh = 2*nh2 + 2\)
  - In the case of multiple-buffering schemes, this array should be initialized to 0 for the first filter block only. Between consecutive blocks, the delay buffer preserves the previous \(r\) output elements needed.
  - The first element in this array is special in that it contains the array index of the oldest input entry in the delay buffer. This is needed for multiple-buffering schemes, and should be initialized to 0 (like all the other array entries) for the first block only.
- **nx** Number of input samples
- **nh2** Half the number of coefficients of the filter (due to symmetry there is no need to provide the other half). For example, if the filter coefficients are \(\{h_0, h_1, h_2, h_2, h_1, h_0\}\), then \(nh2 = 3\). Must be a minimum value of 3. For smaller filters, zero pad the coefficients to meet the minimum value.
- **oflag** Overflow error flag (returned value)
  - If \(oflag = 1\), a 32-bit data overflow occurred in an intermediate or final result.
  - If \(oflag = 0\), a 32-bit overflow has not occurred.
Description
Computes a real FIR filter (direct-form) with nh2 symmetric coefficients using the FIRS instruction approach. The filter is assumed to have a symmetric impulse response, with the first half of the filter coefficients stored in the array h. The real input data is stored in vector x. The filter output result is stored in vector r. This function maintains the array dbuffer containing the previous delayed input values to allow consecutive processing of input data blocks. This function can be used for both block-by-block (nx \( \geq 2 \)) and sample-by-sample filtering (nx = 1). In-place computation (r = x) is allowed.

Algorithm
\[
    r[j] = \sum_{k=0}^{nh2-1} h\ldots[k] \times (x[j - k] + x[j + k - 2 \times nh2 + 1]) \quad \text{where}
    0 \leq j \leq nx
\]

Overflow Handling Methodology
No scaling implemented for overflow prevention.

Special Requirements
- nh must be a minimum value of 3. For smaller filters, zero pad the h[] array.
- Coefficient array h[nh2] must be located in internal memory since it is accessed using the C55x coefficient bus, and that bus does not have access to external memory.

Implementation Notes
The first element in the dbuffer array (index = 0) is the entry index for the input history. It is treated as an unsigned 16-bit value by the function even though it has been declared as signed in C. The value of the entry index is equal to the index – 1 of the oldest input entry in the array. The remaining elements make up the input history. Figure 4–19 shows the array in memory with an entry index of 2. The newest entry in the dbuffer is denoted by x(j–0), which in this case would occupy index = 3 in the array. The next newest entry is x(j–1), and so on. It is assumed that all x() entries were placed into the array by the previous invocation of the function in a multiple-buffering scheme.

The dbuffer array actually contains one more history value than is needed to implement this filter. The value x(j–2*nh2) does not enter into the calculations for the output r(j). However, this value is required in other DSPLIB filter functions that utilize the dual-MAC units on the C55x, such as FIR2. Including this extra location ensures compatibility across all filter functions in the C55x DSPLIB.

Figure 4–19, Figure 4–20, and Figure 4–21 show the dbuffer, x, and r arrays as they appear in memory.
Symmetric FIR Filter

Figure 4–19. dbuffer Array in Memory at Time \( j \)

- Entry index = 2
- \( x(j-2\cdot nh2-2) \)
- \( x(j-2\cdot nh2-1) \)
- \( x(j-2\cdot nh2) \)
- \( x(j-0) \)
- \( x(j-1) \)
- \( x(j-2) \)

Figure 4–20. \( x \) Array in Memory

- Oldest \( x \) entry
- Newest \( x \) entry

Figure 4–21. \( r \) Array in Memory

- Oldest \( x \) entry
- Newest \( x \) entry
### Example

See examples/firs subdirectory

### Benchmarks (preliminary)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles†</td>
<td></td>
</tr>
<tr>
<td>Core:</td>
<td>$nx \times (3 + nh2)$</td>
</tr>
<tr>
<td>Overhead</td>
<td>31</td>
</tr>
<tr>
<td>Code size (in bytes)</td>
<td>133</td>
</tr>
</tbody>
</table>

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
**fltoq15**  
*Floating-point to Q15 Conversion*

**fltoq15**

```c
ushort errorcode = fltoq15 (float *x, DATA *r, ushort nx)
```

(defined in fltoq15.asm)

**Arguments**

- **x[nx]** Pointer to floating-point input vector of size nx. x should contain the numbers normalized between (–1,1). The errorcode returned value will reflect if that condition is not met.
- **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
- **errorcode** The function returns the following error codes:
  - 1 – if any element is too large to represent in Q15 format
  - 2 – if any element is too small to represent in Q15 format
  - 3 – both conditions 1 and 2 were encountered

**Description**

Convert the IEEE floating-point numbers stored in vector x into Q15 numbers stored in vector r. The function returns the error codes if any element x[i] is not representable in Q15 format.

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**

Not applicable

**Overflow Handling Methodology** Saturation implemented for overflow handling

**Special Requirements** None

**Implementation Notes** None

**Example**

See examples/expn subdirectory
Benchmarks (preliminary)

Cycles† Core:  
17 * nx (if x[n] == 0)  
23 * nx (if x[n] is too small for Q15 representation)  
32 * nx (if x[n] is too large for Q15 representation)  
38 * nx (otherwise)  

Overhead: 23

Code size 157
(in bytes)

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
usshort oflag = hilb16 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh)

Arguments

- **x[nx]** Pointer to input vector of nx real elements.
- **h[nh]** Pointer to coefficient vector of size nh in normal order. H = {h0, h1, h2, h3, h4, ...} Every odd valued filter coefficient has to be 0, i.e. h1 = h3 = ... = 0. And H = {h0, 0, h2, 0, h4, 0, ...} where h0 resides at the lowest memory address in the array.
- **r[nx]** Pointer to output vector of nx real elements. In-place computation (r = x) is allowed.
- **dbuffer[nh + 2]** Pointer to delay buffer of length nh = nh + 2
  - In the case of multiple-buffering schemes, this array should be initialized to 0 for the first filter block only. Between consecutive blocks, the delay buffer preserves the previous r output elements needed.
  - The first element in this array is special in that it contains the array index-1 of the oldest input entry in the delay buffer. This is needed for multiple-buffering schemes, and should be initialized to zero (like all the other array entries) for the first block only.
- **nx** Number of real elements in vector x (input samples)
- **nh** The number of coefficients of the filter. For example if the filter coefficients are {h0, h1, h2, h3, h4, h5}, then nh = 6. Must be a minimum value of 6. For smaller filters, zero pad the coefficients to meet the minimum value.
- **oflag** Overflow error flag (returned value)
  - If oflag = 1, a 32-bit data overflow occurred in an intermediate or final result.
  - If oflag = 0, a 32-bit overflow has not occurred.
Description
Computes a real FIR filter (direct-form) using the coefficients stored in vector \( h \). The real input data is stored in vector \( x \). The filter output result is stored in vector \( r \). This function maintains the array \( \text{dbufbuffer} \) containing the previous delayed input values to allow consecutive processing of input data blocks. This function can be used for both block-by-block (\( nx \geq 2 \)) and sample-by-sample filtering (\( nx = 1 \)). In place computation (\( r = x \)) is allowed.

Algorithm

\[
\begin{align*}
    r[j] &= \sum_{k=0}^{nh-1} h[k] x[j - k] \\
    \text{where} \\
    0 &\leq j \leq nx
\end{align*}
\]

Overflow Handling Methodology
No scaling implemented for overflow prevention.

Special Requirements

- Every odd valued filter coefficient has to be 0. This is a requirement for the Hilbert transformer. For example, a 6 tap filter may look like this: \( H = [0.867 \ 0 -0.324 \ 0 -0.002 \ 0] \)

- Always pad 0 to make \( nh \) as a even number. For example, a 5 tap filter with a zero pad may look like this: \( H = [0.867 \ 0 -0.324 \ 0 -0.002 \ 0] \)

- \( nh \) must be a minimum value of 6. For smaller filters, zero pad the \( H[] \) array.

Implementation Notes
The first element in the \( \text{dbufbuffer} \) array (index = 0) is the entry index for the input history. It is treated as an unsigned 16-bit value by the function even though it has been declared as signed in C. The value of the entry index is equal to the index – 1 of the oldest input entry in the array. The remaining elements make up the input history. Figure 4–22 shows the array in memory with an entry index of 2. The newest entry in the \( \text{dbufbuffer} \) is denoted by \( x[j-0] \), which in this case would occupy index = 3 in the array. The next newest entry is \( x[j-1] \), and so on. It is assumed that all \( x() \) entries were placed into the array by the previous invocation of the function in a multiple-buffering scheme.

The \( \text{dbufbuffer} \) array actually contains one more history value than is needed to implement this filter. The value \( x[j-nh] \) does not enter into the calculations for the output \( r(j) \). However, this value is required in other DSPLIB filter functions that utilize the dual-MAC units on the C55x, such as \( \text{FIR2} \). Including this extra location ensures compatibility across all filter functions in the C55x DSPLIB.
Figure 4–22, Figure 4–23, and Figure 4–24 show the dbuffer, x, and r arrays as they appear in memory.

**Figure 4–22. dbuffer Array in Memory at Time j**

- Entry index = 2
- x(j–nh–2)
- x(j–nh–1)
- x(j–nh)
- x(j–0)
- x(j–1)
- x(j–2)
- ...
- x(j–nh–5)
- x(j–nh–4)
- x(j–nh–3)

**Figure 4–23. x Array in Memory**

- Oldest x( ) entry
- x(0)
- x(1)
- ...
- x(nx–2)
- x(nx–1)

**Figure 4–24. r Array in Memory**

- Oldest x( ) entry
- r(0)
- r(1)
- ...
- r(nx–2)
- r(nx–1)
Example

See examples/hilb16 subdirectory

Benchmarks (preliminary)

<table>
<thead>
<tr>
<th>Cycles</th>
<th>Core: ( nx(2+nh/2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead:</td>
<td>28</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code size (in bytes)</th>
<th>108</th>
</tr>
</thead>
</table>
**Double-precision IIR Filter**

```c
ushort oflag = iir32 (DATA *x, LDATA *h, DATA *r, LDATA *dbuffer, ushort nbiq, ushort nr)
```

(defined in iir32.asm)

**Arguments**

- **x [nr]**  
  Pointer to input data vector of size nr

- **h[5*nbiq]**  
  Pointer to the 32-bit filter coefficient vector with the following format. For example for nbiq= 2, h is equal to:

  - `b21` – high  
  - `b21` – low  
  - `b11` – high  
  - `b11` – low  
  - `b01` – high  
  - `b01` – low  
  - `a21` – high  
  - `a21` – low  
  - `a11` – high  
  - `a11` – low  
  - `b22` – high  
  - `b22` – low  
  - `b12` – high  
  - `b12` – low  
  - `b02` – high  
  - `b02` – low  
  - `a22` – high  
  - `a22` – low  
  - `a12` – high  
  - `a12` – low  

- **r[nr]**  
  Pointer to output data vector of size nr. r can be equal or less than x.
Double–precision IIR Filter  

**iir32**

**Function Descriptions**

- **dbuffer[2*nbiq+2]**  
  Pointer to address of 32-bit delay line dbuffer. Each biquad has 3 consecutive delay line elements. For example for nbiq = 2:

  - d1(n–2) – low  
  - d1(n–2) – high  
  - d1(n–1) – low  
  - d1(n–1) – high  
  - beginning of biquad 1

  - d2(n–2) – low  
  - d2(n–2) – high  
  - d2(n–1) – low  
  - d2(n–1) – high  
  - beginning of biquad 2

- In the case of multiple-buffering schemes, this array should be initialized to 0 for the first block only. Between consecutive blocks, the delay buffer preserves the previous r output elements needed.

- Memory alignment: None required for C5510. This is a group of circular buffers. Each biquad’s delay buffer is treated separately. The Buffer Start Address (BSAxx) updated to a new location for each biquad.

- **nbiq**  
  Number of biquads

- **nr**  
  Number of elements of input and output vectors

- **oflag**  
  Overflow flag.

  - If oflag = 1, a 32-bit overflow has occurred
  - If oflag = 0, a 32-bit overflow has not occurred

**Description**

Computes a cascaded IIR filter of nbiquad biquad sections using 32-bit coefficients and 32-bit delay buffers. The input data is assumed to be single-precision (16 bits).

Each biquad section is implemented using Direct-form II. All biquad coefficients (5 per biquad) are stored in vector h. The real data input is stored in vector x. The filter output result is stored in vector r.

This function retains the address of the delay filter memory d containing the previous delayed values to allow consecutive processing of blocks. This function is more efficient for block-by-block filter implementation due to the C-calling overhead. However, it can be used for sample-by-sample filtering (nx = 1).
Double-precision IIR Filter

Algorithm (for biquad)
\[
d(n) = x(n) - a1 \cdot d(n-1) - a2 \cdot d(n-2) \\
y(n) = b0 \cdot d(n) + b1 \cdot d(n-1) + b2 \cdot d(n-2)
\]

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements None

Implementation Notes See program iircas32.asm

Example See examples/iir32 subdirectory

Benchmarks (preliminary)

<table>
<thead>
<tr>
<th>Cycles</th>
<th>Core: ( nx \cdot (7 + 31 \cdot nbiq) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead:</td>
<td>77</td>
</tr>
</tbody>
</table>

| Code size (in bytes) | 203 |
Cascaded IIR Direct Form II Using 4 Coefficients per Biquad

ushort oflag = iircas4 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nbiq, ushort nx)

(defined in iir4cas4.asm)

Arguments

x [nx] Pointer to input data vector of size nx
h[4*nbiq] Pointer to filter coefficient vector with the following format:
h = a11 a21 b21 b11 ....a1i a2i b2i b1i
where i is the biquad index (a21 is the a2 coefficient of biquad 1). Pole (recursive) coefficients = a. Zero (non-recursive) coefficients = b
r[nx] Pointer to output data vector of size nx. r can be equal than x.
dbuffer[2*nbiq] Pointer to address of delay line d.
Each biquad has 2 delay line elements separated by nbiq locations in the following format:
d1(n–1), d2(n–1)…di(n–1) d1(n–2), d2(n–2)…di(n–2)
where i is the biquad index (d2(n–1) is the (n–1)th delay element for biquad 2).

☐ In the case of multiple-buffering schemes, this array should be initialized to 0 for the first block only. Between consecutive blocks, the delay buffer preserves the previous r output elements needed.

☐ Memory alignment: this is a circular buffer and must start in a k-bit boundary (that is, the k LSBs of the starting address must be zeros) where k = log2 (2*nbiq).

nbiq Number of biquads
nx Number of elements of input and output vectors
oflag Overflow flag.

☐ If oflag = 1, a 32-bit overflow has occurred
☐ If oflag = 0, a 32-bit overflow has not occurred
iircas4  Cascaded IIR Direct Form II Using 4 Coefficients per Biquad

Description
Computes a cascade IIR filter of nbiq biquad sections. Each biquad section is implemented using Direct-form II. All biquad coefficients (4 per biquad) are stored in vector h. The real data input is stored in vector x. The filter output result is stored in vector r.

This function retains the address of the delay filter memory d containing the previous delayed values to allow consecutive processing of blocks. This function is more efficient for block-by-block filter implementation due to the C-calling overhead. However, it can be used for sample-by-sample filtering (nx = 1).

Algorithm
(for biquad)
\[
d(n) = x(n) - a1 \cdot d(n - 1) - a2 \cdot d(n - 2)
\]
\[
y(n) = d(n) + b1 \cdot d(n - 1) + b2 \cdot d(n - 2)
\]

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements None

Implementation Notes None

Example  See examples/iircas4 subdirectory

Benchmarks (preliminary)

Cycles†  Core:  nx * (5 + 4 * nbiq)
          Overhead:  58

Code size  122
(in bytes)

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
Cascaded IIR Direct Form II (5 Coefficients per Biquad)

```c
ushort oflag = iircas5 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nbiq, ushort nx)
```

(defined in iircas5.asm)

**Arguments**

- **x [nx]** Pointer to input data vector of size nx
- **h[5*nbiq]** Pointer to filter coefficient vector with the following format:
  
  h = a11 a21 b21 b01 b11 ... a1i a2i b2i b0i b1i
  
  where i is the biquad index a21 is the a2 coefficient of biquad 1). Pole (recursive) coefficients = a. Zero (non-recursive) coefficients = b
- **r[nx]** Pointer to output data vector of size nx. r can be equal than x.
- **dbuffer[2*nbiq]** Pointer to address of delay line d. Each biquad has 2 delay line elements separated by nbiq locations in the following format:
  
  d1(n–1), d2(n–1)...di(n–1) d1(n–2), d2(n–2)...di(n–2)
  
  where i is the biquad index(d2(n–1) is the (n–1) th delay element for biquad 2).
  
  - In the case of multiple-buffering schemes, this array should be initialized to 0 for the first block only. Between consecutive blocks, the delay buffer preserves the previous r output elements needed.
  - Memory alignment: this is a circular buffer and must start in a k-bit boundary(that is, the k LSBs of the starting address must be zeros) where k = log2 (2*nbiq).
- **nbiq** Number of biquads
- **nx** Number of elements of input and output vectors
- **oflag** Overflow flag.
  
  - If oflag = 1, a 32-bit overflow has occurred
  - If oflag = 0, a 32-bit overflow has not occurred
Computes a cascade IIR filter of nbiq biquad sections. Each biquad section is implemented using Direct-form II. All biquad coefficients (5 per biquad) are stored in vector h. The real data input is stored in vector x. The filter output result is stored in vector r.

This function retains the address of the delay filter memory d containing the previous delayed values to allow consecutive processing of blocks. This function is more efficient for block-by-block filter implementation due to the C-calling overhead. However, it can be used for sample-by-sample filtering (nx = 1).

The usage of 5 coefficients instead of 4 facilitates the design of filters with a unit gain of less than 1 (for overflow avoidance), typically achieved by filter coefficient scaling.

\[
\begin{align*}
d(n) &= x(n) - a_1 \cdot d(n - 1) - a_2 \cdot d(n - 2) \\
y(n) &= b_0 \cdot d(n) + b_1 \cdot d(n - 1) + b_2 \cdot d(n - 2)
\end{align*}
\]

Overflow Handling Methodology No scaling implemented for overflow prevention.

Special Requirements None

Implementation Notes None

Example See examples/iircas5 subdirectory

Benchmarks (preliminary)

Cycles† Core: \(nx \cdot (5 \cdot 5 \cdot nbiq)\)
Overhead: 60

Code size 126
(in bytes)

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
Cascaded IIR Direct Form I (5 Coefficients per Biquad) iircas51

ushort oflag = iircas51 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nbiq, ushort nx)

(defined in iircas51.asm)

Arguments

x [nx] Pointer to input data vector of size nx

h[5*nbiq] Pointer to filter coefficient vector with the following format:

\[ h = b01 b11 b21 a11 a21 ... b0i b1i b2i a1i a2i \]

where i is the biquad index (a21 is the a2 coefficient of biquad 1). Pole (recursive) coefficients = a. Zero (non-recursive) coefficients = b

r[nx] Pointer to output data vector of size nx. r can be equal than x.

dbuffer[4*nbiq] Pointer to address of delay line dbuffer. Each biquad has 4 delay line elements stored consecutively in memory in the following format:

\[ x1(n-1), x1(n-2), y1(n-1), y1(n-2) ... xi(n-2), xi(n-2), yi(n-1), yi(n-2) \]

where i is the biquad index(x1(n−1) is the (n−1)th delay element for biquad 1).

- In the case of multiple-buffering schemes, this array should be initialized to 0 for the first block only. Between consecutive blocks, the delay buffer preserves the previous r output elements needed.
- Memory alignment: No need for memory alignment.

nbiq Number of biquads

nx Number of elements of input and output vectors

oflag Overflow flag.

- If oflag = 1, a 32-bit overflow has occurred.
- If oflag = 0, a 32-bit overflow has not occurred.
iircas51  Cascaded IIR Direct Form I (5 Coefficients per Biquad)

Description  Computes a cascade IIR filter of nbiq biquad sections. Each biquad section is implemented using Direct-form I. All biquad coefficients (5 per biquad) are stored in vector h. The real data input is stored in vector x. The filter output result is stored in vector r.

This function retains the address of the delay filter memory d containing the previous delayed values to allow consecutive processing of blocks. This function is more efficient for block-by-block filter implementation due to the C-calling overhead. However, it can be used for sample-by-sample filtering (nx = 1).

The usage of 5 coefficients instead of 4 facilitates the design of filters with a unit gain of less than 1 (for overflow avoidance), typically achieved by filter coefficient scaling.

Algorithm  (for biquad)
\[ y(n) = b0 \times x(n) + b1 \times x(n-1) + b2 \times x(n-2) - a1 \times y(n-1) - a2 \times y(n-2) \]

Overflow Handling Methodology  No scaling implemented for overflow prevention.

Special Requirements  None

Implementation Notes  None

Example  See examples/iircas51 subdirectory

Benchmarks (preliminary)

<table>
<thead>
<tr>
<th>Cycles†</th>
<th>Core: nx * (5 + 8 * nbiq)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead:</td>
<td>68</td>
</tr>
</tbody>
</table>

| Code size | 154 |
| (in bytes) | |

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
ushort oflag = iirlat (DATA *x, DATA *h, DATA *r, DATA *pbuffer, int nx, int nh)

Arguments

x [nx] Pointer to real input vector of nx real elements in normal order:
   x[0]
   x[1]
   .
   .
   x[nx–2]
   x[nx–1]

h[nh] Pointer to lattice coefficient vector of size nh in normal order with the first element zero-padded:
   0
   h[0]
   h[1]
   .
   .
   h[nh–2]
   h[nh–1]

r[nx] Pointer to output vector of nx real elements. In-place computation (r = x) is allowed.
   r[0]
   r[1]
   .
   .
   r[nx–2]
   r[nx–1]

pbuffer[nh] Delay buffer
   In the case of multiple-buffering schemes, this array should be initialized to 0 for the first block only. Between consecutive blocks, the delay buffer preserves the previous r output elements needed.

nx Number of real elements in vector x (input samples)
**iirlat  Lattice Inverse (IIR) Filter**

- **nh**  Number of coefficients
- **oflag**  Overflow error flag
  - If oflag = 1, a 32-bit data overflow has occurred in an intermediate or final result.
  - If oflag = 0, a 32-bit overflow has not occurred.

**Description**  Computes a real lattice IIR filter implementation using coefficient stored in vector h. The real data input is stored in vector x. The filter output result is stored in vector r. This function retains the address of the delay filter memory d containing the previous delayed values to allow consecutive processing of blocks. This function can be used for both block-by-block and sample-by-sample filtering (nx = 1)

**Algorithm**

\[
e_i[n] = x[n],
\]

\[
e_{i-1}[n] = e_i[n] - h_i e_{i-1}[n - 1], \quad i = N, (N-1), ..., 1
\]

\[
e'[n] = -k_i e_{i-1} + e'_{i-1}[n - 1], \quad i = N, (N-1), ..., 1
\]

\[
y[n] = e_0[n] = e'_{[n]}
\]

**Overflow Handling Methodology**  No scaling implemented for overflow prevention.

**Special Requirements**  None

**Implementation Notes**  None

**Example**  See examples/iirlat subdirectory

**Benchmarks (preliminary)**

- **Cycles**  Core: \(4 \times (nh - 1) \times nx\)
  
  Overhead: 24

- **Code size**  54
  
  (in bytes)

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
32-bit by 16-bit Long Division Function

void ldiv16 (LDATA *x, DATA *y, DATA *r, DATA *rexp, ushort nx)

Arguments

x[nx]  Pointer to input data vector 1 of size nx
    x[0]
    x[1]
    .
    .
    x[nx–2]
    x[nx–1]

r[nx]  Pointer to output data buffer
    r[0]
    r[1]
    .
    .
    r[nx–2]
    r[nx–1]

rexp[nx]  Pointer to exponent buffer for output values. These
          exponent values are in integer format.
    rexp[0]
    rexp[1]
    .
    .
    rexp[nx–2]
    rexp[nx–1]

nx  Number of elements of input and output vectors

Description

This routine implements a long division function of a Q31 value divided by a
Q15 value. The reciprocal of the Q15 value, y, is calculated then multiplied by
the Q31 value, x. The result is returned as an exponent such that:

r[i] * rexp[i] = true reciprocal in floating-point

Algorithm

The reciprocal of the Q15 number is calculated using the following equation:

\[ Ym = 2^* Ym - Ym^2 * Xnorm \]

If we start with an initial estimate of Ym, the equation converges to a solution
very rapidly (typically 3 iterations for 16-bit resolution).
The initial estimate can be obtained from a look-up table, from choosing a midpoint, or simply from linear interpolation. The method chosen for this problem is linear interpolation and is accomplished by taking the complement of the least significant bits of the Xnorm value.

The reciprocal is multiplied by the Q31 number to generate the output.

**Overflow Handling Methodology** None

**Special Requirements** None

**Implementation Notes** None

**Example** See examples/ldiv16 subdirectory

**Benchmarks (preliminary)**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles†</td>
<td>Core: 4 * nx Overhead: 14</td>
</tr>
<tr>
<td>Code size (in bytes)</td>
<td>91</td>
</tr>
</tbody>
</table>

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
**Base 10 Logarithm**

ushort oflag = log_10 (DATA *x, LDATA *r, ushort nx)

(defined in log_10.asm)

**Arguments**

- x[nx] Pointer to input vector of size nx.
- r[nx] Pointer to output data vector (Q31 format) of size nx.
- nx Length of input and output data vectors
- oflag Overflow flag.
  - If oflag = 1, a 32-bit overflow has occurred.
  - If oflag = 0, a 32-bit overflow has not occurred.

**Description**

Computes the log base 10 of elements of vector x using Taylor series.

**Algorithm**

\[
\text{for } (i = 0; i < nx; i + + ) \quad y(i) = \log_{10} x(i) \text{ where } -1 < x(i) < 1
\]

**Overflow Handling Methodology**

No scaling implemented for overflow prevention

**Special Requirements**

None

**Implementation Notes**

\[
y = 0.4343 \ln(x) \text{ with } x = M(x)2^P(x) = M2^P
\]

\[
y = 0.4343 \left( \ln(M) + \ln(2)P \right)
\]

\[
y = 0.4343 \left( \ln(2M) + (P-1)\ln(2) \right)
\]

\[
y = 0.4343 \left( \ln((2M-1)+1) + (P-1)\ln(2) \right)
\]

\[
y = 0.4343 \left( (f(2M-1) + (P-1)\ln(2) \right)
\]

with \( f(u) = \ln(1+u) \).

We use a polynomial approximation for \( f(u) \):

\[
f(u) = (((((C6u+C5)u+C4)u+C3)u+C2)u+C1)u+C0
\]

for \( 0 \leq u \leq 1 \).

The polynomial coefficients \( C_i \) are as follows:

- \( C0 = 0.000\ 001\ 472 \)
- \( C1 = 0.999\ 847\ 766 \)
- \( C2 = -0.497\ 373\ 368 \)
- \( C3 = 0.315\ 747\ 760 \)
- \( C4 = -0.190\ 354\ 944 \)
- \( C5 = 0.082\ 691\ 584 \)
- \( C6 = -0.017\ 414\ 144 \)
The coefficients $B_i$ used in the calculation are derived from the $C_i$ as follows:

- $B_0 = Q_{30} = 1581d = 0062Dh$
- $B_1 = Q_{14} = 16381d = 03FFDh$
- $B_2 = Q_{15} = -16298d = 0C056h$
- $B_3 = Q_{16} = 20693d = 050D5h$
- $B_4 = Q_{17} = -24950d = 09E8Ah$
- $B_5 = Q_{18} = 21677d = 054ADh$
- $B_6 = Q_{19} = -9130d = 0DC56h$

Example

See examples/log_10 subdirectory

Benchmarks (preliminary)

- Cycles†
  - Core: $35 \times nx$
  - Overhead: $40$
- Code size (in bytes)
  - $162$

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
**Base 2 Logarithm**

`ushort oflag = log_2 (DATA *x, LDATA *r, ushort nx)`

(defined in `log_2.asm`)

**Arguments**

- `x[nx]` Pointer to input vector of size `nx`.
- `r[nx]` Pointer to output data vector (Q31 format) of size `nx`.
- `nx` Length of input and output data vectors
- `oflag` Overflow flag.
  - If `oflag = 1`, a 32-bit overflow has occurred.
  - If `oflag = 0`, a 32-bit overflow has not occurred.

**Description**

Computes the log base 2 of elements of vector `x` using Taylor series.

**Algorithm**

for (i = 0; i < nx; i += ) \[ y(i) = \log_2 x(i) \text{ where } 0 < x(i) < 1 \]

**Overflow Handling Methodology**

No scaling implemented for overflow prevention

**Special Requirements**

None

**Implementation Notes**

\[ y = 1.4427 \times \ln(x) \text{ with } x = M(x) \times 2^P(x) = M \times 2^P \]

\[ y = 1.4427 \times (\ln(M) + \ln(2)^P) \]

\[ y = 1.4427 \times (\ln(2^M) + (P-1)\ln(2)) \]

\[ y = 1.4427 \times (\ln(2^M-1)+1 + (P-1)\ln(2)) \]

\[ y = 1.4427 \times (f(2^M-1) + (P-1)\ln(2)) \]

with \( f(u) = \ln(1+u) \).

We use a polynomial approximation for \( f(u) \):

\[ f(u) = (((C6\times u + C5)\times u + C4)\times u + C3)\times u + C2)\times u + C1)\times u + C0 \]

for \( 0 \leq u \leq 1 \).

The polynomial coefficients \( C_i \) are as follows:

- \( C_0 = 0.000 001 472 \)
- \( C_1 = 0.999 847 766 \)
- \( C_2 = -0.497 373 368 \)
- \( C_3 = 0.315 747 760 \)
- \( C_4 = -0.190 354 944 \)
- \( C_5 = 0.082 691 584 \)
- \( C_6 = -0.017 414 144 \)
The coefficients $B_i$ used in the calculation are derived from the $C_i$ as follows:

- $B_0 \quad Q_{30} \quad 1581d \quad 0062Dh$
- $B_1 \quad Q_{14} \quad 16381d \quad 03FFDh$
- $B_2 \quad Q_{15} \quad -16298d \quad 0C056h$
- $B_3 \quad Q_{16} \quad 20693d \quad 050D5h$
- $B_4 \quad Q_{17} \quad -24950d \quad 09E8Ah$
- $B_5 \quad Q_{18} \quad 21677d \quad 054ADh$
- $B_6 \quad Q_{19} \quad -9130d \quad 0DC56h$

**Example**

See examples/log_2 subdirectory

**Benchmarks (preliminary)**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles†</td>
<td>Core: 37 * nx</td>
<td>Overhead: 35</td>
</tr>
<tr>
<td>Code size</td>
<td>166 (in bytes)</td>
<td></td>
</tr>
</tbody>
</table>

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
Base e Logarithm (natural logarithm)

logn

ushort oflag = logn (DATA *x, LDATA *r, ushort nx)

(defined in logn.asm)

Arguments

x[nx] Pointer to input vector of size nx.
r[nx] Pointer to output data vector (Q31 format) of size nx.
nx Length of input and output data vectors
oflag Overflow flag.
  □ If oflag = 1, a 32-bit overflow has occurred.
  □ If oflag = 0, a 32-bit overflow has not occurred.

Description Computes the log base e of elements of vector x using Taylor series.

Algorithm for (i = 0; i < nx; i + +)  \( y(i) = \log nx(i) \) where \(-1 < x(i) < 1\)

Overflow Handling Methodology No scaling implemented for overflow prevention

Special Requirements None

Implementation Notes \( y = 0.4343 \times \ln(x) \) with \( x = M(x) \times 2^P(x) = M \times 2^P \)
\( y = 0.4343 \times (\ln(M) + \ln(2) \times P) \)
\( y = 0.4343 \times (\ln(2^M) + (P-1) \times \ln(2)) \)
\( y = 0.4343 \times (\ln((2^M-1)+1) + (P-1) \times \ln(2)) \)
\( y = 0.4343 \times (f(2^M-1) + (P-1) \times \ln(2)) \)
with \( f(u) = \ln(1+u). \)

We use a polynomial approximation for \( f(u) \):
\( f(u) = (((((C6*u+C5)*u+C4)*u+C3)*u+C2)*u+C1)*u+C0 \)
for \( 0 <= u <= 1. \)

The polynomial coefficients \( C_i \) are as follows:
\( C0 = 0.000 \ 001 \ 472 \)
\( C1 = 0.999 \ 847 \ 766 \)
\( C2 = -0.497 \ 373 \ 368 \)
\( C3 = 0.315 \ 747 \ 760 \)
\( C4 = -0.190 \ 354 \ 944 \)
\( C5 = 0.082 \ 691 \ 584 \)
\( C6 = -0.017 \ 414 \ 144 \)
The coefficients $B_i$ used in the calculation are derived from the $C_i$ as follows:

<table>
<thead>
<tr>
<th>$B_0$</th>
<th>Q30</th>
<th>1581d</th>
<th>0062Dh</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_1$</td>
<td>Q14</td>
<td>16381d</td>
<td>03FFDh</td>
</tr>
<tr>
<td>$B_2$</td>
<td>Q15</td>
<td>-16298d</td>
<td>0C056h</td>
</tr>
<tr>
<td>$B_3$</td>
<td>Q16</td>
<td>20693d</td>
<td>050D5h</td>
</tr>
<tr>
<td>$B_4$</td>
<td>Q17</td>
<td>-24950d</td>
<td>09E8Ah</td>
</tr>
<tr>
<td>$B_5$</td>
<td>Q18</td>
<td>21677d</td>
<td>054ADh</td>
</tr>
<tr>
<td>$B_6$</td>
<td>Q19</td>
<td>-9130d</td>
<td>0DC56h</td>
</tr>
</tbody>
</table>

Example

See examples/logn subdirectory

Benchmarks (preliminary)

- Cycles†: $25 \times nx$
- Overhead: 35
- Code size (in bytes): 132

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
short r = maxidx (DATA *x, ushort nx)
(defined in maxidx.asm)

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[nx]</td>
<td>Pointer to input vector of size nx.</td>
</tr>
<tr>
<td>r</td>
<td>Index for vector element with maximum value.</td>
</tr>
<tr>
<td>nx</td>
<td>Length of input data vector (nx ≥ 6)</td>
</tr>
</tbody>
</table>

Description

Returns the index of the maximum element of a vector x. In case of multiple maximum elements, r contains the index of the first maximum element found.

Algorithm

Not applicable

Overflow Handling Methodology

Not applicable

Special Requirements

None

Implementation Notes

None

Example

See examples/maxidx subdirectory

Benchmarks (preliminary)

<table>
<thead>
<tr>
<th>Cycles†</th>
<th>Core: nx * 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead: 7</td>
<td></td>
</tr>
</tbody>
</table>

| Code size (in bytes) | 26 |

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
**maxval**  
*Maximum Value of a Vector*

```c
short r = maxval (DATA *x, ushort nx)
```

(defined in maxval.asm)

**Arguments**

- `x[nx]` Pointer to input vector of size `nx`
- `r` Maximum value of a vector
- `nx` Length of input data vector

**Description**

Returns the maximum element of a vector `x`.

**Algorithm**

Not applicable

**Overflow Handling Methodology**

Not applicable

**Special Requirements**

None

**Implementation Notes**

None

**Example**

See examples/maxval subdirectory

**Benchmarks (preliminary)**

<table>
<thead>
<tr>
<th>Cycles†</th>
<th>Core: <code>nx</code></th>
<th>Overhead: 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code size</td>
<td>20</td>
<td>(in bytes)</td>
</tr>
</tbody>
</table>

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
maxvec  Index and Value of the Maximum Element of a Vector

```c
void maxvec (DATA *x, ushort nx, DATA *r_val, DATA *r_idx)
```

(defined in maxvec.asm)

**Arguments**

- `x[nx]` Pointer to input vector of size `nx`.
- `r_val` maximum value
- `r_idx` Index for vector element with maximum value
- `nx` Length of input data vector (`nx ≥ 6`)

**Description**

This function finds the index for vector element with maximum value. In case of multiple maximum elements, `r_idx` contains the index of the first maximum element found. `r_val` contains the maximum value.

**Algorithm**

Not applicable

**Overflow Handling Methodology** Not applicable

**Special Requirements** None

**Implementation Notes** None

**Example** See examples/maxvec subdirectory

**Benchmarks (preliminary)**

- **Cycles** Core: `nx*3`
  Overhead: 8
- **Code size** (in bytes) 26
**minidx**  
*Index of the Minimum Element of a Vector*

**minidx**  
*Index of the Minimum Element of a Vector*

short \( r = \text{minidx} \) (DATA *x, ushort \( nx \))

(defined in minidx.asm)

**Arguments**

- **x[nx]**  
  Pointer to input vector of size \( nx \).
- **r**  
  Index for vector element with minimum value
- **nx**  
  Length of input data vector

**Description**  
Returns the index of the minimum element of a vector \( x \). In case of multiple minimum elements, \( r \) contains the index of the first minimum element found.

**Algorithm**  
Not applicable

**Overflow Handling Methodology**  
Not applicable

**Special Requirements**  
None

**Implementation Notes**  
None

**Example**  
See examples/minidx subdirectory

**Benchmarks (preliminary)**

- **Cycles**†  
  Core:  \( nx \times 3 \)
  Overhead:  7
- **Code size**  
  26

  (in bytes)

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
Minimum Value of a Vector  

minval

short r = minval (DATA *x, ushort nx)

(defined in minval.asm)

Arguments

x[nx] Pointer to input vector of size nx.
r Minimum value of a vector
nx Length of input data vector

Description

Returns the minimum element of a vector x.

Algorithm

Not applicable

Overflow Handling Methodology

Not applicable

Special Requirements

None

Implementation Notes

None

Example

See examples/minval subdirectory

Benchmarks (preliminary)

Cycles† Core: nx
Overhead: 3

Code size 20
(in bytes)

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
**minvec**  *Index and Value of the Minimum Element of a Vector*

void minvec (DATA *x, ushort nx, DATA *r_val, DATA *r_idx)

(defined in minvec.asm)

**Arguments**

- **x[nx]** Pointer to input vector of size nx.
- **r_val** Minimum value
- **r_idx** Index for vector element with minimum value
- **nx** Length of input data vector (nx ≥ 6)

**Description**

This function finds the index for vector element with minimum value. In case of multiple minimum elements, **r_idx** contains the index of the first minimum element found. **r_val** contains the minimum value.

**Algorithm**

Not applicable

**Overflow Handling Methodology** Not applicable

**Special Requirements** None

**Implementation Notes** None

**Example**

See examples/minvec subdirectory

**Benchmarks (preliminary)**

<table>
<thead>
<tr>
<th>Cycles Core</th>
<th>nx*3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead</td>
<td>8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code size (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
</tr>
</tbody>
</table>
Matrix Multiplication

```
ushort oflag = mmul (DATA *x1, short row1, short col1, DATA *x2, short row2, short col2, DATA *r)
```

(defined in mmul.asm)

**Arguments**

- \( x1[\text{row1}\times\text{col1}] \): Pointer to input vector of size \( nx \)
- Pointer to input matrix of size \( \text{row1}\times\text{col1} \)
  - \( \text{row1} \)
  - \( \text{col1} \)
- \( x2[\text{row2}\times\text{col2}] \): Pointer to input matrix of size \( \text{row2}\times\text{col2} \)
- \( \text{row2} \)
- \( \text{col2} \)
- \( r[\text{row1}\times\text{col2}] \): Pointer to output data vector of size \( \text{row1}\times\text{col2} \)
- \( \text{row1} \)
- \( \text{col1} \)
- \( \text{row2} \)
- \( \text{col2} \)

**Description**

This function multiplies two matrices

**Algorithm**

Multiply input matrix \( A \) (\( M \) by \( N \)) by input matrix \( B \) (\( N \) by \( P \)) using 2 nested loops:

```
for i = 1 to M
  for k = 1 to P
    { temp = 0
    for j = 1 to N
      temp = temp + A(i,j) \cdot B(j,k)
    C(i,k) = temp
  }
```

**Overflow Handling Methodology** Not applicable

**Special Requirements** Verify that the dimensions of input matrices are legal, i.e. \( \text{col1} = \text{row2} \)

**Implementation Notes** In order to take advantage of the dual MAC architecture of the C55x, this implementation checks the size of the matrix \( x1 \). For small matrices \( x1 \) (\( \text{row1} < 4 \) or \( \text{col1} < 2 \)), single MAC loops are used. For larger matrices \( x1 \) (\( \text{row1} \geq 4 \) and \( \text{col1} \geq 2 \)), Dual MAC loops are more efficient and quickly make up for the additional initialization overhead.
**mmul  Matrix Multiplication**

**Example**  
See examples/mmul subdirectory

**Benchmarks (preliminary)**

<table>
<thead>
<tr>
<th>Cycles†</th>
<th>Core:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>if(row1 &lt; 4</td>
</tr>
<tr>
<td></td>
<td>((col1 + 2)*row1 + 4)*col2</td>
</tr>
<tr>
<td></td>
<td>if((row1==even)&amp;&amp;(row1 ≥ 4)&amp;&amp;(col1 ≥ 2)), use dual MAC</td>
</tr>
<tr>
<td></td>
<td>((col1 + 4)<em>0.5</em>row1 + 10)col2</td>
</tr>
<tr>
<td></td>
<td>if((row1==odd)&amp;&amp;(row1 ≥ 4)&amp;&amp;(col1 ≥ 2), use dual MAC</td>
</tr>
<tr>
<td></td>
<td>((col1 + 4)<em>0.5</em>(row1 – 1) + col1 + 12)col2</td>
</tr>
<tr>
<td>Overhead:</td>
<td>30</td>
</tr>
</tbody>
</table>

| Code size (in bytes) | 215 |

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
Matrix Transpose

mtrans

ushort oflag = mtrans (DATA *x, short row, short col, DATA *r)

(defined in mtrans.asm)

Arguments

x[row*col]  Pointer to input matrix. In-place processing is not allowed.
row  number of rows in matrix
col  number of columns in matrix
r[row*col]  Pointer to output data vector

Description

This function transposes matrix x

Algorithm

for i = 1 to M
for j = 1 to N
   C(j,i) = A(i,j)

Overflow Handling Methodology

Not applicable

Special Requirements

None

Implementation Notes

None

Example

See examples/mtrans subdirectory

Benchmarks (preliminary)

Cycles†  Core:  (1 + col) * row
Overhead:  23

Code size  65
(in bytes)

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
mul32  32-bit Vector Multiplication

mul32  32-bit Vector Multiplication

ushort oflag = mul32 (LDATA *x, LDATA *y, LDATA *r, ushort nx)
(defined in mul32.asm)

Arguments

x[nx]     Pointer to input data vector 1 of size nx. In-place processing allowed (r can be = x = y)
y[nx]     Pointer to input data vector 2 of size nx
r[nx]     Pointer to output data vector of size nx containing
nx        Number of elements of input and output vectors. Nx ≥ 4
oflag     Overflow flag
         □ If oflag = 1, a 32-bit overflow has occurred
         □ If oflag = 0, a 32-bit overflow has not occurred

Description

This function multiplies two 32-bit Q31 vectors, element by element, and produces a 32-bit Q31 vector.

Algorithm

for (i=0; i < nx; i++)
z (i) = x (i) * y (i)

Overflow Handling Methodology

Scaling implemented for overflow prevention (user selectable)

Special Requirements

None

Implementation Notes

None

Example

See examples/add subdirectory

Benchmarks

<table>
<thead>
<tr>
<th>Cycles</th>
<th>Core:</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4*nx + 4</td>
<td>21</td>
</tr>
</tbody>
</table>

Code size 73
(in bytes)
Vector Negate

`neg`

```c
ushort oflag = neg (DATA *x, DATA *r, ushort nx)
```

(defined in neg.asm)

Arguments

- **x[nx]** Pointer to input data vector 1 of size nx. In-place processing allowed (r can be = x = y)
- **r[nx]** Pointer to output data vector of size nx. In-place processing allowed
  - Special cases:
    - If `x[I] = -1 = 32768`, then `r = 1 = 321767` with `oflag = 1`
    - If `x= 1 = 32767`, then `r = –1 = 321768` with `oflag = 1`
- **nx** Number of elements of input and output vectors.
  - `nx ≥ 4`
- **oflag** Overflow flag.
  - If `oflag = 1`, a 32-bit overflow has occurred.
  - If `oflag = 0`, a 32-bit overflow has not occurred.
  - Caution: overflow in negation of a Q15 number can happen naturally when negating (–1).

Description

This function negates each of the elements of a vector (fractional values).

Algorithm

```c
for (i = 0; i < nx; i ++ )  x(i) = –x(i)
```

Overflow Handling Methodology

Saturation implemented for overflow handling

Special Requirements

None

Implementation Notes

None

Example

See examples/neg subdirectory

Benchmarks (preliminary)

- **Cycles†** Core: 4 * nx
- **Overhead:** 13
- **Code size** (in bytes) 61

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
### neg32 Vector Negate (double precision)

**ushort oflag = neg32 (LDATA *x, LDATA *r, ushort nx)**

(defined in neg.asm)

**Arguments**

- **x[nx]** Pointer to input data vector 1 of size nx. In-place processing allowed (r can be = x = y)
- **r[nx]** Pointer to output data vector of size nx. In-place processing allowed
  - Special cases:
    - if \( x = -1 = 32768 \times 2^{16} \), then \( r = 1 = 321767 \times 2^{16} \) with oflag = 1
    - if \( x = 1 = 32767 \times 2^{16} \), then \( r = -1 = 321768 \times 2^{16} \) with oflag = 1
- **nx** Number of elements of input and output vectors.
  - \( nx \geq 4 \)
- **oflag** Overflow flag.
  - If oflag = 1, a 32-bit overflow has occurred.
  - If oflag = 0, a 32-bit overflow has not occurred.
  - Caution: overflow in negation of a Q31 number can happen naturally when negating \((-1)\).

**Description**

This function negates each of the elements of a vector (fractional values).

**Algorithm**

\[
for (i = 0; i < nx; i + +) \quad x(i) = -x(i)
\]

**Overflow Handling Methodology** Saturation implemented for overflow handling

**Special Requirements** None

**Implementation Notes** None

**Example**

See examples/neg32 subdirectory

**Benchmarks (preliminary)**

- **Cycles**
  - Core: \( 4 \times nx \)
  - Overhead: 13

- **Code size**
  - 61

\( \dagger \) Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
**power**

```c
ushort oflag = power (DATA *x, LDATA *r, ushort nx)
```

(defined in power.asm)

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[nx]</td>
<td>Pointer to input data vector 1 of size nx. In-place processing allowed (r can be x = y)</td>
</tr>
<tr>
<td>r[1]</td>
<td>Pointer to output data vector element in Q31 format</td>
</tr>
<tr>
<td>nx</td>
<td>Number of elements of input vectors. nx ≥ 4</td>
</tr>
<tr>
<td>oflag</td>
<td>Overflow flag.</td>
</tr>
</tbody>
</table>

**Special cases:**

- if x = -1 = 32768*2^16, then r = 1 = 321767*2^16 with oflag = 1
- if x = 1 = 32767*2^16, then r = -1 = 321768*2^16 with oflag = 1

**Description**

This function calculates the power (sum of products) of a vector.

**Algorithm**

Power = 0 for (i = 0; i < nx; i++)  power += x(i) * x(i)

**Overflow Handling Methodology**

No scaling implemented for overflow handling

**Special Requirements**

None

**Implementation Notes**

None

**Example**

See examples/power subdirectory

**Benchmarks (preliminary)**

<table>
<thead>
<tr>
<th>Category</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles†</td>
<td>Core: nx – 1</td>
</tr>
<tr>
<td></td>
<td>Overhead: 12</td>
</tr>
<tr>
<td>Code size</td>
<td>54 (in bytes)</td>
</tr>
</tbody>
</table>

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
**q15tofl**  
*Q15 to Floating-point Conversion*

```c
ushort q15tofl (DATA *x, float *r, ushort nx)
(defined in q152fl.asm)
```

**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 Q15 stored in vector `x` to IEEE floating-point numbers stored in vector `r`.

**Algorithm**

Not applicable

**Overflow Handling Methodology** Saturation implemented for overflow handling

**Special Requirements** None

**Implementation Notes** None

**Example**

See examples/ug subdirectory

**Benchmarks (preliminary)**

- **Cycles†** Core:  
  - 7 * `nx` (if `x[n] == 0`)
  - 29 * `nx` (otherwise)

- **Overhead:** 19

- **Code size** (in bytes) 124

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
Random Number Generation Algorithm

**rand16**

```c
ushort oflag = rand16 (DATA *r, ushort nr)
```

**Arguments**

- **r** Pointer to the array where the 16-bit random numbers are stored
- **nr** Number of random numbers that are generated
- **oflag** Overflow error flag (returned value)
  - If oflag = 1, a 32-bit data overflow occurred in an intermediate or final result.
  - If oflag = 0, a 32-bit overflow has not occurred.

**Description**

This algorithm computes an array of random numbers based on the linear congruential method introduced by D. Lehmer in 1951. This is one of the fastest and simplest techniques of generating random numbers. The code shown here generates 16-bit integers, however, if a 32-bit value is desired the code can be modified to perform 32-bit multiplies using the defined constants RNDMULT and RNDINC. The disadvantage of this technique is that it is very sensitive to the choice of RNDMULT and RNDINC.

**Algorithm**

\[ r[n] = \left\{ (r[n-1]*\text{RNDMULT}) + \text{RNDINC} \right\} \mod M \]

where \( 0 \leq n \leq \text{nr} \) and \( 0 \leq M \leq 65536 \)

**Overflow Handling Methodology**

No scaling implemented for overflow prevention.

**Special Requirements**

No special requirements.

**Implementation Notes**

Rand16() is written so that it can be called from a C program. Prior to calling rand16(), rand16i() can be called to initialize the random number generator seed value. The C routine passes two parameters to rand16(): A pointer to the random number array *r and the count of random numbers (nr) desired. The random numbers are declared as short or 16 bit values. Two constants RNDMULT and RNDINC are defined in the function. The algorithm is sensitive to the choice of RNDMULT and RNDINC so exercise caution when changing these.
M This value is based on the system that the routine runs. This routine returns a random number from 0 to 65536 (64K) and is NOT internally bounded. If you need a min/max limit, this must be coded externally to this routine.

RNDSEED An arbitrary constant that can be any value between 0 and 64K. If 0 (zero) is chosen, then RNDINC should be some value greater than 1. Otherwise, the first two values will be 0 and 1. To change the set of random numbers generated by this routine, change the RNDSEED value. In this routine, RNDSEED is initialized to 21845, which is 65536/3.

RNDMULT Should be chosen such that the last three digits fall in the pattern even_digit–2–1 such as xx821, xx421 etc. RNDMULT = 31821 is used in this routine.

RNDINC In general, this constant can be any prime number related to M. Research shows that RNDINC (the increment value) should be chosen by the following formula: RNDINC = ((1/2 − (1/6 * SQRT(3))) * M). Using M=65536, RNDINC was picked as 13849.

The random seed initialized in rand16i() is used to generate the first random number. Each random number generated is used to generate the next number in the series. The random number is generated in the accumulator (32 bits) by using the multiply-accumulate (MAC) unit to do the computation. In the course of the algorithm if there is intermediate overflow, the overflow flag bit in status register is set. At the end of the algorithm, the overflow flag is tested for any intermediate overflow conditions.

Example
See examples/rand16 subdirectory

Benchmarks

<table>
<thead>
<tr>
<th></th>
<th>Core: 13 + nr*2</th>
<th>Overhead: 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code size</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>(in bytes)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

C54x Benchmark for Comparison

<table>
<thead>
<tr>
<th></th>
<th>Core: 10 + nr*4</th>
<th>Overhead: 16</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code size</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>(in bytes)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
void rand16init(void)

Arguments
None

Description
Initializes seed for 16-bit random number generator.

Algorithm
Not applicable

Overflow Handling Methodology
No scaling implemented for overflow prevention.

Special Requirements
Allocation of .bss section is required in linker command file.

Implementation Notes
This function initializes a global variable rndseed in global memory to be used for the 16 bit random number generation routine (rand16)

Example
See examples/rand16i subdirectory

Benchmarks

<table>
<thead>
<tr>
<th>Cycles</th>
<th>Code size (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>9</td>
</tr>
</tbody>
</table>

C54x Benchmark for Comparison

<table>
<thead>
<tr>
<th>Cycles</th>
<th>Code size (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>10</td>
</tr>
</tbody>
</table>
void recip16 (DATA *x, DATA *r, DATA *rexp, ushort nx)

Arguments

x[nx] Pointer to input data vector 1 of size nx.
    x[0]
    x[1]
    ...
    x[nx–2]
    x[nx–1]

r[nx] Pointer to output data buffer
    r[0]
    r[1]
    ...
    r[nx–2]
    r[nx–1]

rexp[nx] Pointer to exponent buffer for output values. These exponent values are in integer format.
    rexp[0]
    rexp[1]
    ...
    rexp[nx–2]
    rexp[nx–1]

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:

\[ r[i] \times \text{rexp}[i] = \text{true reciprocal in floating-point} \]

Algorithm

\[ Ym = 2 \times Ym - Ym^2 \times Xnorm \]

If we start with an initial estimate of Ym, the equation converges to a solution very rapidly (typically 3 iterations for 16-bit resolution).
The initial estimate can be obtained from a look-up table, from choosing a mid-point, or simply from linear interpolation. The method chosen for this problem is linear interpolation and is accomplished by taking the complement of the least significant bits of the Xnorm value.

**Overflow Handling Methodology** None

**Special Requirements** None

**Implementation Notes** None

**Example** See examples/recip16 subdirectory

**Benchmarks (preliminary)**

<table>
<thead>
<tr>
<th>Cycles†</th>
<th>Core: 33 * nx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead: 12</td>
<td></td>
</tr>
</tbody>
</table>

| Code size (in bytes) | 69 |

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
**rfft**  \textit{Forward Real FFT (in-place)}

void rfft (DATA *x, ushort nx, ushort scale);

(reference cfft.asm and unpack.asm)

**Arguments**

x [nx] Pointer to input vector containing nx real elements. On output, vector x contains the first half (nx/2 complex elements) of the FFT output in the following order. Real FFT is a symmetric function around the Nyquist point, and for this reason only half of the FFT(x) elements are required.

On output x will contain the FFT(x) = y in the following format:
- y(0)Re y(nx/2)Im → DC and Nyquist
- y(1)Re y(1)Im
- y(2)Re y(2)Im
- ....
- y(nx/2)Re y(nx/2)Im

Complex numbers are stored in Re-Im format

nx Number of real elements in vector x. can take the following values.

nx   = 16, 32, 64, 128, 256, 512, 1024

scale Flag to indicate whether or not scaling should be implemented during computation.

if (scale == 0)
  scale factor = nx;
else
  scale factor = 1;
end

**Description**

Computes a Radix-2 real DIT FFT of the nx real elements stored in vector x in bit-reversed order. The original content of vector x is destroyed in the process. The first nx/2 complex elements of the FFT(x) are stored in vector x in normal-order.

**Algorithm**

(DFT)

See CFFT
Special Requirements

- All above mentioned functions are required for the FFT
- unpack.asm (containing code to for unpacking results)

Implementation Notes

Implemented as a complex FFT of size \( \text{nx}/2 \) followed by an unpack stage to unpack the real FFT results. Therefore, Implementation Notes for the cfft function apply to this case.

Notice that normally an FFT of a real sequence of size \( N \), produces a complex sequence of size \( N \) (or \( 2 \times N \) real numbers) that will not fit in the input sequence. To accommodate all the results without requiring extra memory locations, the output reflects only half of the spectrum (complex output). This still provides the full information because an FFT of a real sequence has even symmetry around the center or nyquist point \( (N/2) \).

When scale = 1, this routine prevents overflow by scaling by 2 at each FFT intermediate stages and at the unpacking stage.

Example

See examples/rfft Zip file
rifft  Inverse Real FFT (in-place)

void rifft (DATA *x, ushort nx, ushort scale);

(reference cifft.asm)

Arguments

x [nx] Pointer to input vector x containing nx real elements. The unpacki routine should be called to unpack the rfft sequence before calling the bit reversal routine. (See examples directory for calling sequence)

On output, the vector x contains nx complex elements corresponding to RIFFT(x) or the signal itself.

nx Number of real elements in vector x. nx can take the following values.

nx = 16, 32, 64, 128, 256, 512, 1024

scale Flag to indicate whether or not scaling should be implemented during computation.

If (scale == 0)

    scale factor = nx;

else

    scale factor = 1;

end

Description Computes a Radix-2 real DIT IFFT of the nx real elements stored in vector x in bit-reversed order. The original content of vector x is destroyed in the process. The first nx/2 complex elements of the IFFT(x) are stored in vector x in normal-order.

Algorithm (IDFT)

See CIFFT

Special Requirements This function should work with unpacki.asm and/or cbrev.asm for proper result. See example in examples/rifft directory.
Implementation Notes Implemented as a complex IFFT of size nx/2 followed by an unpack stage to unpack the real IFFT results. Therefore, Implementation Notes for the cfft function apply to this case.

Notice that normally an IFFT of a real sequence of size N, produces a complex sequence of size N (or 2*N real numbers) that will not fit in the input sequence. To accommodate all the results without requiring extra memory locations, the output reflects only half of the spectrum (complex output). This still provides the full information because an IFFT of a real sequence has even symmetry around the center or nyquist point(N/2).

When scale = 1, this routine prevents overflow by scaling by 2 at each IFFT intermediate stages and at the unpacking stage.

Example See examples/rifft subdirectory
sine  Sine

Arguments

\[ x[nx] \quad \text{Pointer to input vector of size} \ nx. \ x \text{ contains the angle in} \ \text{radians between} \ [-\pi, \ \pi] \ \text{normalized between} \ (-1,1) \ \text{in} \ q15 \ \text{format} \]
\[ x = \text{xrad}/\pi \]
\[ \text{For example:} \]
\[ 45^\circ = \pi/4 \text{ is equivalent to} \ x = 1/4 = 0.25 = 0x200 \ \text{in} \ q15 \ \text{format}. \]
\[ r[nx] \quad \text{Pointer to output vector containing the sine of vector} \ x \ \text{in} \ \text{q15 format} \]
\[ nx \quad \text{Number of elements of input and output vectors.} \]
\[ nx \geq 4 \]
\[ oflag \quad \text{Overflow flag.} \]
\[ \square \quad \text{If} \ oflag = 1, \ \text{a} \ 32\text{-bit overflow has occurred.} \]
\[ \square \quad \text{If} \ oflag = 0, \ \text{a} \ 32\text{-bit overflow has not occurred.} \]

Description

Computes the sine of elements of vector \( x \). It uses Taylor series to compute the sine of angle \( x \).

Algorithm

\[ \text{for} \ (i = 0; \ i < nx; \ i++) \quad y(i) = \sin(x(i)) \ \text{where} \ x(i) - \frac{xrad}{\pi} \]

Overflow Handling Methodology

Not applicable

Special Requirements

None

Implementation Notes

Computes the sine of elements of vector \( x \). It uses the following Taylor series to compute the angle \( x \) in quadrant 1 \((0–\pi/2)\).

\[ \sin(x) = c1*x + c2*x^2 + c3*x^3 + c4*x^4 + c5*x^5 \]

\[ c1 = 3.140625x \]
\[ c2 = 0.02026367 \]
\[ c3 = -5.3251 \]
\[ c4 = 0.5446778 \]
\[ c5 = 1.800293 \]

The angle \( x \) in other quadrant is calculated by using symmetries that map the angle \( x \) into quadrant 1.
Example
See examples/sine subdirectory

Benchmarks (preliminary)

<table>
<thead>
<tr>
<th>Cycles†</th>
<th>Core: 19 * nx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead: 17</td>
<td></td>
</tr>
</tbody>
</table>

Code size
93 program; 3 data
(in bytes)

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
**sqrt_16**  
*Square Root of a 16-bit Number*

`ushort oflag = sqrt_16 (DATA *x, DATA *r, short nx)`  
(defined in sqrtv.asm)

**Arguments**

- `x[nx]` Pointer to input vector of size `nx`.  
- `r[nx]` Pointer to output vector of size `nx` containing the `sqrt(x)`.  
- `nx` Number of elements of input and output vectors.  
- `oflag` Overflow flag.  
  - If `oflag = 1`, a 32-bit overflow has occurred.  
  - If `oflag = 0`, a 32-bit overflow has not occurred.

**Description**  
Calculates the square root for each element in input vector `x`, storing results in output vector `r`.

**Algorithm**  
for `(i = 0; i < nx; i++)`  
\[ r[i] = \sqrt{x(i)} \text{ where } 0 \leq i \leq nx \]

**Overflow Handling Methodology**  
Not applicable

**Special Requirements**  
None

**Implementation Notes**  
The square root of a number(`x`) can be calculated using Newton's method. An initial approximation is guessed and then the approximation gets recomputed using the formula,

\[ \text{new approximation} = \text{old approximation} - \frac{(\text{old approximation}^2 - x)}{2} . \]

The new approximation then becomes the old approximation and the process is repeated until the desired accuracy is reached.

**Example**  
See examples/sqrtv subdirectory

**Benchmarks (preliminary)**

- Cycles\(^\dagger\)  
  - Core: 35 * `nx`  
  - Overhead: 14

- Code size (in bytes)  
  - 84 program; 5 data

\(^\dagger\) Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
**Vector Subtract**

short oflag = sub (DATA *x, DATA *y, DATA *r, ushort nx, ushort scale)
(defined in sub.asm)

**Arguments**

- **x[nx]** Pointer to input data vector 1 of size nx. In-place processing allowed (r can be = x = y)
- **y[nx]** Pointer to input data vector 2 of size nx
- **r[nx]** Pointer to output data vector of size nx containing
  - (x–y) if scale =0
  - (x–y)/2 if scale =1
- **nx** Number of elements of input and output vectors. nx ≥ 4
- **scale** Scale selection
  - If scale = 1, divide the result by 2 to prevent overflow.
  - If scale = 0, do not divide by 2.
- **oflag** Overflow flag.
  - If oflag = 1, a 32-bit overflow has occurred.
  - If oflag = 0, a 32-bit overflow has not occurred.

**Description**

This function subtracts two vectors, element by element.

**Algorithm**

\[
\text{for (} i = 0; i < nx; i + + \text{) } z(i) = x(i) - y(i)
\]

**Overflow Handling Methodology** Scaling implemented for overflow prevention (User selectable)

**Special Requirements** None

**Implementation Notes** None

**Example** See examples/sub subdirectory

**Benchmarks (preliminary)**

- **Cycles†** Core: 3 * nx  
  Overhead: 23

- **Code size** 60  
  (in bytes)

† Assumes all data is in on-chip dual-access RAM and that there is no bus conflict due to twiddle table reads and instruction fetches (provided linker command file reflects those conditions).
All functions in the DSPLIB are provided with execution time and code size benchmarks. While developing the included functions, we tried to compromise between speed, code size, and ease of use. However, with few exceptions, the highest priority was given to optimize for speed and ease of use, and last for code size.

Even though DSPLIB can be used as a first estimation of processor performance for a specific function, you should know that the generic nature of DSPLIB may add extra cycles not required for customer specific usage.
5.1 What DSPLIB Benchmarks are Provided

DSPLIB documentation includes benchmarks for instruction cycles and memory consumption. The following benchmarks are typically included:

- Calling and register initialization overhead
- Number of cycles in the kernel code: Typically provided in the form of an equation that is a function of the data size parameters. We consider the kernel (or core) code, the instructions contained between the _start and _end labels that you can see in each of the functions.
- Memory consumption: Typically program size in bytes is reported. For functions requiring significant internal data allocation, data memory consumption is also provided. When stack usage for local variables is minimum, that data consumption is not reported.

For functions in which it is difficult to determine the number of cycles in the kernel code as a function of the data size parameters, we have included direct cycle count for specific data sizes.

5.2 Performance Considerations

Benchmark cycles presented assume best case conditions, typically assuming:

- 0 wait-state memory external memory for program and data
- data allocation to on-chip DARAM
- no pipeline hits

A linker command file showing the memory allocation used during testing and benchmarking in the Code Composer C55x Simulator is included under the example subdirectory.

Remember, execution speed in a system is dependent on where the different sections of program and data are located in memory. Be sure to account for such differences, when trying to explain why a routine is taking more time that the reported DSPLIB benchmarks.
Chapter 6

Licensing, Warranty, and Support

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Licensing and Warranty</td>
<td>6-2</td>
</tr>
<tr>
<td>6.2 DSPLIB Software Updates</td>
<td>6-2</td>
</tr>
<tr>
<td>6.3 DSPLIB Customer Support</td>
<td>6-2</td>
</tr>
</tbody>
</table>
6.1 Licensing and Warranty

C55x DSPLIB is distributed as a free-of-charge product under the generic Texas Instrument License Form presented in Appendix C.

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 will typically follow after a month of the Beta release but no explicit guarantees are paced on that date.

6.2 DSPLIB Software Updates

C55x DSPLIB Software updates will be periodically released, incorporating product enhancement and fixes.

DSPLIB Software Updates will be posted as they become available in the same location you download this information. Source Code for previous releases will be kept public to prevent any customer problem in case we decide to discontinue or change the functionality of one of the DSPLIB functions. Make sure to read the readme.1st file available in the root directory of every release.

6.3 DSPLIB Customer Support

If you have any questions or want to report problems or suggestions regarding the C55x DSPLIB, contact Texas Instruments at dpsh@ti.com.

We encourage the use of the software report form (report.txt) contained in the DSPLIB root directory to report any problem associated with the C55x DSPLIB.
Overview of Fractional Q Formats

Unless specifically noted, DSPLIB functions use Q15 format or to be more exact Q0.15. In a Q\textit{m.n} format, there are \textit{m} bits used to represent the two’s complement integer portion of the number, and \textit{n} bits used to represent the two’s complement fractional portion. \textit{m+n+1} bits are needed to store a general Q\textit{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 1. 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.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 Q3.12 Format</td>
<td>A-2</td>
</tr>
<tr>
<td>A.2 Q.15 Format</td>
<td>A-2</td>
</tr>
<tr>
<td>A.3 Q.31 Format</td>
<td>A-2</td>
</tr>
</tbody>
</table>
A.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. Q.3.12 Bit Fields

<table>
<thead>
<tr>
<th>Bit</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>…</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>S</td>
<td>I3</td>
<td>I2</td>
<td>I1</td>
<td>Q11</td>
<td>Q10</td>
<td>Q9</td>
<td>…</td>
<td>Q0</td>
</tr>
</tbody>
</table>

A.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

<table>
<thead>
<tr>
<th>Bit</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>…</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>S</td>
<td>Q14</td>
<td>Q13</td>
<td>Q12</td>
<td>Q11</td>
<td>Q10</td>
<td>Q9</td>
<td>…</td>
<td>Q0</td>
</tr>
</tbody>
</table>

A.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

<table>
<thead>
<tr>
<th>Bit</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>…</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>Q15</td>
<td>Q14</td>
<td>Q13</td>
<td>Q12</td>
<td>…</td>
<td>Q3</td>
<td>Q2</td>
<td>Q1</td>
<td>Q0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Bit</th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>…</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>S</td>
<td>Q30</td>
<td>Q29</td>
<td>Q28</td>
<td>…</td>
<td>Q19</td>
<td>Q18</td>
<td>Q17</td>
<td>Q16</td>
</tr>
</tbody>
</table>
Calculating the Reciprocal of a Q15 Number

The most optimal method for calculating the inverse of a fractional number (Y=1/X) is to normalize the number first. This limits the range of the number as follows:

0.5 ≤ Xnorm < 1
-1 ≤ Xnorm ≤ -0.5

The resulting equation becomes

\[ Y = \frac{1}{(X_{\text{norm}} + 2^{-n})} \]

or

\[ Y = \frac{2^n}{X_{\text{norm}}} \]  

where \( n = 1, 2, 3, \ldots, 14, 15 \)

Letting \( Y_e = 2^n \):

\[ Y_e = 2^n \]  

Substituting (3) into equation (2):

\[ Y = Y_e \times \frac{1}{X_{\text{norm}}} \]  

Letting \( Y_m = \frac{1}{X_{\text{norm}}} \):

\[ Y_m = \frac{1}{X_{\text{norm}}} \]  

Substituting (5) into equation (4):

\[ Y = Y_e \times Y_m \]  

For the given range of Xnorm, the range of Ym is:

1 ≤ Ym < 2
-2 ≤ Ym ≤ -1

To calculate the value of Ym, various options are possible:

a) Taylor Series Expansion
b) 2nd, 3rd, 4th... Order Polynomial (Line Of Best Fit)
c) Successive Approximation
The method chosen in this example is (c). Successive approximation yields the most optimum code versus speed versus accuracy option. The method outlined below yields an accuracy of 15 bits.

Assume $Y_m(\text{new})$ = exact value of $\frac{1}{X_{\text{norm}}}$.

\[ Y_m(\text{new}) = \frac{1}{X_{\text{norm}}} \]  

\[ \text{or} \]

\[ Y_m(\text{new}) \times X = 1 \]  

Assume $Y_m(\text{old}) = \text{estimate of value} \frac{1}{X}$.

\[ Y_m(\text{old}) \times X_{\text{norm}} = 1 + D_{xy} \]

\[ \text{or} \]

\[ D_{xy} = Y_m(\text{old}) \times X_{\text{norm}} - 1 \]  

where $D_{xy}$ = error in calculation

Assume that $Y_m(\text{new})$ and $Y_m(\text{old})$ are related as follows:

\[ Y_m(\text{new}) = Y_m(\text{old}) - D_y \]

where $D_y$ = difference in values

Substituting (c2) and (c4) into (c3):

\[ Y_m(\text{old}) \times X_{\text{norm}} = Y_m(\text{new}) \times X_{\text{norm}} + D_{xy} \]

\[ (Y_m(\text{new}) + D_y) \times X_{\text{norm}} = Y_m(\text{new}) \times X_{\text{norm}} + D_{xy} \]

\[ Y_m(\text{new}) \times X_{\text{norm}} + D_y \times X_{\text{norm}} = Y_m(\text{new}) \times X_{\text{norm}} + D_{xy} \]

\[ D_y \times X_{\text{norm}} = D_{xy} \]

\[ D_y = D_{xy} \times \frac{1}{X_{\text{norm}}} \]  

Assume that $1/X_{\text{norm}}$ is approximately equal to $Y_m(\text{old})$:

\[ D_y = D_{xy} \times Y_m(\text{old}) \text{ (approx)} \]

Substituting (c6) into (c4):

\[ Y_m(\text{new}) = Y_m(\text{old}) - D_{xy} \times Y_m(\text{old}) \]

Substituting for $D_{xy}$ from (c3) into (c7):

\[ Y_m(\text{new}) = Y_m(\text{old}) - (Y_m(\text{old}) \times X_{\text{norm}} - 1) \times Y_m(\text{old}) \]

\[ Y_m(\text{new}) = Y_m(\text{old}) - Y_m(\text{old})^2 \times X_{\text{norm}} + Y_m(\text{old}) \]

\[ Y_m(\text{new}) = 2 \times Y_m(\text{old}) - Y_m(\text{old})^2 \times X_{\text{norm}} \]
If after each calculation we equate $Y_m(\text{old})$ to $Y_m(\text{new})$:

$$Y_m(\text{old}) = Y_m(\text{new}) = Y_m$$

Then equation (c8) evaluates to:

$$Y_m = 2 \cdot Y_m - Y_m^2 \cdot X_{\text{norm}}$$  \hspace{1cm} (c9)

If we start with an initial estimate of $Y_m$, then equation (c9) converges to a solution very rapidly (typically 3 iterations for 16-bit resolution).

The initial estimate can be obtained from a look-up table, from choosing a mid-point, or simply from linear interpolation. The method chosen for this problem is linear interpolation and accomplished by taking the complement of the least significant bits of the $X_{\text{norm}}$ value.
Texas Instruments License Agreement for DSP Code

Texas Instruments License Agreement for DSP Code IF YOU DOWNLOAD OR USE THIS PROGRAM YOU AGREE TO THESE TERMS.

Texas Instruments Incorporated grants you a license to use the Program only in the country where you acquired it. The Program is copyrighted and licensed (not sold). We do not transfer title to the Program to you. You obtain no rights other than those granted you under this license.

Under this license, you may:

1) use the Program on one or more machines at a time;
2) make copies of the Program for use or backup purposes within your Enterprise;
3) modify the Program and merge it into another program; and
4) make copies of the original file you downloaded and distribute it, provided that you transfer a copy of this license to the other party. The other party agrees to these terms by its first use of the Program.

You must reproduce the copyright notice and any other legend of ownership on each copy or partial copy, of the Program.

You may NOT:

1) sublicense, rent, lease, or assign the Program; and
2) reverse assemble, reverse compile, or otherwise translate the Program.
3) Use it in non-TI DSPs

We do not warrant that the Program is free from claims by a third party of copyright, patent, trademark, trade secret, or any other intellectual property infringement.

Under no circumstances are we liable for any of the following:

1) third-party claims against you for losses or damages;
2) loss of, or damage to, your records or data; or
3) economic consequential damages (including lost profits or savings) or incidental damages, even if we are informed of their possibility.
Some jurisdictions do not allow these limitations or exclusions, so they may not apply to you.

We do not warrant uninterrupted or error free operation of the Program. We have no obligation to provide service, defect correction, or any maintenance for the Program. We have no obligation to supply any Program updates or enhancements to you even if such are or later become available.

IF YOU DOWNLOAD OR USE THIS PROGRAM YOU AGREE TO THESE TERMS.

THERE ARE NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Some jurisdictions do not allow the exclusion of implied warranties, so the above exclusion may not apply to you.

You may terminate this license at any time. We may terminate this license if you fail to comply with any of its terms. In either event, you must destroy all your copies of the Program.

You are responsible for the payment of any taxes resulting from this license.

You may not sell, transfer, assign, or subcontract any of your rights or obligations under this license. Any attempt to do so is void.

Neither of us may bring a legal action more than two years after the cause of action arose.

This license is governed by the laws of the State of Texas.
<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit reciprocal function</td>
</tr>
<tr>
<td>32-bit by 16-bit long division function</td>
</tr>
<tr>
<td>32-bit vector multiplication</td>
</tr>
</tbody>
</table>

**A**
- acorr | 4-7 |
- adaptive delayed LMS filter | 4-34 |
- add | 4-10 |
- arctangent 2 implementation | 4-11 |
- arctangent implementation | 4-13 |
- atan16 | 4-13 |
- atan2_16 | 4-11 |
- autocorrelation | 4-7 |

**B**
- base 10 logarithm | 4-77 |
- base 2 logarithm | 4-79 |
- base e logarithm | 4-81 |
- bexp | 4-15 |
- block exponent implementation | 4-15 |
- block FIR filter (fast) | 4-43 |

**C**
- cascaded IIR direct form I | 4-71 |
- cascaded IIR direct form II | 4-67, 4-69 |
- cbrev | 4-16 |
- cfft | 4-17 |
- cifr | 4-19 |
- cift | 4-23 |
- complex bit reverse | 4-16 |
- complex FIR filter | 4-19 |

**D**
- conversion
  - floating-point to Q15 (fltoq15) | 4-58 |
  - Q15 to floating-point (q15tolf) | 4-96 |
- convol | 4-25 |
- convol1 | 4-27 |
- convol2 | 4-29 |
- convolution | 4-25 |
- convolution (fast) | 4-27 |
- convolution (fastest) | 4-29 |
- corr | 4-32 |
- correlation
  - auto (acorr) | 4-7 |
  - full-length (corr) | 4-32 |
  - correlation (full-length) | 4-32 |

**E**
- expn | 4-37 |
- exponential base e | 4-37 |
Index

F

FFT
  forward complex (cfft) 4-17
  forward real in-place (rfft) 4-102
  inverse complex (cifft) 4-23
  inverse real in-place (rifft) 4-104
fir 4-39
FIR filter 4-39
  block filter fast (fir2) 4-43
  complex (cfir) 4-19
  decimating (firdec) 4-47
  direct form (fir) 4-39
  Hilbert Transformer 4-60
  interpolating (firinterp) 4-49
  lattice forward (firlat) 4-51
  symmetric (firs) 4-54
FIR Hilbert Transformer 4-60
fir2 4-43
firdec 4-47
firinterp 4-49
firlat 4-51
firs 4-54
  floating-point to Q15 conversion 4-58
fltoq15 4-58
  forward complex FFT 4-17
  forward real FFT (in-place) 4-102

H

hilb16 4-60

I

IIR filter
  cascaded direct form I (iircas51) 4-71
  cascaded direct form II (iircas4) 4-67
  cascaded direct form II (iircas5) 4-69
  double-precision (iir32) 4-64
  lattice inverse (iirlat) 4-73
iir32 4-64
iircas4 4-67
iircas5 4-69
iircas51 4-71
iirlat 4-73

index and value of maximum element of a vector 4-85
index and value of minimum element of a vector 4-88
index of maximum element of a vector 4-83
index of minimum element of a vector 4-86
interpolating FIR filter 4-49
inverse complex FFT 4-23
inverse real FFT (in-place) 4-104

L

lattice forward (FIR) filter 4-51
lattice inverse (IIR) filter 4-73
ldiv16 4-75
log_10 4-77
log_2 4-79
logarithm
  base 10 (log_10) 4-77
  base 2 (log_2) 4-79
  base e (logn) 4-81
logn 4-81

M

matrix multiplication 4-89
matrix transpose 4-91
maxidx 4-83
maximum element of a vector
  index (maxidx) 4-83
  index and value (maxvec) 4-85
maximum value of a vector 4-84
maxval 4-84
maxvec 4-85
minidx 4-86
minimum element of a vector
  index (minidx) 4-86
  index and value (minvec) 4-88
minimum value of a vector 4-87
minval 4-87
minvec 4-88
mul 4-92
mul32 4-92
Index

N
natural logarithm (logn) 4-81
neg 4-93
neg32 4-94

P
power 4-95

Q
Q15 to floating-point conversion 4-96
q15tofl 4-96

R
rand16 4-97
rand16init 4-99
random number generation
  algorithm 4-97
  initialization 4-99

recip16 4-100
rfft 4-102
rifft 4-104

S
sine 4-106
sqrt_16 4-108
square root of a 16-bit number 4-108
sub 4-109
symmetric FIR filter 4-54

V
vector add 4-10
vector negate 4-93
vector negate, double-precision 4-94
vector power 4-95
vector subtract 4-109