Texas Instruments

Application Report
SPRA480B – October 2000

Optimized DSP Library for C Programmers on the TMS320C54x

C5000 DSP Software Application Group — Texas Instruments Incorporated

ABSTRACT

The TMS320C54x™ DSPLIB is an optimized DSP Function Library for C programmers on TMS320C54x 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 to providing ready-to-use DSP functions, TI DSPLIB can significantly shorten your DSP application development time.

The TI DSPLIB includes commonly used DSP routines. Source code is provided 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:

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

Contents

1 Introduction

1.1 Features and Benefits

- Hand-coded assembly optimized routines
- C-callable routines fully compatible with the TI ’C54x compiler
- Support also provided for ’C54x devices with extended program memory addressing (Far mode)
- Fractional Q15-format operand supported
- Complete set of examples on use provided
- Benchmarks (time and code) provided
- Tested against Matlab scripts

1.2 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 ’C54x DSPLIB for use on TI ’C54x DSPs with no royalty payments. Refer to Appendix C — free-license agreement section and to section 3.8 — Where DSPLIB Goes from Here of this application report for details.

2 Installing DSPLIB

2.1 DSPLIB Content

The TI DSPLIB software consists of four parts:

1. A header file for C programmers:
   - *dsplib.h*
2. Two object libraries for the two different memory models supported by TI compilers:
   - *54xdsp.lib for standards short-call mode (16-bit)*
   - *54xdspf.lib for far-call mode (24-bits)*
3. One source library to allow function customization by the end user

   54xdsp.src
4. Example programs and linker command files used under the “54x_test” subdirectory.

2.2 How to Install DSPLIB

Read README.1ST file for specific details of release.

First Step: De-archive DSPLIB

DSPLIB is distributed in the form of an executable self-extracting ZIP file (54xdsplib.exe) that will automatically restore the DSPLIB individual components in the same directory you “execute” the self-extracting file from. Following is an example on how to install DSPLIB. Just type:

54xdsplib.exe -d

The DSPLIB directory structure and content you will find is as follows:

54xdsplib (dir)	
54xdsp.lib	: use for standards short-call mode
54xdspf.lib	: use for far-call mode
blt54x.bat	: re-generate 54xdsp.lib based on 54xdsp.src
blt54xf.bat	: re-generate 54xdspf.lib based on 54xdsp.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.lib	: include file with type definitions to increase TMS320 portability
doc(dir)	
dsplib.pdf	: DSPLIB application report (this document) in PDF format
code(dir)	: contains the examples shown in the application report

Second Step: Update Your C_DIR Environment Variable

Append the full path of the 54xdsplib directory path to your C_DIR environment variable. For example, if you run the 54xdsplib.exe self-extracting file in c:\54xdsplib, and your TI DSP development tools were installed in *c:\dsptools*, add this line to your *c:\autoexec.bat* file.

Set C_DIR=. C:\54xdsplib c:\dsptools

This allows the ’C54x compiler/linker to find the ’C54x DSPLIB object libraries, 54xdsp.lib or 54xdspf.lib.

2.3 How to Rebuild DSPLIB

For full-rebuild of 54xdsp.lib and/or 54xdspf.lib

- To rebuild 54xdsp.lib, simply execute the blt54x.bat.
  Warning: This will overwrite the existing 54xdsp.lib
- To rebuild 54xdspf.lib, simply execute the blt54xf.bat.
  Warning: This will overwrite the existing 54xdspf.lib

For partial rebuild of 54xdsp.lib and/or 54xdspf.lib (modification of a specific DSPLIB function, for example fir.asm)

1. Extract the source for the selected function from the source archive:
   ar500 x 54xdsp.src fir.asm
2. Reassemble your new fir.asm assembly source file:
   asm500 –g fir.asm
3. Replace the object, fir.obj, in the dsplib.lib object library with the newly formed object:
   ar500 r 54xdsp.lib fir.obj

3 Using DSPLIB

3.1 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 type 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 type 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 DSPLIB Arguments

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

3.3 Calling a DSPLIB Function from C

In addition to correctly 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 one of the two DSPLIB object code libraries, *54xdsp.lib* or *54xdspf.lib*, depending on whether you need *far mode*.

- Use a correct linker command file describing the memory configuration available in your ’C54x board.

For example, the following code contains a call to the *acorr*, *q15tofl* and *fltoq15* routines in DSPLIB:

```
// User’s Guide example

#include "dsplib.h"

float xf[3] = { 0.1 , 0.2, 0.3};
float yf[3] = { 0 , 0, 0};

short x[3];
short y[3];
short i;


main() {

for (i=0; i<3; i++) y[i] = x[i] = 0;

fltoq15(xf,x,3);
acorr(x,y,3,3,raw);
q15tofl(y,yf,3);

}
```

In this example, the *fltoq15* and *q15tofl* DSPLIB functions are used to convert between floating point fractional values to Q15 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 under the /doc/code subdirectory. To compile and link this code with 54xdsp.lib simply issue the following command:

cl500 -pk -g -o3 -i. ug.c -z -v0 54x.cmd 54xdsp.lib -m ug.map -oug.out

or

cl500 –v548 –mf -pk -g -o3 -i. ug.c -z -v0 54x.cmd 54xdsp.lib -m ug.map -oug.out

Note: The examples presented in this application report have been tested using the Texas Instruments ’C54x EVM containing a ’C541. Therefore, the linker command file used reflects the memory configuration available in that board. Customization may be required to use it with a different board. No overlay mode is assumed (default after ’C54x device reset)

Refer to the *TMS320C54x Optimizing C Compiler User’s Guide* if more in-depth explanation is required.

Warning:

DSPLIB routines modify the 54x FRCT bit. This can cause problems for users of versions of the compiler (cl500) prior to version 3.1 if interrupt service routines (ISRs) are implemented in 'C'. Versions prior to 3.1 do not preserve the FRCT bit on ISR entry, therefore the FRCT bit may be corrupted and not restored which will lead to incorrect results. One solution is to implement the ISRs in assembly and preserve the FRCT bit. Users with version 3.1 and above need not worry about this.

3.4 Calling a DSPLIB Function from Assembly

The 'C54x 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 'C54x C compiler calling conventions. This means that the DSPLIB functions expect parameters to be passed on the stack in reverse order (except for the first argument that is passed in the 'C54x Accumulator A). Refer to the *TMS320C54x Optimizing C Compiler User's Guide* if a more in-depth explanation is required.

Keep in mind 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 might not be optimal due to unnecessary C-calling overhead.

3.5 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.
- *abias*.cmd: an example of a linker command you can use for this function ('C541 evm specific)

3.6 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. Notice 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, as a user, may have on this respect.

3.7 How DSPLIB Deals With Overflow and Scaling Issues

One of the inherent difficulties of programming for fixed-point processors, is to determine 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 is important to note that a ’C54x architectural feature that makes overflow easier to deal with is the presence of guard bits in both ’C54x accumulators. The 40-bit ’C54x accumulators provide eight guard bits to allow up to 256 consecutive MAC operations before an accumulator overrun – a very useful feature when implementing for example FIR filters.

There are four 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.
- **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 ’C54x 32-bit saturation mode (OVM bit = 1). This is the case of certain basic math functions that require the saturation mode to be enabled to work.
- **Not applicable:** In this type of function, due to the nature of the function operations, there is no overflow to worry about.

A couple of additional DSPLIB features relate to overflow/scaling handling:

- **DSPLIB reporting of overflow conditions (overflow flag):** Due to the sometimes not predictible 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, keep in mind that due to the guard-bits, the ’C54x is capable of dealing with intermediate 32-bit overflows, and still producing the correct final result. Therefore, the oflag parameter should be taken in the context of a warning but not a definitive error.
- **Functions for handling of scaling and data block exponent:** DSPLIB includes a bexp that will return the maximum exponent (extra sign bits) of a vector to allow determination of correct input scaling.

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

3.8 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 grow overtime. 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 make sure to include it in the next DSPLIB software release. Your contribution will be fully acknowledged and recognized by TI in the DSPLIB Application Report Acknowledgment Section. Use this opportunity to make your name known by your DSP industry peers. ***Simply email your contribution to dsph@ti.com and we will get in contact with you.***
- **Improved Testing Methodology and function characterization:** See section 3.6 — How DSPLIB is Tested - Allowable Error.
- **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 that working in conjunction with 'C54x compiler intrinsics make C-developing easier for fixed-point devices. However, it is 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. In the event of the endorsement by the DSP community of a standard DSP library spec, TI will take the necessary steps to evolve DSPLIB into industry compliance.

4 Function Descriptions

4.1 Arguments and Conventions Used

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

Argument	Description
x,y	Argument reflecting input data vector
r	Argument reflecting output data vector
nx,ny,nr	Arguments reflecting the size of vectors x,y, and r respectively. In functions in which case nx= nr=nr, only nx has been used across.
h	Argument reflecting filter coefficient vector (filter routines only)
nh	Argument reflecting the size of vector h
DATA	Data type definition equating a short, a 16-bit value representing a Q15 number. Use of DATA instead of short is recommended to increase future portability across devices.
LDATA	Data type definition equating a long, a 32-bit value representing a Q31 number. Use of LDATA instead of short is recommended to increase future portability across devices.
ushort	Unsigned short (16-bit). You can used this data type directly, because it has been defined in dsplib.h

4.2 DSPLIB Functions – Summary Table

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

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

Functions	Description
FFT	
void cfft (DATA x, nx, short scale)	Radix-2 complex forward FFT - MACRO
void cifft (DATA x, nx, short scale)	Radix-2 complex inverse FFT – MACRO
void rfft (DATA x, nx, short scale)	Radix-2 real forward FFT - MACRO
void rifft (DATA x, nx, short scale)	Radix-2 real inverse FFT - MACRO
void cbrev (DATA *a, DATA *r, ushort n)	Complex bit-reverse function
Filtering and Convolution	
short fir (DATA *x, DATA *h, DATA *r, DATA **dbuffer, ushort nx, ushort nh)	FIR Direct form
short firs (DATA *x, DATA *r, DATA **dbuffer, ushort nh2, ushort nx)	Symmetric FIR Direct form Optimized routine)
short int firs2 (DATA *x, DATA *h, DATA *r, DATA **dbuffer, ushort nh2, ushort nx)	Symmetric FIR Direct form (generic routine)
short firdec (DATA *x, DATA *h, DATA *r, DATA **dbuffer , ushort nh, ushort nx, ushort D)	Decimating FIR filter
short firinterp (DATA *x, DATA *h, DATA *r, DATA **dbuffer , ushort nh, ushort nx, ushort I)	Interpolating FIR filter
short cfir (DATA *x, DATA *h, DATA *r, DATA **dbuffer, ushort nh, ushort nx)	Complex FIR direct form
short convol (DATA *a, DATA *h, DATA *r, ushort na, ushort nh)	Convolution
short hilb16 (DATA *x, DATA *h, DATA *r, DATA *db, ushort nh, ushort nx)	16-bit fir Hilbert Transformer
short iircas4(DATA *x, DATA *h, DATA *r, DATA **dbuffer, ushort nbiq, ushort nx)	IIR cascade Direct Form 2. 4 coefficients per biquad.
short iircas5(DATA *x, DATA *h, DATA *r, DATA **dbuffer, ushort nbiq, ushort nx)	IIR cascade Direct Form 2. 5 coefficients per biquad
short iircas51(DATA *x, DATA *h, DATA *r, DATA **dbuffer, ushort nbiq, ushort nx)	IIR cascade Direct Form 1. 5 coefficients per biquad
short iir32(DATA *x, LDATA *h, DATA *r, LDATA **dbuffer, ushort nbiq, ushort nx)	32-bit IIR cascade Direct Form 2. 5 coefficients per biquad.
short iirlat (DATA *x, DATA *h, DATA *r, DATA *d, ushort nh, ushort nx)	Lattice inverse IIR filter
short firlat (DATA *x, DATA *h, DATA *r, DATA *d, ushort nx, ushort nh)	Lattice forward FIR filter

Functions	Description
Adaptive Filtering	
short dlms (DATA *x, DATA *h, DATA *r, DATA **d, DATA *des, DATA step, ushort nh, ushort nx)	LMS FIR (delayed version)
short ndlms (DATA *x, DATA *h, DATA *r, DATA *dbuffer, DATA *des, ushort nh, ushort nx, int l_tau, int cutoff, int gain, DATA *norm_d)	Normalized delayed LMS implementation
short nblms (DATA *x,DATA *h,DATA *r, DATA *dbuffer, DATA *des, ushort nh, ushort nx, ushort nb, DATA *norm_e, int l_tau, int cutoff, int gain)	Normalized Block LMS implementation
Correlation	
short acorr (DATA *x, DATA *r, ushort nx, ushort nr, type)	Auto-correlation (positive side only) – MACRO
short corr (DATA *x, DATA *y, DATA *r, ushort nx, ushort ny, type)	Correlation (full-length) – MACRO
Trigonometric	
Short sine (DATA *x, DATA *r, ushort nx)	sine of a vector
Short atan2_16(DATA *q, DATA *i, DATA *r, ushort nx)	4 - Quadrant Inverse Tangent of a vector
Short atan16(DATA *x, DATA *r, ushort nx)	Arctan of a vector
Math	
short add (DATA *x, DATA *y, DATA *r, ushort nx, ushort scale)	Optimized vector addition
short expn (DATA *x, DATA *r, ushort nx)	Exponent of a vector
short ldiv16(LDATA *x, DATA *y, DATA *r, DATA *exp, ushort nx)	Signed vector divide
short logn (DATA *x, LDATA *r, ushort nx)	Natural log of a vector
short log_2 (DATA *x, LDATA *r, ushort nx)	Log base 2 of a vector
Short log_10 (DATA *x, LDATA *r, ushort nx)	Log base 10 of a vector
short maxidx (DATA *x, ushort nx)	Index for maximum magnitude in a vector
short maxval (DATA *x, ushort nx)	Maximum magnitude in a vector
short minidx (DATA *x, ushort nx)	Index for minimum magnitude in a vector
short minval (DATA *x, ushort nx)	Minimum element in a vector
short mul32(LDATA *x, LDATA *y, LDATA *r, ushort nx)	32-bit vector multiply
short neg (DATA *x, DATA *r, ushort nx)	16-bit vector negate
short neg32 (LDATA *x, LDATA *r, ushort nx)	32-bit vector negate
short power (DATA *x, LDATA *r, ushort nx)	sum of squares of a vector (power)
short rand16(DATA *x, ushort nx)	Random number vector generator
void rand16init(void)	Random number generator initialization
void recip16 (DATA *x, DATA *r, DATA *rzexp, ushort nx)	Vector reciprocal
short sqrt_16 (DATA *x, DATA *r, short nx)	Square root of a vector
short sub (DATA *x, DATA *y, DATA *r, ushort nx, ushort scale)	Vector subtraction
Matrix	
short mmul (DATA *x1,short row1,short col1,DATA *x2,short row2, short col2,DATA *r)	matrix multiply
short mtrans(DATA *x, DATA *r, ushort nx)	matrix transponse
Miscellaneous	
short bexp(DATA *x, ushort nx)	max exponent (extra sign-bits) of vector. (to allow determination of correct inputscaling)
void fltoq15 (float *x, DATA *r, ushort nx)	Float to Q15 conversion
void q15tofl (DATA *x, float *r, ushort nx)	Q15 to float conversion

acorr *Auto-correlation*

short 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 >= nr

r[nr] Pointer to real output vector containing the first nr elements of the positive side of the auto-correlation 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 Auto-correlation type selector. Types supported:

- If type = raw, r will contain the raw autocorrelation of x
- If type = bias, r will contain the biased autocorrelation of x
- If type = unbias, r will contain 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 auto-correlation of the real vector x and stores the results are stored in real output vector r. Notice that the full-length auto-correlation of vector x will have 2*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 Auto-correlation: $r[j] = \sum_{k=0}^{nx-j-1} x[j+k]x[k]$ $\quad 0 <= j <= nr$

Biased Auto-correlation: $r[j] = 1/nx \sum_{k=0}^{nx-j-1} x[j+k]x[k]$ $\quad 0 <= j <= nr$

Unbiased Auto-correlation: $r[j] = 1/(nx-abs(j)) \sum_{k=0}^{nx-j-1} x[j+k]x[k]$ $\quad 0 <= j <= nr$

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:

Cycles	**Abias**	
	Core: ((na-1) * (na-2)) + ((nlags) * 13) + 26	
	Overhead 68	
Code size (in 16-bit words)	**Araw** Core: 19 + (nr * 10) + ((na-2) * (na-3))	
	Overhead	61
	Aubias Core: 4 + ((nr-2) * 37) + ((na-1) * (na-2))	
	Overhead	68
Code size (in 16-bit words)	**Abias:** **Araw:** **Aubias:**	95 words 79 words 94 words

add *Vector Add*

short 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 • Scale = 1 divide the result by 2 to prevent overflow • Scale = 0 does 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:

Cycles	Core: 12 + 3*nx/2
	Overhead 30
Code size (in 16-bit words)	39

atan16 Arctangent Implementation

short 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) e.g. 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] in 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:

Cycles	Core: 11 * nx	
	Overhead	39
Code size (in 16-bit words)	32	

atan2_16 Arctangent 2 Implementation

short oflag = atan2_16(DATA *q, DATA *i, DATA *r, ushort nx)

(defined in arct2.asm)

Arguments:

q[nx]	Pointer to quadrature input vector (in Q15 format) of size nx
i[nx]	Pointer to in-phase input vector (in Q15 format) 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) * (1/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 arc tangent 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, 0x7ffff, 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, 0xf000, 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:

Cycles	Core: 107 * nx Overhead 47
Code size (in 16-bit words)	Code size (in 16-bit words)

bexp *Block Exponent Implementation*

short maxexp = bexp(DATA *x, ushort nx)

Arguments:

	maxexp	Return value – max exponent that may be used in scaling
	x[nx]	Pointer to input vector of size nx
	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: 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:

```
for (short j=0; j<nx; j++)
        temp = exp(x[j]);
        if (temp < maxexp) maxexp = temp;
} return maxexp;
```

Overflow Handling Methodology: Not applicable

Special Requirements: None

Implementation Notes: None

Example: See examples/bexp subdirectory

Benchmarks:

Cycles	Core: 9 * nx	
	Overhead	28
Code size (in 16-bit words)	29 words	

cbrev *Complex Bit-Reverse*

void cbrev (DATA *x, 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

Note: The ’C54x 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.
- In-place bit-reversing (x = r) is much more cycle consuming compared with the off-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:

Cycles	Core: 2 + 3 * nx (off-place) 13 * nx – 26 (in-place) Overhead 21
Code size (in 16-bit words)	50 (includes support for both in-place and off-place bit-reverse)

Note: The ’C54x is capable to do an off-place bit-reverse in 2*n by using the following code:

```
stm     #N,ar0
stm      #INPUT, ar2           ; source address of data
rpt         #N*2 -1            ; looping 2*N times
mvdk    *ar2+0b, #DATA
```

The drawback of this implementation is the hard-coding of the destination address with label #DATA. The cbrev DSPLIB implementation has chosen a more generic solution at the expense at one extra cycle (3*nx).

cfir *Complex FIR Filter*

short oflag = cfir (DATA *x, DATA *h, DATA *r, DATA **dbuffer, ushort nh, ushort nx)

Arguments:

x[2*nx] Pointer to compex input vector of nx complex elements (re-Im in consecutive locations)

h[2*nh] Pointer to coefficient vector of size 2*nh (nh complex elements with re-Im in consecutive locations) in normal order. For example if nh=3: h = b0re, b0im, b1re,b1im,b2re,b2im.

- 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*nh).

r[2*nx] Pointer to complex output vector of nx complex elements (re-Im in consecutive locations)

In-place computation (r = x) is allowed

dbuffer[2*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.
- 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*nh).

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

nh Number of complex coefficients

oflag Overflow error flag
= 1 if an 32-bit data overflow has occurred in an intermediate or final result
= 0 if no 32-bit data overflow has occurred in an intermediate or final result

Description: Computes a 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:

$$r[j] = \sum_{k=0}^{nh} h[k]x[j-k] \qquad 0 <= j <= nx$$

Overflow Handling Methodology: No scaling implemented for overflow prevention.

Special Requirements: None

Implementation Notes: None

Example: See examples/cfir subdirectory

Benchmarks:

Cycles	Core: nx*(13 + 8*nh)	
	Overhead	49
Code size (in 16-bit words)	66	

Cfft *Forward Complex FFT*

void cfft (DATA x, nx, short scale);

(defined in cfft#.asm where #=nx)

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 Re-Im.

x must be aligned at 2*nx boundary, where nx = # = FFT size.
The log(nx) + 1 LSBits of address x must be zero

nx — Number of complex elements in vector x. nx must be a constant number (not a variable) and can take the following values.

nx = 8,16,32,64,128,256,512,1024

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 complex DIT FFT of the nx complex elements stored in vector x 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: (DFT)

$$y[k] = 1/(\text{scale factor}) * \sum_{i=0}^{nx-1} x[i] * (\cos(2 * pi * i * k / nx) + j \sin(2 * pi * i * k / nx))$$

Overflow Handling Methodology: Scaling implemented for overflow prevention

Special Requirements:

- Special linker command file sections required: .sintab (containing the twiddle table). For .sintab section size refer to the benchmark information below.
- This function requires the inclusion of two other files during assembling (automatically included):
  - macros.asm (contains all macros used for this code)
  - sintab.q15 (contains twiddle table section .sintab)

Implementation Notes:

- This is an FFT optimized for time. Space consumption is high due to the use of a separate sine table in each stage. This reduce MIPS count but also increases twiddle table data space.

- First 2 FFT stages implemented are implemented as a radix-4. Last stage is also unrolled for optimization. Twiddle factors are built-in and provided in the sintab.q15 that is automatically included during the assembly process.
- Special debugging consideration: This function is implemented as a macro that invokes different FFT routines according to the size. As a consequence, instead of the cfft symbol being defined, multiple cfft# symbols are (where # = nx = FFT complex size).
- This routine prevents overflow by scaling by 2 at each FFT intermediate stages.

Example: See examples/cfft subdirectory

Benchmarks:

- 8 cycles (butterfly core only)

FFT size	Cycles (Note)	Code-Size (words) .text section	Data-Size (words) .sintab section
8	149	109	0
16	322	151	11
32	733	199	34
64	1672	247	81
128	3795	295	176
256	8542	343	367
512	19049	391	750
1024	42098	439	1517

Note: 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)

cifft *Inverse Complex FFT*

void cifft (DATA x, nx, short scale)

(defined in cfft#.asm where #=nx)

Arguments:

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

X must be aligned at 2*nx boundary, where nx = # = IFFT size.
The log(nx) + 1 LSBits of address x must be zero

nx — Number of complex elements in vector x. nx must be a constant number (not a variable) and can take the following values.

nx = 8,16,32,64,128,256,512,1024

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 complex DIT IFFT of the nx complex elements stored in vector x 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] = 1/(scale factor)* $\sum_{i=0}^{nx-1} X(w) * (\cos(2 * pi * i * k / nx) - j \sin(2 * pi * i * k / nx))$

Overflow Handling Methodology: Scaling implemented for overflow prevention

Special Requirements:

- Special linker command file sections required: .sintab (containing the twiddle table). For .sintab section size refer to the benchmark information below.
- This function requires the inclusion of two other files during assembling (automatically included):
  - macrosi.asm (contains all macros used for this code)
  - sintab.q15 (contains twiddle table section .sintab)

Implementation Notes:

- This is an IFFT optimized for time. Space consumption is high due to the use of a separate sine table in each stage. This reduce MIPS count but also increases twiddle table data space.

- First 2 IFFT stages implemented are implemented as a radix-4. Last stage is also unrolled for optimization. Twiddle factors are built-in and provided in the sintab.q15 that is automatically included during the assembly process.
- Special debugging consideration: This function is implemented as a macro that invokes different IFFT routines according to the size. As a consequence, instead of the cifft symbol being defined, multiple cifft# symbols are (where # = nx = IFFT complex size)
- This routine prevents overflow by scaling by 2 at each IFFT intermediate stages.

Example: See examples/cfft subdirectory

Benchmarks:

- 8 cycles (butterfly core only)

IFFT size	Cycles(Note)	Code-size (words) .text section	data-size (words) .sintab section
8	149	109	0
16	322	151	11
32	733	199	34
64	1672	247	81
128	3795	295	176
256	8542	343	367
512	19049	391	750
1024	42098	439	1517

Note: 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) linker command file reflects those conditions)

convol *Convolution*

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

Arguments:

x[nr+nh-1]	Pointer to real input vector a of nr+nh-1 real elements
h[nh]	Pointer to real input vector h of nh real elements
r	Pointer to real output vector h of nr real elements
nr	Number of real elements in vector r
nh	Number of elements in vector h
oflag	Overflow error flag = 1 if an 32-bit data overflow has occurred in an intermediate or final result = 0 if no 32-bit data overflow has occurred in an intermediate or final result

Description: Computes the real convolution (positive) of 2 vectors a and h and places the results in vector r. Typically used for block-by-block FIR filter computation without any need of using circular addressing or restricted data alignment. This function can be used for both block-by-block and sample-by-sample filtering (nr=1).

Algorithm: $r[j] = \sum_{k=0}^{nh} h[k]x[j-k] \qquad 0 <= j <= nr$

Overflow Handling Methodology: No scaling implemented for overflow prevention

Special Requirements: None

Implementation Notes: None

Example: See examples/convol subdirectory

Benchmarks:

Cycles	Core: nr * (nh + 4)	
	Overhead	35
Code size (in 16-bit words)	43	

corr *Correlation (full-length)*

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

(defined in craw.asm, cbias.asm , cubias.asm)

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 will contain the raw correlation • If type = bias, r will contain the biased-correlation • If type = unbias, r will contain 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] = $\sum_{k=0}^{nr-j-1} x[j+k] * y[k]$ 0 <=j <=nr = nx+ny-1

Biased correlation: r[j] = 1/nr $\sum_{k=0}^{nr-j-1} x[j+k] * y[k]$ 0 <=j <=nr = nx+ny-1

Unbiased correlation: r[j] = 1/(nx – abs(j)) $\sum_{k=0}^{nr-j-1} x[j+k] * y[k]$ 0 <=j <=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

Example: See examples/cbias, examples/cubias, examples/craw subdirectories

Benchmarks:

Cycles	**Raw:** Core: 41 + (16 + (na-3)(na-2) + 17 * (na-3)) + (14 + (nb-na+1)(na-2+8))	
	Overhead	36
	Unbias: Core: 26 + (((na-3)*53) + (na-3)(na-2))+ (38 + (nb-na+1)*(11+na-2))	
	Overhead	51
	Bias: Core: 59 + (2 * ((na-3)*12 + (na-3)(na-2)/2)) + ((nb - na + 1) * (12 + na-2))	
	Overhead	51
Code size (in 16-bit words)	**Raw:**	105
	Unbias:	255
	Bias:	132

dlms *Adaptive Delayed lms Filter*

short oflag = dlms (DATA *x, DATA *h, DATA *r, DATA **d, DATA *des, 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
dbuffer[nh]	Pointer to location containing the address of the delay buffer • Memory alignment: the delay buffer 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)
des[nx]	Pointer to expected output array
step	Scale factor to control learning curve rate = 2*mu
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 LMS (Least-mean-square) 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 a. The filter output result is stored in vector r. LMS algorithm is used but adaptation using the previous error and the previous sample (“delayed”) to take advantage of the ’C54x LMS instruction.

Algorithm: FIR portion:

$$r[i] = \sum_{k=0}^{nh-1} b[k] * x[i-k] \qquad 0 <= i <= nx$$

Adaptation using the previous error and the previous sample:
e(i) = des(i)- r(i)
bk(i+1) = bk(i) + 2*mu*e(i-1)*x(i-k-1)

Overflow Handling Methodology: No scaling implemented for overflow prevention

Special Requirements: None

Implementation Notes:

- Delayed version implemented to take advantage of the ’C54x LMS instruction. Effect on covergence minimum. For reference, following is the algorithm for the regular LMS (non-delayed):

FIR portion

$$r[i] = \sum_{k=0}^{nh-1} b[k] * x[i-k] \qquad 0 <= i <= nx$$

Adaptation using the current error and the current sample:

e(i) = des(i)- r(i)
*bk(i+1) = bk(i) + 2*mu*e(i)*x(i-k)*

Example: See examples/dlms subdirectory

Benchmarks:

Cycles	Core: nx * (18 + 2*(nh-2)) = nx * (14+ 2*nh)
	Overhead 45
Code size (in 16-bit words)	62

expn *Exponential Base e*

short 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)= ex(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) = c1*x + c2* x ^2 + c3*x^3 + c4*x^4 + c5*x^5

  where

  c1 = 0.0139
  c2 = 0.0348
  c3 = 0.1705
  c4 = 0.4990
  c5 = 1.0001

Example: See examples/expn subdirectory

Benchmarks:

Cycles	Core: 12*nx	
	Overhead	32
Code size (in 16-bit words)	36	

fir *FIR F*ilter

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

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 …

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).

r[nx] — Pointer to real input vector of nx real elements. In-place computation (r = x) is allowed.

dbuffer[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.
- 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 (input samples)

nh — Number of coefficients

oflag — Overflow error flag
= 1 if an 32-bit data overflow has occurred in an intermediate or final result
= 0 if no 32-bit data overflow has occurred in an intermediate or final result

Description: Computes a 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[j] = $\sum_{k=0}^{nh} h[k]x[j-k]$ 0 <=j <=nx

Overflow Handling Methodology: No scaling implemented for overflow prevention.

Special Requirements: None

Implementation Notes

- You can also use the convolution function for filtering, by having an input buffer x padded with nh-1 zeros at the beginning of the x buffer. However, having an fir filter implementation that uses a totally independent delay buffer *(dbuffer)* gives you more control in the relocation in memory of your data buffers in the case of a dual-buffering filtering scheme.

Example: See examples/fir subdirectory

Benchmarks:

Cycles	Core: 4 + nx*(4+nh)
	Overhead 34
Code size (in 16-bit words)	42

firdec *Decimating FIR Filter*

short 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 … 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).
r[nx/D]	Pointer to real input vector of nx/D real elements. In-place computation (r = x) is allowed.
dbuffer[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. • 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 = 1 if an 32-bit data overflow has occurred in an intermediate or final result = 0 if no 32-bit data overflow has occurred in an intermediate or final result

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).

Algorithm: $r[j] = \sum_{k=0}^{nh} h[k]x[j * D - k]$ 0 <=j <=nx

Overflow Handling Methodology: No scaling implemented for overflow prevention.

Special Requirements: None

Implementation Notes: None

Example: See examples/decim subdirectory

Benchmarks:

Cycles	Cycles (nx/D)*(12+nh+4(D-1))	
	Overhead	86
Code size (in 16-bit words)	67	

firinterp *Interpolating FIR Filter*

short 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 = b0 b1 b2 b3 …

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).

r[nx*I] Pointer to real output vector of nx real elements. In-place computation (r = x) is allowed.

dbuffer[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.
- 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 and r

nh Number of coefficients

I Interpolation factor. For example an I = 2 means you will add one sample result for every sample

oflag Overflow error flag
= 1 if an 32-bit data overflow has occurred in an intermediate or final result
= 0 if no 32-bit data overflow has occurred in an intermediate or final result

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

Overflow Handling Methodology: No scaling implemented for overflow prevention.

Special Requirements: None

Implementation Notes: None

Example: See examples/decimate subdirectory

Benchmarks:

Cycles	Core: nx*(6+(I-1)*(17+(nh/I) Overhead 88
Code size (in 16-bit words)	74

firs *Symmetric FIR Filter*

short oflag = int firs (DATA *x, DATA *r, DATA **dbuffer, ushort nh2, ushort nx)

Arguments:

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

r[nx] Pointer to real input vector of nx real elements. In-place computation (r = x) is allowed.

dbuffer[2*nh2] Delay buffer of size nh = 2*nh2

- 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*nh2).

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

nh2 Half the number of coefficients of the filter (due to symmetry there is no need to provide the other half)

oflag Overflow error flag
= 1 if an 32-bit data overflow has occurred in an intermediate or final result
= 0 if no "

Description: Computes a real FIR filter (direct-form) using the nh2 coefficients stored in program location pointed by TI_LIB_COEFFS global label. The filter is assumed to have a symmetric impulse response, with the first half of the filter coefficients stored in locations pointed by TI_LIB_COEFFS. 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: $r[j] = \sum_{k=0}^{nh} h[k]x[t-k]$ 0 <=j <=nx

where h is symmetric (for example h = h0 h1 h2 h2 h1 h0

where nh2 = 3. Only h0, h1, h2 are stored in program memory pointed by the TI_LIB_COEFFS global label)

Overflow Handling Methodology: No scaling implemented for overflow prevention.

Special Requirements:

- Filter coefficients must be provided in program space with a global label called TI_LIB_COEFFS pointing to the start of the coefficient table.

Implementation Notes

- Although this routine is faster than the generic symmetric filter routine (firs2) included in DSPLIB, it is restrictive in that the address for the coefficients is hard-coded to the global label TI_LIB_COEFFS in program memory. This could be a problem in the event you want to use multiple filtering routines with different coefficient values. If that is the case, use the firs2 routine

Example: See examples/firs subdirectory

Benchmarks:

Cycles	Core: nx * (16+nh) Overhead 35
Code size (in 16-bit words)	56

firs2 Symmetric FIR Filter (generic)

short oflag = int firs2 (DATA *x, DATA *h, DATA *r, DATA **dbuffer, ushort nh2, ushort nx)

Arguments:

x[nx]	Pointer to real input vector of nx real elements.
r[nx]	Pointer to real input vector of nx real elements. In-place computation (r = x) is allowed.
h[nh2]	Pointer to vector containing 1st half the filter coefficients. It assumes that the filter has a symmetric impulse response (filter coefficients). The total number of filter coefficients is 2*nh2. For example if: The filter coefficients are b0 b1 b1 b0 then nh2 = 2 and h = { b0, b1 }
dbuffer[2*nh2]	Delay buffer of size nh = 2*nh2 • 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*nh2).
nx	Number of real elements in vector x (input samples)
nh2	Half the number of coefficients of the filter (due to symmetry there is no need to provide the other half)
oflag	Overflow error flag = 1 if an 32-bit data overflow has occurred in an intermediate or final result = 0 if no "

Description: Computes a real FIR filter (direct-form) using the nh2 coefficients stored in array h (data memory). The filter is assumed to have a symmetric impulse response, so array h stores only the first half of the filter coefficients. 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: $r[j] = \sum_{k=0}^{nh} h[k]x[t-k] \qquad 0 <= j <= nx$

where h is symmetric (for example h = h0 h1 h2 h2 h1 h0
where nh2 = 3. Only h0, h1, h2 are stored in data memory)

Overflow Handling Methodology: No scaling implemented for overflow prevention.

Special Requirements: None

Implementation Notes

- Although this routine is slower than the symmetric filter routine (firs) included in DSPLIB, it does not impose any restrictions in the location of the coefficient vector or in the use of multiple filtering routines in the same executable.

Example: See examples/firs2 subdirectory

Benchmarks:

Cycles	Core: nx*(15 + 2*nh2)
	Overhead 43
Code size (in 16-bit words)	58

fltoq15 *Float to q15 Conversion*

short 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 & 2 were encountered

Description: Convert the IEEE floating point numbers store 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:

Cycles	Core: 19 + 40*nx	
	Overhead	43
Code size (in 16-bit words)	60	

hilb16 *FIR Hilbert Transformer*

oflag = short hilb16 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nh, ushort nx)

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 b4 …..
Every odd valued filter coefficient has to be 0, i.e. b1 = b3= … = 0
And h = b0 0 b2 0 b4 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).

r[nx] — Pointer to real input vector of nx real elements. In-place computation (r = x) is allowed

dbuffer[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.
- 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 (input samples)

nh — Number of coefficients

oflag — Overflow error flag
= 1 if an 32-bit data overflow has occurred in an intermediate or final result
= 0 if no 32-bit data overflow has occurred in an intermediate or final result

Description: Computes a 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: $r[j] = \sum_{k=0}^{nh} h[k]x[j-k]$ 0 <=j <=nx

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 5 tap filter may look like this:
h = [0.876 0 –0.324 0 –0.002]

Implementation Notes You can also use the convolution function for filtering, by having an input buffer x padded with nh-1 zeros at the beginning of the x buffer. However, having an fir filter implementation that uses a totally independent delay buffer *(dbuffer)* gives you more control in the relocation in memory of your data buffers in the case of a dual-buffering filtering scheme.

Example: See examples/fir subdirectory

Benchmarks:

Cycles	Core: nx*(4+nh)	
	Overhead	53
Code size (in 16-bit words)	42	

iir32 Double-precision IIR Filter

short oflag = iir32(DATA *x, LDATA *h, DATA *r, LDATA **dbuffer, ushort nbiq, ushort nx)

Arguments:

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

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 beginning of biquad 1
b21 – low
b11 – high
b11 – low
b01 – high
b01 – low
a21 – high
a21 – low
a11 – high
a11 – low
b22 – high beginning of biquad 2 coefs
b22 – low
b12 – high
b12 – low
b02 – high
b02 – low
a22 – high
a22 – low
a12 – high
a12 – low

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

dbuffer[3*nbiq] — 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 beginning of biquad 1
d1(n-2) – high
d1(n-1) – low
d1(n-1) – high
d1(n) – low
d1(n) – high
d2(n-2) - low beginning of biquad 2
d2(n-2) - high
d2(n-1) – low
d2(n-1) – high
d2(n) – low
d2(n) – high

- 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 (3*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

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 a. 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*d(n-1) – a2*d(n-2)*
*y(n) = b0*d(n) + b1*d(n-1) + b2*d(n-2)*

Overflow Handling Methodology: No scaling implemented for overflow prevention.

Special Requirements: None

Implementation Notes: None

Example: See examples/iir32 subdirectory

Benchmarks:

Cycles	Core: 4 + nx*(12 + 48*nbiq) Overhead 58
Code size (in 16-bit words)	110

iircas4 *Cascaded IIR Direct Form II Using 4-Coefs per Biquad*

short 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 (i.e. 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 (i.e. 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

Description: Computes a cascade IIR filter of nbiquad 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 a. 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*d(n-1) – a2*d(n-2)*
*y(n) = d(n) + b1*d(n-1) + b2*d(n-2)*

Overflow Handling Methodology: No scaling implemented for overflow prevention.

Special Requirements: None

Implementation Notes: None

Example: See examples/iircas4 subdirectory

Benchmarks:

Cycles	Core: nx * (11 + 4*nbiq) Overhead 40
Code size (in 16-bit words)	50

iircas5 *Cascaded IIR Direct Form II (5-Coefs per Biquad)*

short 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 (i.e. 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(i.e. 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

Description: Computes a cascade IIR filter of nbiquad 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 a. 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 use of 5 coefficients instead of 4, facilitates the design of filters with Unit gain less that one (for overflow avoidance) typically achieved by filter coefficient scaling.

Algorithm: (for biquad)

*d(n) = x(n) – a1*d(n-1) – a2*d(n-2)*
*y(n) = b0*d(n) + b1*d(n-1) + b2*d(n-2)*

Overflow Handling Methodology: No scaling implemented for overflow prevention.

Special Requirements: None

Implementation Notes: None

Example: See examples/iircas5 subdirectory

Benchmarks:

Cycles	Core: nx * (11 + 5*nbiq) Overhead 40
Code size (in 16-bit words)	51

iircas51 *Cascaded IIR Direct Form I (5-Coefs per Biquad)*

short 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 (i.e. a21: is the a2 coefficient of biquad 1) where I is the biquad index (i.e. 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 adress 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(i.e. 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

Description: Computes a cascade IIR filter of nbiquad 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 a. The filter output result is stored in vector r.

Computes a cascade IIR filter of nbiquad 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 a. The filter output result is stored in vector r.

The use of 5 coefficients instead of 4, facilitates the design of filters with Unit gain less that one (for overflow avoidance) typically achieved by filter coefficient scaling.

Algorithm: (for biquad)

*y(n)= b0*x(n) + b1*x(n-1) + b2*x(n-2) - a1*y(n-1) - a2*y(n-2)*

Overflow Handling Methodology: No scaling implemented for overflow prevention.

Special Requirements: None

Implementation Notes: This implementation does not use circular addressing for the delay buffer. Instead it takes advantage of the 54x DELAY instruction. For this reason the delay buffer pointer will always point to the top between successive block calls.

Example: See examples/iircas51 subdirectory

Benchmarks:

Cycles	Core: nx * (13 + 8*nbiq)
	Overhead 44
Code size (in 16-bit words)	58

iirlat *Lattice Inverse (IIR) Filter*

short oflag = iirlat (DATA *x, DATA *h, DATA *r, DATA *d, int nh, int nx)

Arguments:

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

h[nh] Pointer to lattice coefficient vector of size nh in normal order:
h = b0 b1 b2 b3 …

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).

r[nx] Pointer to real input vector of nx real elements. In-place computation (r = x) is allowed.

d[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.
- 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 (input samples)

nh Number of coefficients

oflag Overflow error flag
= 1 if an 32-bit data overflow has occurred in an intermediate or final result
= 0 if no 32-bit data overflow has occurred in an intermediate or final result

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:

$e_N[n] = x[n]$
$e_{i-1}[n] = e_i[n] + hie'_{i-1}[n-1]$, $i = N, (N-1), \ldots, 1$
$e'_i[n] = -k_i e_{i-1}[n] + e'_{i-1}[n-1]$, $i = N, (N-1), \ldots, 1$
$y[n] = e_0[n] = e'_0[n]$

Overflow Handling Methodology: No scaling implemented for overflow prevention.

Special Requirements: None

Implementation Notes: None

Example: See examples/iirlat subdirectory

Benchmarks:

Cycles	Core: nx[(3*nh) + 14]
Overhead	48
Code size (in 16-bit words)	49

firlat *Lattice Forward (FIR) Filter*

short oflag = firlat (DATA *x, DATA *h, DATA *r, DATA *d, int nx, int nh)

Arguments:

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

h[nh] Pointer to lattice coefficient vector of size nh in normal order:
h = b0 b1 b2 b3 …

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).

r[nx] Pointer to real input vector of nx real elements. In-place computation (r = x) is allowed.

d[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.
- 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 (input samples)

nh Number of coefficients

oflag Overflow error flag
= 1 if an 32-bit data overflow has occurred in an intermediate or final result
= 0 if no 32-bit data overflow has occurred in an intermediate or final result

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 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:

$e_0[n] = e'_0[n] = x[n],$
$e_i[n] = e_{i-1}[n] - hie'_{i-1}[n-1],$ $\quad i = 1, 2, \ldots, N$
$e'_i[n] = -h_i e_{i-1}[n] + e'_{i-1}[n-1],$ $\quad i = 1, 2, \ldots, N$
$y[n] = e_N[n]$

Overflow Handling Methodology: No scaling implemented for overflow prevention.

Special Requirements: None

Implementation Notes: None

Example: See examples/firlat subdirectory

Benchmarks:

Cycles	Core: nx[(3*nh) + 18]	
	Overhead	61
Code size (in 16-bit words)	64	

log_2 Base 2 Logarithm

short 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)= log2 x(i) where -1 < x(i) < 1

Overflow Handling Methodology: No scaling implemented for overflow prevention

Special Requirements: None

Implementation Notes:

y = 1.4427 * ln(x) with x = M(x)*2^P(x) = M*2^P
y = 1.4427 * (ln(M) + ln(2)*P)
y = 1.4427 * (ln(2*M) + (P-1)*ln(2))
y = 1.4427 * (ln((2*M-1)+1) + (P-1)*ln(2))
y = 1.4427 * (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*u+C5)*u+C4)*u+C3)*u+C2)*u+C1)*u+C0
for 0<= u <= 1.

The polynomial coefficients Ci 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 Bi used in the calculation are derived from the Ci as follows:

B0	Q30	1581d	0062Dh
B1	Q14	16381d	03FFDh
B2	Q15	-16298d	0C056h
B3	Q16	20693d	050D5h
B4	Q17	-24950d	09E8Ah
B5	Q18	21677d	054Adh
B6	Q19	-9130d	0DC56h

Example: See examples/log_2 subdirectory

Benchmarks:

Cycles	Core: 60*nx	
	Overhead	56
Code size (in 16-bit words)	85	

log_10 Base 10 Logarithm

short 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: for (i=0; i<nx; i++) y(i)= log10 x(i) where -1 < x(i) < 1

Overflow Handling Methodology: No scaling implemented for overflow prevention

Special Requirements: None

Implementation Notes:

y = 0.4343 * ln(x) with x = M(x)*2^P(x) = M*2^P
y = 0.4343 * (ln(M) + ln(2)*P)
y = 0.4343 * (ln(2*M) + (P-1)*ln(2))
y = 0.4343 * (ln((2*M-1)+1) + (P-1)*ln(2))
y = 0.4343 * (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*u+C5)*u+C4)*u+C3)*u+C2)*u+C1)*u+C0
for 0<= u <= 1.

The polynomial coefficients Ci 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 Bi used in the calculation are derived from the Ci as follows:

B0	Q30	1581d	0062Dh
B1	Q14	16381d	03FFDh
B2	Q15	-16298d	0C056h
B3	Q16	20693d	050D5h
B4	Q17	-24950d	09E8Ah
B5	Q18	21677d	054ADh
B6	Q19	-9130d	0DC56h

Example: See examples/log_10 subdirectory

Benchmarks:

Cycles	Core: 55*nx	
	Overhead	56
Code size (in 16-bit words)	82	

logn *Base e Logarithm (natural logarithm)*

short 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)= logn x(i) where -1 < x(i) < 1

Overflow Handling Methodology: No scaling implemented for overflow prevention

Special Requirements: None

Implementation Notes:

y = 0.4343 * ln(x) with x = M(x)*2^P(x) = M*2^P
y = 0.4343 * (ln(M) + ln(2)*P)
y = 0.4343 * (ln(2*M) + (P-1)*ln(2))
y = 0.4343 * (ln((2*M-1)+1) + (P-1)*ln(2))
y = 0.4343 * (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*u+C5)*u+C4)*u+C3)*u+C2)*u+C1)*u+C0
for 0<= u <= 1.

The polynomial coefficients Ci 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 Bi used in the calculation are derived from the Ci as follows:

B0	Q30	1581d	0062Dh
B1	Q14	16381d	03FFDh
B2	Q15	-16298d	0C056h
B3	Q16	20693d	050D5h
B4	Q17	-24950d	09E8Ah
B5	Q18	21677d	054ADh
B6	Q19	-9130d	0DC56h

Example: See examples/logn subdirectory

Benchmarks:

Cycles	Core: 39*nx	
	Overhead	56
Code size (in 16-bit words)	67	

maxidx *Index of the Maximum Element of a Vector*

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

(defined in maxidx.asm)

Arguments:

x[nx]	Pointer to input vector of size nx
r	Index for vector element with maximum value
nx	Length of input data vector (nx >= 6)

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

Algorithm: Not applicable

Overflow Handling Methodology: Not applicable

Special Requirements: None

Implementation Notes: None

Example: See examples/maxidx subdirectory

Benchmarks:

Cycles — Core:

- 27 + 3*nx (if n even) – approx
- 31 + 3*nx

Overhead 27

Code size (in 16-bit words) 66

maxval *Maximum Value of a Vector*

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:

Cycles	Core: 2*nx	
	Overhead	16
Code size (in 16-bit words)	13	

minidx *Index of the Minimum Element of a Vector*

short r = 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	Lenght of input data vector (nx >= 6)

Description: Returns the index of the minimum element of a vector x. In case of multiple minimum elements, r contains the index of the last minimum element found.

Algorithm: Not applicable

Overflow Handling Methodology: Not applicable

Special Requirements: None

Implementation Notes: Different implementation than maxidx because unable to use cmps instruction with min.

Example: See examples/minidx subdirectory

Benchmarks:

Cycles	Core: 4 + 5*nx	
	Overhead	18
Code size (in 16-bit words)	22	

minval *Minimum Value of a Vector*

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

(defined in minval.asm)

Arguments:	x[nx]	Pointer to input vector of size nx
	r	Maximum value of a vector
	nx	Lenght 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:

Cycles	Core: 2*nx	
	Overhead	16
Code size (in 16-bit words)	13	

mmul *Matrix Multiplication*

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

Arguments:

x1[row1*col1]:	Pointer to input vector of size nx Pointer to input matrix of size row1*col1 ; row1 : ; :; : ; r[row1*col2] : Pointer to output data vector of size row1*col2
row1	number of rows in matrix 1
col1	number of columns in matrix 1
x2[row2*col2]:	Pointer to input matrix of size row2*col2
row2	number of rows in matrix 2
col2	number of columns in matrix 2
r[row1*col2]	Pointer to output matrix of size row1*col2
nx	Length of input data vector

Description: Returns the minimum element of a vector x

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) * B(j,k)
       C(i,k) = temp
   }
```

Overflow Handling Methodology: Not applicable

Special Requirements: Verify that the dimensions of input matrices are legal.

Implementation Notes: None

Example: See examples/minval subdirectory

Benchmarks:

Cycles Core:
row1*(7+(11+(6*col1))*col2)

Overhead 71

Code size (in 16-bit words) 65

mtrans *Matrix Transpose*

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

(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 of size nx containing

Description: This function transponse matrix x

Algorithm:

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

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

Special Requirements: None

Implementation Notes: None

Example: See examples/mtrans subdirectory

Benchmarks:

Cycles	Core: [5+(col*6)]	
	Overhead	44
Code size (in 16-bit words)	34	

mul32 32-bit Vector Multiply

short 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 multiply two 2 32-bit Q31 vectors, element by element, and produce 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:

Cycles	Core: 7*nx + 4	
	Overhead	29
Code size (in 16-bit words)	35	

nblms *Normalized Block LMS Block Filter*

short oflag = nblms (DATA *x,DATA *h,DATA *r, DATA **dbuffer, DATA *des, ushort nh, ushort nx, ushort nb, DATA **norm_e, int l_tau, int cutoff, int gain)

(defined in nblms.asm)

Arguments:

x[nx]	Input data vector of size nx (reference input)
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.
dbuffer[nh]	Pointer to location containing the address of the delay buffer Memory alignment: the delay buffer 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)
des[nx]	Pointer to expected output array
nh	Number of filter coefficients. Filter order = nh-1. nh >=3
nx	Length of input and output data vectors
nb	number of blocks
bsize	blocksize (number of coefficients to be updated for each input sample) **Note:** nh (number of coefficients) = nb*bsize
norm_e	pointer to normalized error buffer
l_tau	decay constant for long-term filtering of power estimate
cutoff	the lowest allowed value for power estimate
gain	step size constant: 2*beta= beta1/abs_power = 2^(gain) / abs_power
oflag	Overflow flag • If oflag = 1 a 32-bit overflow has occurred • If oflag =0 a 32-bit overflow has not occurred

Description: Normalized Delayed LMS (NDLMS) Block FIR implementation using coefficients stored in vector h. Coefficients are updated after each sample based on the LMS algorithm. The real data input is stored in vector a. The filter output result is stored in vector r.

LMS algorithm is used but adaptation uses the previous error and the previous sample (“delayed”) and takes advantage of the ’C54x LMS instruction.

Restrictions: This version does not allow consecutive calls to this routine in a dual buffering fashion.

Algorithm: For a more detailed description of the algorithm, refer to [4].

FIR portion

$$r[i] = \sum_{k=0}^{nh-1} b[k] * x[i-k] \qquad 0 <= i <= nx$$

Adaptation using the previous error and the previous sample

e(i) = d(i) - y(i); (error)
*var(i) = (1-beta)*var(i-1) + beta*[abs(x(i)) + cutoff];* (signal power estimate)

for *(j=0: j<nb; j++)*
{
*bkj(i+1) = bkj(i) + [2*mu*e(i)*x(i-k)]/[var(i)^2]*
}

Overflow Handling Methodology: No scaling implemented for overflow prevention

Special Requirements:

- Linker command file: you must allocate .ebuffer section (for polynomial coefficients)

Implementation Notes:

- Delayed version implemented to take advantage of the ’C54x LMS instruction. Effect on covergence minimum. For reference, following is the algorithm for the regular LMS (non-delayed):

FIR portion

$$r[i] = \sum_{k=0}^{nh-1} b[k] * x[i-k] \qquad 0 <= i <= nx$$

Adaptation using the current error and the current sample:

e(i) = des(i)- r(i)
*bk(i+1) = bk(i) + 2*mu*e(i)*x(i-k)*

Example: See examples/ndlms subdirectory

Benchmarks:

Cycles	Core: [85+bsize+nh+((18+bsize)*nb)]*nx	
	Overhead	88
Code size (in 16-bit words)	144	

ndlms *Normalized Delayed LMS Filter*

short oflag = ndlms (DATA *x, DATA *h, DATA *r, DATA *dbuffer, DATA *des, ushort nh, ushort nx, int l_tau, int cutoff, int gain, DATA *norm_d)

(defined in ndlms.asm)

Arguments:

x[nx]	input data vector of size nx (reference input)
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.
dbuffer[nh]	Pointer to location containing the address of the delay buffer Memory alignment: the delay buffer 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)
des[nx]	Pointer to expected output array
nh	Number of filter coefficients. Filter order = nh-1. nh >=3
nx	Length of input and output data vectors
l_tau	Decay constant for long-term filtering of power estimate
cutoff	the lowest allowed value for power estimate
gain	step size constant: 2*beta= beta1/abs_power = 2^(gain) / abs_power
norm_d	pointer to normalized delay buffer
oflag	Overflow flag. • If oflag = 1 a 32-bit overflow has occurred • If oflag =0 a 32-bit overflow has not occurred

Description: Normalized Delayed LMS (NDLMS) Block FIR implementation using coefficients stored in vector h. Coefficients are updated after each sample based on the LMS algorithm. The real data input is stored in vector a. The filter output result is stored in vector r.

LMS algorithm is used but adaptation using the previous error and the previous sample (“delayed”) to take advantage of the ’C54x LMS instruction.

Restrictions: This version does not allow consecutive calls to this routine in a dual buffering fashion.

Algorithm: For a more detailed description of the algorithm, refer to [4].

FIR portion

$$r[i] = \sum_{k=0}^{nh-1} b[k] * x[i-k] \qquad 0 <= i <= nx$$

Adaptation using the previous error and the previous sample

e(i) = des(i)- r(i)
*var(i) = (1-beta)*var(i-1) + beta*[abs(x(i)) + cutoff];*
*bk(i+1) = bk(i) + [2*mu*e(i-1)*x(i-k-1)]/[var(i)^2]*

Overflow Handling Methodology: No scaling implemented for overflow prevention

Special Requirements: None

Implementation Notes:

Delayed version implemented to take advantage of the ’C54x LMS instruction. Effect on covergence minimum.

Example: See examples/ndlms subdirectory

Benchmarks:

Cycles	Core: [85+bsize+nh+((18+bsize)*nb)]*nx
	Overhead 88
Code size (in 16-bit words)	144

neg *Vector Negate*

short 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

This shoud be taken it as a warning: 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: 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:

Cycles	Core: [85+bsize+nh+((18+bsize)*nb)]*nx
	Overhead 20
Code size (in 16-bit words)	21

neg32 *Vector Negate (double-precision)*

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

(defined in neg32.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*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
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 This should be take it as a warning: 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++)
x (i) = -x (i)
```

Overflow Handling Methodology: Saturation implemented for overflow handling

Special Requirements: None

Implementation Notes: None

Example: See examples/neg32 subdirectory

Benchmarks:

Cycles	Core: 4*nx + 4	
	Overhead	18
Code size (in 16-bit words)	19	

power *Vector Power*

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

(defined in power.asm)

Arguments:

x[nx]	Pointer to input data vector 1 of size nx. In-place processing allowed (r can be = x = y)
r[1]	Pointer to output data vector element in Q31 format 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
nx	Number of elements of input 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 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:

Cycles	Core: nx + 4	
	Overhead	18
Code size (in 16-bit words)	18	

q15tofl *Q15 to Float Conversion*

void 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 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:

Cycles	Core: 11+36*nx	
	Overhead	15
Code size (in 16-bit words)	56	

rand16init Initialize Random Number Generator

void rand16init(void)

(defined in rand16i.asm)

Arguments: None

Description: Initializes seed for 16 bit random number generation routine

Algorithm: Not applicable

Overflow Handling Methodology: No scaling implemented for overflow handling

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

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

Example: See examples/rand subdirectory

Benchmarks:

Cycles	Total 7
Code size (in 16-bit words)	5

rand16 Random Vector Generation

short oflag = rand16(DATA *x, ushort nx)

(defined in rand16.asm)

Arguments:

x[nx]	Pointer to input data vector 1 of size nx
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: Computes vector of 16 bit random numbers

Algorithm: Linear Congruential Method

Overflow Handling Methodology: Not applicable

Special Requirements: None

Implementation Notes: None

Example: See examples/rand16 subdirectory

Benchmarks:

Cycles	Core: 13 + nx*4	
	Overhead	10
Code size (in 16-bit words)	28	

recip16 *16-bit Reciprocal Function*

void recip16 (DATA *x, DATA *r, DATA *rexp, ushort nx)

(defined in recip16.asm)

Arguments:

x[nx]	Pointer to input data vector 1 of size nx
r[nx]	Pointer to output data buffer
rexp[nx]	Pointer to exponent buffer for output values. These exponent values are in integer format.
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] * rexp[i] = true reciprocal in floating-point

Algorithm: Appendix-Calculating a reciprocal of a Q15 number

Overflow Handling Methodology: None

Special Requirements: None

Implementation Notes: None

Example See examples/recip16 subdirectory

Benchmarks:

Cycles	Core: 4 + nx * 54
	Overhead 24
Code size (in 16-bit words)	77 words + 15 words of data space

rfft *Forward Real FFT (in-place)*

void rfft (DATA x, nx, short scale)

(defined in rfft#.asm where #=nx)

Arguments:

x[nx] Pointer to input vector containing nx real elements in bit-reversed order. On output, vector x contains the 1st 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

x must be aligned at 2*nx boundary, where nx = # = FFT size.
The log(nx) + 1 LSBits of address x must be zero

nx Number of real elements in vector x. nx **must be a constant number** (not a variable) and 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)

y[k] = 1/(scale factor) * $\sum_{i=0}^{nx-1} x[i] * (\cos(2 * pi * i * k / nx) + j \sin(2 * pi * i * k / nx))$

Overflow Handling Methodology: Scaling implemented for overflow prevention (See section 6.3)

Special Requirements:

- Special linker command file sections required: .sintab (containing the twiddle table). For .sintab section size refer to the benchmark information below.
- This function requires the inclusion of two other files during assembling (automatically included):
  - macros.asm (contains all macros used for this code)
  - sintab.q15 (contains twiddle table section .sintab)
  - unpack.asm (containing code to for unpacking results)

Implementation Notes:

- Implemented as a complex FFT of size 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*N real numbers) that will not fit in the input sequence. To accomodate 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).
- Special debugging consideration: This function is implemented as a macro that invokes different FFT routines according to the size. As a consequence, instead of the rfft symbol being defined, multiple rfft# symbols are (where # = nx = FFT real size)
- 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 subdirectory

Benchmarks:

- 8 cycles (butterfly core only)

FFT size	Cycles(Note)	Code-size (words) .text section	data-size (words) .sintab section
16	264	171	11
32	’C541	213	34
64	1160	261	81
128	2516	309	176
256	5470	357	367
512	11881	405	750
1024	25716	453	1517

Note: 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 that)

rifft *Inverse Real FFT (in-place)*

void rifft (DATA x, nx, short scale)

(defined in rifft#.asm where #=nx)

Arguments:

x[nx] Pointer to input vector x containing nx real elements in bit-reversed order, shown below for nx = 8:

Y(0)Re y(nx/2)im → DC and Nyquist
y(2)Re y(2)Im
y(1)Re y(1)Im
y(nx/2)Re y(nx/2)Im

where y = fft(x)

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

Complex numbers are stored in Re-Im format

x must be aligned at 2*nx boundary, where nx = # = IFFT size. The log(nx) + 1 LSBits of address x must be zero

nx Number of real elements in vector x. nx **must be a constant number** (not a variable) and 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 1st nx/2 complex elements of the IFFT(x) are stored in vector x in normal-order.

Algorithm: (IDFT)

$$y[k] = 1/(\text{scale factor}) * \sum_{i=0}^{nx-1} X(w) * (\cos(2 * pi * i * k / nx) - j \sin(2 * pi * i * k / nx))$$

Overflow Handling Methodology: Scaling implemented for overflow prevention

Special Requirements:

- Special linker command file sections required: .sintab (containing the twiddle table). For .sintab section size refer to the benchmark information below.
- This function requires the inclusion of two other files during assembling (automatically included):
  - macrosi.asm (contains all macros used for this code)
  - sintab.q15 (contains twiddle table section .sintab)
  - unpacki.asm (containing code to for unpacking results)

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 accomodate 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).
- Special debugging consideration: This function is implemented as a macro that invokes different IFFT routines according to the size. As a consequence, instead of the rfft symbol being defined, multiple rifft# symbols are (where # = nx = IFFT real size)
- 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

Benchmarks:

- 8 cycles (butterfly core only)

IFFT size	Cycles (Note)	Code-size (words) .text section	data-size (words) .sintab section
16	264	171	11
32	’C541	213	34
64	1160	261	81
128	2516	309	176
256	5470	357	367
512	11881	405	750
1024	25716	453	1517

Note: 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 that)

sine *Sine*

short oflag = sine (DATA *x, DATA *r, ushort nx)

(defined in sine.asm)

Arguments:

x[nx] Pointer to input vector of size nx. x contains the angle in radians between [-pi, pi] normalized between [-1,1) in q15 format
x = xrad /pi

For example:
45o = pi/4 will be equivalent to x = 1/4 = 0.25 = 0x200 in q15 format.

r[nx] Pointer to output vector containing the sine of vector x in q15 format

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: 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)

Algorithm:

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

where x(i) = xrad/pi

Overflow Handling Methodology: Not applicable

Special Requirements:

- Linker command file: .data section must be allocated.

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:

Cycles	Core:	
	20*nx	(worst case)
	18*nx	(best case)
	Overhead	23
Code size (in 16-bit words)	41 (in program space)	
	6 (in data space)	

sqrt_16 Square Root of a 16-bit Number

short 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). In-place operation is allowed (r can be equal to 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))          where  0<=i <=nx
```

Overflow Handling Methodology: Not applicable

Special Requirements: None

Implementation Notes: None

Example: See examples/sine subdirectory

Benchmarks:

Cycles	Core: 42*nx	
	Overhead	41
Code size (in 16-bit words)	68	

sub *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 • Scale = 1 divide the result by 2 to prevent overflow • Scale = 0 does 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/sub subdirectory

Benchmarks:

Cycles	Core: 12 + 3*nx/2	
	Overhead	30
Code size (in 16-bit words)	39	

5 DSPLIB Benchmarks and Performance Issues

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 an specific function, you should have in mind that the generic nature of DSPLIB might add extra cycles not required for customer specific use.

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 16-bit words 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 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 TI ’C54x EVM 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.

6 Licensing, Warranty and Support

6.1 Licensing and Warranty

’C54x 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

’C54x 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 question or want to report problems or suggestions regarding the ’C54x DSPLIB, contact Texas Instruments at dsph@ti.com. We encourage the use of the software report form (report.txt) contained in the DSPLIB doc directory to report any problem associated with the ’C54xDSPLIB.

7 References

1. The MathWorks, Inc. *Matlab Signal Processing Toolbox User’s Guide.* Natick, MA: The MathWorks, Inc., 1996.
2. Lehmer, D.H. "Mathematical Methods in large-scale computing units." *Proc. 2nd Sympos. on Large-Scale Digital Calculating Machinery, Cambridge, MA, 1949*. Cambridge, MA: Harvard University Press, 1951.
3. Oppenheim, Alan V. and Ronald W Schafer. *Discrete-Time Signal Processing.* Englewood Cliffs, NJ: Prentice Hall, 1989.
4. Digital Signal Processing with the TMS320 Family (SPR012)
5. TMS320C54x DSP CPU and Peripherals. Reference Set – Volume 1 (SPRU131)
6. TMS320C54x Optimizing C Compiler – User’s Guide (SPRU103)

Matlab is a registered trademark of The MathWorks, Inc.

8 Acknowledgments

DSPLIB includes code contributed by the following people:

Aaron Aboagye
Jeff Axelrod
Karen Baldwin
Philippe Cavalier
Pascal Dorster
Allison Frantz
Pedro Gelabert
Mike Hannah
Jeff Hayes
Natalie Messine
Jelena Nikolic
Greg Peake
Rosemarie Piedra
Cesar Ramirez
Alex Tessarolo
Carol Chow
Pierre Ponce
Julius Kusuma

Appendix A. Overview of Fractional Q Formats

Unless specifically noted, DSPLIB functions use Q15 format or to be more exact Q0.15. In a Qm.n format, there are m bits used to represent the twos complement integer portion of the number, and n bits used to represent the twos complement fractional portion. m+n+1 bits are needed to store a general Qm.n number. The extra bit is needed to store the sign of the number in the most-significant bit position. The representable integer range is specified by (-2^m,2^m) and the finest fractional resolution is 2^{-n}.

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

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 twos 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} = 2.441x10^4.

Table 1. Q3.12 Bit Fields

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

A.2 Q.15 Format

Q.15 format places the sign bit at the leftmost binary digit, and the next 15 leftmost bits contain the twos 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 x 10^{-5}.

Table 2. Q.15 Bit Fields

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

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 x 10^{-10}.

Table 3. Q.31 Low Memory Location Bit Fields

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

Table 4. Q.31 High Memory Location Bit Fields

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

Appendix B. 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 (1)

The resulting equation becomes:

Y = 1/(Xnorm*2^-n)

Or Y = 2^n/Xnorm (2)

where n = 1,2,3,...,14,15

Letting Ye = 2^n:
Ye = 2^n (3)

Substituting (3) into equation (2):
Y = Ye * 1/Xnorm (4)

Letting Ym = 1/Xnorm:
Ym = 1/Xnorm (5)

Substituting (5) into equation (4):
Y = Ye * Ym (6)

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

1 <= Ym < 2
-2 <= Ym <= -1 (7)

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 Ym(new) = exact value of 1/Xnorm:

Ym(new) = 1/Xnorm (c1)

or Ym(new)*X = 1 (c2)

Assume Ym(old) = estimate of value 1/X:

Ym(old)*Xnorm = 1 + Dyx

or Dxy = Ym(old)*Xnorm - 1 (c3)

where Dyx = error in calculation

Assume that Ym(new) and Ym(old) are related as follows:

Ym(new) = Ym(old) - Dy (c4)

where Dy = difference in values

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

Ym(old)*Xnorm = Ym(new)*Xnorm + Dxy

(Ym(new) + Dy)*Xnorm = Ym(new)*Xnorm + Dxy

Ym(new)*Xnorm + Dy*Xnorm = Ym(new)*Xnorm + Dxy

Dy*Xnorm = Dxy

Dy = Dxy * 1/Xnorm (c5)

Assume that 1/Xnorm is approximately equal to Ym(old):

Dy = Dxy * Ym(old) (approx) (c6)

Substituting (c6) into (c4):

Ym(new) = Ym(old) - Dxy*Ym(old) (c7)

Substituting for Dxy from (c3) into (c7):

Ym(new) = Ym(old) - (Ym(old)*Xnorm - 1)*Ym(old)

Ym(new) = Ym(old) - Ym(old)^2*Xnorm + Ym(old)

Ym(new) = 2*Ym(old) - Ym(old)^2*Xnorm (c8)

If after each calculation we equate Ym(old) to Ym(new):

Ym(old) = Ym(new) = Ym

Then equation (c8) evaluates to:

Ym = 2*Ym - Ym^2*Xnorm (c9)

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

The initial estimate can either be obtained from a look up table, or from choosing a mid-point, or simply from linear interpolation. The method chosen for this problem is the latter. This is simply accomplished by taking the complement of the least significant bits of the Xnorm value.

Appendix C. 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.
4. 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.