RM0020
Reference manual

SPC56xx and RPC56xx DSP function library 2

Introduction

The SPC56xx and RPC56xx DSP function library 2 contains optimized functions for Root part number 1 family of processors with Signal Processing Engine (SPE APU).

Contents

List of tables

1 Library functions

The library functions are presented in the following sections.

1.1 FFT

- **bitrev_table_16bit** - Bit Reversal Permutation for 16-bit data (frac16)
- **bitrev_table_32bit** - Bit Reversal Permutation for 32-bit data (frac32 and float)
- **fft_radix4_float** - N-point radix-4 complex to complex float in-place FFT, N is even power of two
- **fft_radix4_frac32** - N-point radix-4 complex to complex frac16/frac32 in-place FFT, N is even power of two
- **fft_quad_float** - N-point quad radix-2 complex to complex float in-place FFT, N is even power of two
- **fft_quad_frac32** - N-point quad radix-2 complex to complex frac16/frac32 in-place FFT, N is even power of two
- **fft_radix2_last_stage_float** - calculates the last stage of N-point radix-2 complex to complex float in-place FFT, together with **fft_quad_float** function is used to calculate N-point complex to complex float in-place FFT where N is odd power of two
- **fft_radix2_last_stage_frac32** - calculates the last stage of N-point radix-2 complex to complex frac32 in-place radix-2 FFT, together with **fft_quad_frac32** function is used to calculate N-point complex to complex in-place frac16/frac32 FFT where N is odd power of two
- **fft_real_split_float** - split function for N-point real to complex float FFT
- **fft_real_split_frac32** - split function for N-point real to complex frac32 FFT
- **window_apply_complex_float** - complex float windowing function
- **window_apply_complex_frac16** - complex frac16 windowing function
- **window_apply_real_float** - real float windowing function
- **window_apply_real_frac16** - real frac16 windowing function

1.2 Pressure sensing

- **mfb50_pmh** - calculates mass fraction burned (MFB), MFB50 value and PMH index
- **mfb50_index** - calculates MFB50 position index

1.3 Image

- **conv3x3** - 2-D convolution with 3x3 kernel
- **sobel3x3** - sobel filter
- **sobel3x3_horizontal** - horizontal sobel filter
- **sobel3x3_vertical** - vertical sobel filter

1.4 Filter

- **corr_frac16** - frac16 correlation function
- **fir_float** - float FIR filter
- **fir_frac16** - frac16 FIR filter
- **iir_float_1st** - float first-order IIR filter
- **iir_float_2nd** - float second-order IIR filter
- **iir_float_casc** - cascade of float second-order IIR filters
- **iir_frac16_1st** - frac16 first-order IIR filter
- **iir_frac16_2nd** - frac16 second-order IIR filter
- **iir_frac16_casc** - cascade of frac16 second-order IIR filters
- **iir_frac16_2nd_hc** - frac16 second-order IIR filter with half coefficients

1.5 Macros

The library defines macros in libdsp2.h that encapsulate some of the previous functions to allow easy usage; see the libdsp2.h file and the examples in example directory.

2 Supported compilers

The library was built and tested using the following compilers:

- CodeWarrior for PowerPC V1.5 beta2
- Green Hills MULTI for PowerPC v4.2.1
- Wind River Compiler Version 5.2.1.0

3 Directory structure

- **doc** - contains library user's manual
- **example** - contains library example projects for Axiom-0321 MPC5554 development board, for each supported compiler
- **include** - contains library header file libdsp2.h
- **src** - contains library source files for CodeWarrior compiler and C source files
- **src\src_GHS** - contains library source files for Green Hills compiler
- **src\src_WR** - contains library source files for Wind River compiler

4 How to use the library in a project

- CodeWarrior
  - add library source files with required functions into your project window
  - add path to library include file libdsp2.h to “Target Settings (Alt-F7)->Access Paths->User Paths”
  - include file libdsp2.h into your source file
- Green Hills
  - add library source files with required functions into your project window
  - use -I compiler option -I{path to libdsp2.h},
  - include file libdsp2.h into your source file
- Wind River
  - add library source files with required functions into your project window
  - use -I compiler option -I{path to libdsp2.h}
  - include file libdsp2.h into your source file

Code example:

```
#include "libdsp2.h"
#define N_MAX 512
float inout_buffer [2*N_MAX+8];    /* +8 to ensure there is 32 bytes
of readable memory behind buffer, must be double-word aligned */
void main(void)
{   /* compute complex to complex float in-place FFT */
     fft_quad_float_512(inout_buffer);
}
```

5 Example projects

The library contains ready to run example projects which are located in example directory. The projects are created for Axiom MPC5554 development board.

- CodeWarrior
  - CW_example.mcp - examples of float FFTs
  - CW_example1.mcp - examples of frac16/frac32 FFTs
  - CW_example2.mcp - examples of real to complex float FFTs
  - CW_example3.mcp - examples of real to complex frac16/frac32 FFTs
  - CW_example4.mcp - examples of real and complex float windowing functions
  - CW_example5.mcp - examples of real and complex frac16 windowing functions
  - CW_example6.mcp - examples of pressure sensing functions mfb50_pmh and mfb50_index
  - CW_example7.mcp - examples of 2D convolution with 3x3 kernel
  - CW_example8.mcp - examples of sobel filters
  - CW_example9.mcp - examples of frac16 data correlation and auto-correlation function
  - CW_example9a.mcp - examples of FIR filters
  - CW_example9b.mcp - examples of float data IIR filters
  - CW_example9c.mcp - examples of frac16 data IIR filters
- Green Hills
  - GHS_example.gpj - examples of float FFTs
  - GHS_example1.gpj - examples of frac16/frac32 FFTs
  - GHS_example2.gpj - examples of real to complex float FFTs
  - GHS_example3.gpj - examples of real to complex frac16/frac32 FFTs
  - GHS_example4.gpj - examples of real and complex float windowing functions
  - GHS_example5.gpj - examples of real and complex frac16 windowing functions
  - GHS_example6.gpj - examples of pressure sensing functions mfb50_pmh and mfb50_index
  - GHS_example7.gpj - examples of 2D convolution with 3x3 kernel
  - GHS_example8.gpj - examples of sobel filters
  - GHS_example9.gpj - examples of frac16 data correlation and auto-correlation function
  - GHS_example9a.gpj - examples of FIR filters
  - GHS_example9b.gpj - examples of float data IIR filters
  - GHS_example9c.gpj - examples of frac16 data IIR filters
- Wind River
  - makefile - examples of float FFTs
  - makefile1 - examples of frac16/frac32 FFTs
  - makefile2 - examples of real to complex float FFTs
  - makefile3 - examples of real to complex frac16/frac32 FFTs
  - makefile4 - examples of real and complex float windowing functions

- makefile5 - examples of real and complex frac16 windowing functions
- makefile6 - examples of pressure sensing functions mfb50_pmh and mfb50_index
- makefile7 - examples of 2D convolution with 3x3 kernel
- makefile8 - examples of sobel filters
- makefile9 - examples of frac16 data correlation and auto-correlation function
- makefile9a - examples of FIR filters
- makefile9b - examples of float data IIR filters
- makefile9c - examples of frac16 data IIR filters

6 Function API

6.1 Bit reversal permutation for 16-bit data

Function call:

```
void bitrev_table_16bit(unsigned int n, short *inout_buffer,
            unsigned short *seed_table);
```

Arguments:

Table 1. bitrev_table_16bit arguments

n	in	number of groups: for radix-2 bit reverse permutation: n = 8 for 64 and 128-point FFT n = 16 for 256 and 512-point FFT n = 32 for 1024 and 2048 point FFT n = 64 for 4096 point FFT for radix-4 2-bit reverse permutation: n = 4 for 64-point FFT n = 16 for 256 and 1024-point FFT n = 64 for 4096 point FFT
inout_buffer	in/out	pointer to the input/output vector of length 2*N, vector must be word aligned memory layout: x_Re(0) x_Im(0) x_Re(1) x_Im(1) ... x_Re(N-1) x_Im(N-1) (Re - real part, Im - imaginary part)
seed_table	in	pointer to the seed table, use one of the predefined tables: seed_radix2_16bit_64 seed_radix2_16bit_128 seed_radix2_16bit_256 seed_radix2_16bit_512 seed_radix2_16bit_1024 seed_radix2_16bit_2048 seed_radix2_16bit_4096 seed_radix4_16bit_64 seed_radix4_16bit_256 seed_radix4_16bit_1024 seed_radix4_16bit_4096

Description: Implements fast bit reversal permutation on 16-bit complex data using seed table as described in David M.W.Evans, *An Improved Digit-Reversal Permutation Algorithm for the Fast Fourier and Hartley Transforms* IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. Assp-35, No. 8, August 1987.

Performance: See *Chapter 7: Performance*.

Example 1. bitrev_table_16bit

```
#include "libdsp2.h"
int inout_buffer[2*256];
void main() {
    /* bit reverse permutation and calculation of 256-point FFT */
    bitrev_table_16bit(16, (short *)(inout_buffer+256),
seed_radix2_16bit_256);
    fft_quad_frac32(256, inout_buffer, w_table_radix2_frac32_256);}
```

6.2 Bit reversal permutation for 32-bit data

Function call:

```
void bitrev_table_32bit(unsigned int n, float *inout_buffer,
            unsigned short *seed_table);
```

Arguments:

Table 2. bitrev_table_32bit arguments

n	in	number of groups: for radix-2 bit reverse permutation: n = 8 for 64 and 128-point FFT n = 16 for 256 and 512-point FFT n = 32 for 1024 and 2048 point FFT n = 64 for 4096 point FFT for radix-4 2-bit reverse permutation: n = 4 for 64-point FFT n = 16 for 256 and 1024-point FFT n = 64 for 4096 point FFT
inout_buffer	in/out	pointer to the input/output vector of length 2*N, vector must be double word aligned memory layout: x_Re(0) x_Im(0) x_Re(1) x_Im(1) ... x_Re(N-1) x_Im(N-1) (Re - real part, Im - imaginary part)
seed_table	in	pointer to the seed table, use one of the predefined tables: seed_radix2_32bit_64 seed_radix2_32bit_128 seed_radix2_32bit_256 seed_radix2_32bit_512 seed_radix2_32bit_1024 seed_radix2_32bit_2048 seed_radix2_32bit_4096 seed_radix4_32bit_64 seed_radix4_32bit_256 seed_radix4_32bit_1024 seed_radix4_32bit_4096

Description: Implements fast bit reversal permutation on 32-bit complex data using seed table as described in David M.W.Evans, *An Improved Digit-Reversal Permutation Algorithm for the Fast Fourier and Hartley Transforms* IEEE Transactions on Acoustics, Speech, and Signal Processing, Vol. Assp-35, No. 8, August 1987.

Performance: See *Chapter 7: Performance*.

Example 2. bitrev_table_32bit

```
#include "libdsp2.h"
float inout_buffer[2*256];
void main() {
    /* bit reverse permutation and calculation of 256-point FFT */
    bitrev_table_32bit(16, inout_buffer, seed_radix2_32bit_256);
    fft_quad_float(256, inout_buffer, w_table_radix2_float_256);
}
```

6.3 N-point radix-4 complex to complex float in-place FFT

Function call:

```
void fft_radix4_float(unsigned int N, float *inout_buffer,
            float *twiddle_factor_table);
```

Arguments:

Table 3. fft_radix4_float arguments

N	in	FFT length, must be even power of two, tested for 64, 256, 1024, 4096
inout_buffer	in/out	pointer to the input/output vector of length 2*N, vector must be double word aligned memory layout: x_Re(0) x_Im(0) x_Re(1) x_Im(1) ... x_Re(N-1) x_Im(N-1) (Re - real part, Im - imaginary part)
twiddle_factor_table	in	pointer to the vector of twiddle factors of length 3*N/2-4, vector must be double word aligned memory layout: w_Re(0) w_Im(0) w_Re(1) w_Im(1) ... w_Re(3*N/4-3) w_Im(3*N/4-3) Use predefined w_table_radix4_float_N arrays. The w_table_radix4_float_N is computed according to this formula: $w\_Re(k) + i \cdot w\_Im(k) = w_N^k$ for k = 0, 1, ... 3*N/4-3

Description: Computes the N-point radix-4 complex to complex float in-place fast Fourier transform (FFT). Input 2-bit reversed data are stored in inout_buffer, output data are written over the input data into inout_buffer. There is used radix-4 FFT algorithm.

Algorithm:

Equation 1

$$X(n) = \sum_{k=0}^{N-1} x(k) \cdot w_N^{n \cdot k} \quad \text{for each n from 0 to N-1}$$

Equation 2

$$w_N = e^{-\frac{2\pi i}{N}}$$

where i is the imaginary unit with the property that: $i^2 = -1$.

Note: There must be at least 32 bytes of readable memory behind inout_buffer.

Performance: See *Chapter 7: Performance* and *Table 32*.

Example 3. fft_radix4_float

```
#include "libdsp2.h"
float inout_buffer[2*256];
void main() {
    /* bit reverse permutation and calculation of 256-point FFT */
    bitrev_table_32bit(16, inout_buffer, seed_radix4_32bit_256);
    fft_radix4_float (256, inout_buffer, w_table_radix4_float_256);
}
```

6.4 N-point radix-4 complex to complex frac16/frac32 in-place FFT

Function call:

```
void fft_radix4_frac32(unsigned int N, int *inout_buffer,
            int *twiddle_factor_table);
```

Arguments:

Table 4. fft_radix4_frac32 arguments

N	in	FFT length, must be even power of two, tested for 64, 256, 1024, 4096
inout_buffer	in/out	pointer to the input/output vector of length 2*N, vector must be double word aligned input data alignment: 16-bit fractional input data in range -1 to 1-2^-15 are stored in second half of inout_buffer in order real, imag, real imag, ... 0, 0, 0, 0, ... x_Re(0) x_Im(0), x_Re(1) x_Im(1), ... x_Re(N-1) x_Im(N-1), output data alignment: 32-bit fractional output data in range -1 to 1-2^-31 are stored: X_Re(0) X_Im(0) X_Re(1) X_Im(1) ... X_Re(N-1) X_Im(N-1) (Re - real part, Im - imaginary part)

Table 4. fft_radix4_frac32 arguments (continued)

twiddle_factor_table	in	pointer to the vector of twiddle factors of length 3*N/2-4, vector must be double word aligned memory layout: w_Re(0) w_Im(0) w_Re(1) w_Im(1) ... w_Re(3*N/4-3) w_Im(3*N/4-3) Use predefined w_table_radix4_frac32_N arrays. The w_table_radix4_frac32_N is computed according to this formula: $w\_Re(k) + i \cdot w\_Im(k) = w_N^k$ for k = 0, 1, ... 3*N/4-3

Description: Computes the N-point radix-4 complex to complex frac16/frac32 in-place fast Fourier transform (FFT). Input 2-bit reversed 16-bit fractional data are stored in second half of inout_buffer, output 32-bit fractional data are written over the input data into inout_buffer. There is used radix-4 FFT algorithm.

The output scaling is configurable by constant definition in the function source file:

```
.set    SCALING                 1    #/* 0 - off, 1 - on */
```

By default it is turned on. The scaling is performed by dividing by 4 before each radix-4 stage.

Algorithm:

Equation 3

$$X(n) = \sum_{k=0}^{N-1} x(k) \cdot w_N^{n \cdot k}$$

for each n from 0 to N-1, scaled down by N if scaling is turned on

Equation 4

$$w_N = e^{-\frac{2\pi i}{N}}$$

where i is the imaginary unit with the property that: $i^2 = -1$.

Note: There must be at least 32 bytes of readable memory behind inout_buffer.

Performance: See *Chapter 7: Performance* and *Table 35*.

Example 4. fft_radix4_frac32

```
#include "libdsp2.h"
int inout_buffer[2*256];
void main() {
    /* bit reverse permutation and calculation of 256-point FFT */
    bitrev_table_16bit(16, (short *)(inout_buffer+256),
seed_radix4_16bit_256);
```

```
    fft_radix4_frac32(256, inout_buffer,
w_table_radix4_frac32_256);
}
```

6.5 N-point quad radix-2 complex to complex float in-place FFT

Function call:

```
void fft_quad_float(unsigned int N, float *inout_buffer,
            float *twiddle_factor_table);
```

Arguments:

Table 5. fft_quad_float arguments

N	in	FFT length, must be even power of two, tested for 64, 256, 1024, 4096
inout_buffer	in/out	pointer to the input/output vector of length 2*N, vector must be double word aligned memory layout: x_Re(0) x_Im(0) x_Re(1) x_Im(1) ... x_Re(N-1) x_Im(N-1) (Re - real part, Im - imaginary part)
twiddle_factor_table	in	pointer to the vector of twiddle factors of length N, vector must be double word aligned memory layout: w_Re(0) w_Im(0) w_Re(1) w_Im(1) ... w_Re(N/2-1) w_Im(N/2-1) Use predefined w_table_radix2_float_N arrays. The w_table_radix2_float_N is computed according to this formula: $w\_Re(k) + i \cdot w\_Im(k) = w_N^k$ for k = 0, 1, ... N/2-1

Description: Computes the N-point quad radix-2 complex to complex float in-place fast Fourier transform (FFT). Input bit reversed data are stored in inout_buffer, output data are written over the input data into inout_buffer. There is used radix-2 algorithm in which two stages are calculated in one loop (quad butterfly).

Algorithm:

Equation 5

$$X(n) = \sum_{k=0}^{N-1} x(k) \cdot w_N^{n \cdot k}$$ for each n from 0 to N-1

Equation 6

$$w_N = e^{-\frac{2\pi i}{N}}$$

where i is the imaginary unit with the property that: $i^2 = -1$.

Note: There must be at least 32 bytes of readable memory behind inout_buffer.

Performance: See *Section 7* and *Table 33*.

Example 5. fft_quad_float

```
#include "libdsp2.h"
float inout_buffer[2*256];
void main() {
    /* bit reverse permutation and calculation of 256-point FFT */
    bitrev_table_32bit(16, inout_buffer, seed_radix2_32bit_256);
    fft_quad_float(256, inout_buffer, w_table_radix2_float_256);
}
```

6.6 N-point quad radix-2 complex to complex frac16/frac32 in-place FFT

Function call:

```
void fft_quad_frac32(unsigned int N, int *inout_buffer,
            int *twiddle_factor_table);
```

Arguments:

Table 6. fft_quad_frac32 arguments

N	in	FFT length, must be even power of two, tested for 64, 256, 1024, 4096
inout_buffer	in/out	pointer to the input/output vector of length 2*N, vector must be double word aligned input data alignment: 16-bit fractional input data in range -1 to 1-2^-15 are stored in second half of inout_buffer in order real, imag, real imag, ... 0, 0, 0, 0, ... x_Re(0) x_Im(0), x_Re(1) x_Im(1), ... x_Re(N-1) x_Im(N-1), output data alignment: 32-bit fractional output data in range -1 to 1-2^-31 are stored: X_Re(0) X_Im(0) X_Re(1) X_Im(1) ... X_Re(N-1) X_Im(N-1) (Re - real part, Im - imaginary part)
twiddle_factor_table	in	pointer to the vector of twiddle factors of length N, vector must be double word aligned memory layout: w_Re(0) w_Im(0) w_Re(1) w_Im(1) ... w_Re(N/2-1) w_Im(N/2-1) Use predefined w_table_radix2_frac32_N arrays. The w_table_radix2_frac32_N is computed according to this formula: $w\_Re(k) + i \cdot w\_Im(k) = w_N^k$ for k = 0, 1, ... N/2-1

Description: Computes the N-point quad radix-2 complex to complex frac16/frac32 in-place fast Fourier transform (FFT). Input bit reversed 16-bit fractional data are stored in second half of inout_buffer, output 32-bit fractional data are written over the input data into inout_buffer. There is used radix-2 algorithm in which two stages are calculated in one loop (quad butterfly).

The output scaling is configurable by constant definition in the function source file:

```
.set    SCALING                 1    #/* 0 - off, 1 - on */
```

By default it is turned on. The scaling is performed by dividing by 4 before each pair of radix-2 stages.

Algorithm:

Equation 7

$$X(n) = \sum_{k=0}^{N-1} x(k) \cdot w_N^{n \cdot k}$$

for each n from 0 to N-1, scaled down by N if scaling is turned on

Equation 8

$$w_N = e^{-\frac{2\pi i}{N}}$$

where i is the imaginary unit with the property that: $i^2 = -1$.

Note: There must be at least 16 bytes of readable memory behind inout_buffer.

Performance: See *Chapter 7: Performance* and *Table 36*.

Example 6. fft_quad_frac32

```
#include "libdsp2.h"
int inout_buffer[2*256];
void main() {
    /* bit reverse permutation and calculation of 256-point FFT */
    bitrev_table_16bit(16, (short *)(inout_buffer+256),
seed_radix2_16bit_256);
    fft_quad_frac32(256, inout_buffer, w_table_radix2_frac32_256);
}
```

6.7 Last stage of N-point radix-2 complex to complex float in-place FFT

Function call:

```
void fft_radix2_last_stage_float(unsigned int N, float
*inout_buffer,
            float *twiddle_factor_table);
```

Arguments:

Table 7. fft_radix2_last_stage_float arguments

N	in	FFT length, tested for 128, 512, 2048
inout_buffer	in/out	pointer to the input/output vector of length 2*N, vector must be double word aligned memory layout: x_Re(0) x_Im(0) x_Re(1) x_Im(1) ... x_Re(N-1) x_Im(N-1) (Re - real part, Im - imaginary part)
twiddle_factor_table	in	pointer to the vector of twiddle factors of length N, vector must be double word aligned memory layout: w_Re(0) w_Im(0) w_Re(1) w_Im(1) ... w_Re(N/2-1) w_Im(N/2-1) Use predefined w_table_radix2_float_N arrays. The w_table_radix2_float_N is computed according to this formula: $w\_Re(k) + i \cdot w\_Im(k) = w_N^k$ for k = 0, 1, ... N/2-1

Description: Computes the last stage of N-point radix-2 complex to complex float in-place fast Fourier transform (FFT). Input data are stored in inout_buffer, output data are written over the input data into inout_buffer. This function is used to calculate the FFT for lengths which are odd power of two (128, 512, 2048).

Note: There must be at least 16 bytes of readable memory behind inout_buffer.

Performance: See *Chapter 7: Performance* and *Table 34*.

Example 7. fft_radix2_last_stage_float

```
#include "libdsp2.h"
float inout_buffer[2*512];
void main() {
    /* bit reverse permutation and calculation of 512-point FFT */
    bitrev_table_32bit(16, inout_buffer, seed_radix2_32bit_512);
    fft_quad_float(512, inout_buffer, w_table_radix2_float_512);
    fft_radix2_last_stage_float(512, inout_buffer,
w_table_radix2_float_512);
}
```

6.8 Last stage of N-point radix-2 complex to complex frac32 in-place FFT

Function call:

```
void fft_radix2_last_stage_frac32(unsigned int N, int
*inout_buffer,
            int *twiddle_factor_table);
```

Arguments:

Table 8. fft_radix2_last_stage_frac32 arguments

N	in	FFT length, tested for 128, 512, 2048
inout_buffer	in/out	pointer to the input/output vector of length 2*N, vector must be double word aligned memory layout: x_Re(0) x_Im(0) x_Re(1) x_Im(1) ... x_Re(N-1) x_Im(N-1) (Re - real part, Im - imaginary part)
twiddle_factor_table	in	pointer to the vector of twiddle factors of length N, vector must be double word aligned memory layout: w_Re(0) w_Im(0) w_Re(1) w_Im(1) ... w_Re(N/2-1) w_Im(N/2-1) Use predefined w_table_radix2_frac32_N arrays. The w_table_radix2_frac32_N is computed according to this formula: $w\_Re(k) + i \cdot w\_Im(k) = w_N^k$ for k = 0, 1, ... N/2-1

Description: Computes the last stage of N-point radix-2 complex to complex frac32 in-place fast Fourier transform (FFT). Input data are stored in inout_buffer, output data are written over the input data into inout_buffer. This function is used to calculate the FFT for lengths which are odd power of two (128, 512, 2048).

Note: There must be at least 16 bytes of readable memory behind inout_buffer.

Performance: See *Chapter 7: Performance* and *Table 37*.

Example 8. fft_radix2_last_stage_frac32

```
#include "libdsp2.h"
int inout_buffer[2*512];
void main() {
    /* bit reverse permutation and calculation of 512-point FFT */
    bitrev_table_16bit(16, (short *)(inout_buffer+512),
seed_radix2_16bit_512);
    fft_quad_frac32(512, inout_buffer, w_table_radix2_frac32_512);
    fft_radix2_last_stage_frac32(512, inout_buffer,
w_table_radix2_frac32_512);
}
```

6.9 Split function of N-point real to complex float FFT

Function call:

```
void fft_real_split_float(unsigned int N, float *y, float *wa,
              float *wb);
```

Arguments:

Table 9. fft_real_split_float arguments

N	in	FFT length, tested for 128, 256, 512, 1024, 2048, 4096
y	in/out	pointer to the input/output vector of length N, vector must be double word aligned input: as input pass the output from the complex to complex float FFT of half the length, data ordered x_Re(0) x_Im(0) x_Re(1) x_Im(1)... x_Re(N/2-1) x_Im(N/2-1) output memory layout: X_Re(0) {0 or X_Re(N/2)} X_Re(1) X_Im(1) ... X_Re(N/2-1) X_Im(N/2-1) (Re - real part, Im - imaginary part)
wa	in	pointer to the wa vector of twiddle factors of length N/4+1, vector must be double word aligned memory layout: wa_Re(0) wa_Im(0) wa_Re(1) wa_Im(1) ... wa_Re(N/4+1) wa_Im(N/4+1) Use predefined wa_table_float_N arrays. The wa_table_float_N is computed according to this formula: $wa(k) = \frac{1}{2}\left(1 - iW_N^k\right)$ for k = 0, 1, ... N/4+1 $W_N = e^{-i\frac{2\pi}{N}}$, where i is the imaginary unit
wb	in	pointer to the wb vector of twiddle factors of length N/4+1, vector must be double word aligned memory layout: wb_Re(0) wb_Im(0) wb_Re(1) wb_Im(1) ... wb_Re(N/4+1) wb_Im(N/4+1) Use predefined wb_table_float_N arrays. The wb_table_float_N is computed according to this formula: $wb(k) = \frac{1}{2}\left(1 + iW_N^k\right)$ for k = 0, 1, ... N/4+1 $W_N = e^{-i\frac{2\pi}{N}}$, where i is the imaginary unit

Description: Split function of N-point real to complex float in-place fast Fourier transform (FFT). Input data are stored in y, output data are written over the input data into y. The function gives the first half of the output spectrum.

The calculation of the N/2 item is configurable by constant definition in the function source file:

```
.set    N2_REALPART,          0    #/* 0/1 - real part of N/2 item
isn't/is inserted in imaginary part of 0th item, default value is 0
*/
```

By default the N/2 output item is not calculated.

Here is the summary how the real to complex float in-place FFT can be calculated using the split function:

– on real input data in_Re(0) in_Re(1) ... in_Re(N-1), calculate complex to complex FFT of half the length
– call the split function which calculates the half of the spectrum X_Re(0) X_Im(0) X_Re(1) X_Im(1) ... X_Re(N/2-1) X_Im(N/2-1)
– if N2_REALPART is defined to 1, the split function saves the X_Re(N/2) into X_Im(0)

Algorithm:

Input data in memory:

$$x\_Re(0)x\_Im(0)x\_Re(1)x\_Im(1)\ldots x\_Re\left(\frac{N}{2}-1\right)x\_Im\left(\frac{N}{2}-1\right)$$

where

Re is the real part,

Im is the imaginary part

Equation 9

$$x(k) = x\_Re(k) + ix\_Im(k)$$

The split function performs calculation of the output sequence $X(k)$ from the input sequence $x(k)$ according to the following equations:

Equation 10

$$X(0) = x\_Re(0) + x\_Im(0)$$

Equation 11

$$X\left(\frac{N}{2}\right) = x\_Re(0) - x\_Im(0)$$

Equation 12

$$X(k) = wa(k)x(k) + wb(k)x^{*}\left(\frac{N}{2}-k\right)$$

for each k from 1 to N/2-1

where:

Equation 13

$$wa(k) = \frac{1}{2}\left(1 - iW_N^{k}\right)$$

Equation 14

$$wb(k) = \frac{1}{2}\left(1 + iW_N^{k}\right)$$

Equation 15

$$W_N = e^{-i\frac{2\pi}{N}}$$

* denotes complex conjugate operation

i is the imaginary unit

Equation 16

$$X(k) = X\_Re(k) + iX\_Im(k)$$

output data in memory:

$$X\_Re(0)X\_Im(0)X\_Re(1)X\_Im(1)\ldots X\_Re\left(\frac{N}{2}-1\right)X\_Im\left(\frac{N}{2}-1\right)$$

when

N2_REALPART is defined to 0

$$X\_Re(0)X\_Re\left(\frac{N}{2}\right)X\_Re(1)X\_Im(1)\ldots X\_Re\left(\frac{N}{2}-1\right)X\_Im\left(\frac{N}{2}-1\right)$$

when

N2_REALPART is defined to 1

Performance: See *Chapter 7: Performance* and *Table 38*.

Example 9. fft_real_split_float

```
#include "libdsp2.h"
float inout_buffer [128+8];    /* +8 to ensure there is 32 bytes of
readable memory behind buffer */

void main(void)
{
```

```
    int i;

    /* example real input data, sequence from 1 to 128 */
    for (i = 0; i < 128; i++) inout_buffer[i] = i+1;
    /* compute complex to complex FFT of half the length */
    fft_radix4_float_64(inout_buffer);
    /* call split function to get correct FFT result */
    fft_real_split_float(128, inout_buffer, wa_table_float_128,
wb_table_float_128);
}
```

6.10 Split function of N-point real to complex frac32 FFT

Function call:

```
void fft_real_split_frac32(unsigned int N, int *y, int *wa, int
*wb);
```

Arguments:

Table 10. fft_real_split_frac32 arguments

N	in	FFT length, tested for 128, 256, 512, 1024, 2048, 4096
y	in/out	pointer to the input/output vector of length N, vector must be double word aligned input: as input pass the output from the complex to complex frac32 FFT of half the length, data ordered x_Re(0) x_Im(0) x_Re(1) x_Im(1)... x_Re(N/2-1) x_Im(N/2-1) output memory layout: X_Re(0) {0 or X_Re(N/2)} X_Re(1) X_Im(1) ... X_Re(N/2-1) X_Im(N/2-1) (Re - real part, Im - imaginary part)
wa	in	pointer to the wa vector of twiddle factors of length N/4+1, vector must be double word aligned memory layout: wa_Re(0) wa_Im(0) wa_Re(1) wa_Im(1) ... wa_Re(N/4+1) wa_Im(N/4+1) Use predefined wa_table_frac32_N arrays. The wa_table_frac32_N is computed according to this formula: $wa(k) = \frac{1}{2}\left(1 - iW_N^k\right)$ for k = 0, 1, ... N/4+1 $W_N = e^{-i\frac{2\pi}{N}}$, where i is the imaginary unit

Table 10. fft_real_split_frac32 arguments (continued)

wb	in	pointer to the wb vector of twiddle factors of length N/4+1, vector must be double word aligned memory layout: wb_Re(0) wb_Im(0) wb_Re(1) wb_Im(1) ... wb_Re(N/4+1) wb_Im(N/4+1) Use predefined wb_table_frac32_N arrays. The wb_table_frac32_N is computed according to this formula: $wb(k) = \frac{1}{2}\left(1 + iW_N^k\right)$ for k = 0, 1, ... N/4+1 $W_N = e^{-i\frac{2\pi}{N}}$, where i is the imaginary unit

Description: Split function of N-point real to complex frac32 in-place fast Fourier transform (FFT). Input 32-bit fractional data are stored in y, output 32-bit fractional data are written over the input data into y. The function gives the first half of the output spectrum.

The calculation of the N/2 item is configurable by constant definition in the function source file:

```
.set    N2_REALPART,          0    #/* 0/1 - real part of N/2 item
isn't/is inserted in imaginary part of 0th item, default value is 0
*/
```

By default the N/2 output item is not calculated.

The output scaling is configurable by constant definition in the function source file:

```
.set    SCALING               1    #/* 0 - off, 1 - on */
```

By default it is turned on. The scaling is performed by dividing input data by 2 before calculation.

Here is the summary how the real to complex frac32 in-place FFT can be calculated using the split function:

– on real input data in_Re(0) in_Re(1) ... in_Re(N-1), calculate complex to complex FFT of half the length
– call the split function which calculates the half of the spectrum X_Re(0) X_Im(0) X_Re(1) X_Im(1) ... X_Re(N/2-1) X_Im(N/2-1)
– if N2_REALPART is defined to 1, the split function saves the X_Re(N/2) into X_Im(0)
– note that the FFT output is divided by N since scaling is by default turned on

Algorithm:

Input data in memory:

,

$$x\_Re(0)x\_Im(0)x\_Re(1)x\_Im(1)\ldots x\_Re\left(\frac{N}{2}-1\right)x\_Im\left(\frac{N}{2}-1\right)$$

where

Re is the real part,

Im is the imaginary part

Equation 17,

$$x(k) = \frac{x\_Re(k)}{2} + i\frac{x\_Im(k)}{2}$$

divided by two only in the case when

SCALING is set to 1

The split function performs calculation of the output sequence $X(k)$ from the input sequence $x(k)$ according to the following equations:

Equation 18

$$X(0) = x\_Re(0) + x\_Im(0)$$

Equation 19

$$X\left(\frac{N}{2}\right) = x\_Re(0) - x\_Im(0)$$

Equation 20

$$X(k) = wa(k)x(k) + wb(k)x^{*}\left(\frac{N}{2} - k\right)$$

for each k from 1 to N/2-1 where:

Equation 21

$$wa(k) = \frac{1}{2}\left(1 - iW_N^{k}\right)$$

Equation 22

$$wb(k) = \frac{1}{2}\left(1 + iW_N^{k}\right)$$

Equation 23

$$W_N = e^{-i\frac{2\pi}{N}}$$

* denotes complex conjugate operation

i is the imaginary unit

Equation 24

$$X(k) = X\_Re(k) + iX\_Im(k)$$

output data in memory:

$$X\_Re(0)X\_Im(0)X\_Re(1)X\_Im(1)\ldots X\_Re\left(\frac{N}{2}-1\right)X\_Im\left(\frac{N}{2}-1\right)$$

when

N2_REALPART is defined to 0

$$X\_Re(0)X\_Re\left(\frac{N}{2}\right)X\_Re(1)X\_Im(1)\ldots X\_Re\left(\frac{N}{2}-1\right)X\_Im\left(\frac{N}{2}-1\right)$$

when

N2_REALPART is defined to 1

Performance: See *Chapter 7: Performance* and *Table 40*.

Example 10. fft_real_split_frac32

```
#include "libdsp2.h"
int inout_buffer [128+8];    /* +8 to ensure there is 32 bytes of
readable memory behind buffer */

void main(void)
{
    int i;

    /* example real input data, sequence from 1/2^15 to 128/2^15 */
    for (i = 0; i < 128; i++) *((short *)(inout_buffer + 64)+i) =
(short) (i+1);
    /* compute complex to complex FFT of half the length */
    fft_radix4_frac32_64(inout_buffer);
    /* call split function to get correct FFT result divided by N */
    fft_real_split_frac32(128, inout_buffer, wa_table_frac32_128,
wb_table_frac32_128);
}
```

6.11 Complex float windowing function

Function call:

```
void window_apply_complex_float(unsigned int N, float *x, float *y,
                float *w);
```

Arguments:

Table 11. window_apply_complex_float arguments

N	in	length of complex vectors, must be multiple of 4
x(1)	in	pointer to the input vector of length 2*N
y(1)	out	pointer to the output vector of length 2*N, y vector can be the same as x vector (in-place windowing).
w(1)	in	pointer to the window vector of length 2*N

1. All vectors must be double word aligned. All vectors have the following memory layout: *re*(0) *im*(0) *re*(1) *im*(1) ... *re*(N-1) *im*(N-1), where *re* is real part and *im* is imaginary part.

Description: Windowing function for complex float data. Input data are stored in x vector, output data are written into y vector. The y vector can be the same as x vector (in-place windowing).

Algorithm:

Equation 25

$$y(k) = x(k) \times w(k)$$

for each k = 0,1,...N-1

where x denotes complex multiplication.

Note: There must be 16 bytes of readable memory behind x and w buffers.

Performance: See *Chapter 7: Performance* and *Table 42*.

Example 11. window_apply_complex_float

```
#include "libdsp2.h"
float x[2*64+4];    /* +4 to ensure there is 16 bytes of readable
memory behind buffer, must be aligned to 8 bytes */
float w[2*64+4];    /* +4 to ensure there is 16 bytes of readable
memory behind buffer, must be aligned to 8 bytes */

void main(void)
{
    unsigned int i, N;

    /* length of complex vectors is 64 */
    N = 64;
    /* prepare example input data */
    for (i = 0; i < 2*N; i++) x[i] = i+1;
```

```
    for (i = 0; i < 2*N; i++) w[i] = i+2*N;
    /* windowing function on complex float data, x[i] = x[i]*w[i] */
    window_apply_complex_float(N, x, x, w);
}
```

6.12 Complex frac16 windowing function

Function call:

```
void window_apply_complex_frac16(unsigned int N, short *x, short *y,
                short *w);
```

Arguments:

Table 12. window_apply_complex_frac16 arguments

N	in	length of complex vectors, must be multiple of 4
x(1)	in	pointer to the input vector of length 2*N
y(1)	out	pointer to the output vector of length 2*N, y vector can be the same as x vector (in-place windowing).
w(1)	in	pointer to the window vector of length 2*N

1. All vectors must be word aligned. All vectors have the following memory layout: *re*(0) *im*(0) *re*(1) *im*(1)... *re*(N-1) *im*(N-1), where *re* is real part and *im* is imaginary part.

Description: Windowing function for complex frac16 data. All the data are handled as signed fractional 16-bit (range -1 to 1-2^-15). Input data are stored in x vector, output data are written into y vector. The y vector can be the same as x vector (in-place windowing).

The output scaling is configurable by constant definition in the function source file:

```
.set    SCALING,            1    #/* 0 - off, 1 - on (output divided
by 2) */
```

By default it is turned on. The scaling is performed by dividing output data by 2.

Algorithm:

Equation 26

$$y(k) = x(k) \times w(k)$$

for each k = 0,1,...N-1

where x denotes complex multiplication. The output value $y(k)$ is divided by 2 when SCALING == 1.

Note: There must be 8 bytes of readable memory behind x and w buffers.

If SCALING == 1, then w array mustn't have any value equal to -1 (-32768) since there is used integer multiplication for fractional data.

Performance: See *Chapter 7: Performance* and *Table 43*.

Example 12. window_apply_complex_frac16

```
#include "libdsp2.h"
short x[2*64+4];    /* +4 to ensure there is 8 bytes of readable
memory behind buffer, must be aligned to 4 bytes */
short w[2*64+4];    /* +4 to ensure there is 8 bytes of readable
memory behind buffer, must be aligned to 4 bytes */
void main(void)
{
    unsigned int i, N;

    /* length of complex vectors is 64 */
    N = 64;
    /* prepare example input data */
    for (i = 0; i < 2*N; i++) x[i] = (i+1)<<5;
    for (i = 0; i < 2*N; i++) w[i] = (i+2*N)<<5;
    /* windowing function on complex frac16 data, x[i] = x[i]*w[i]
*/
    window_apply_complex_frac16(N, x, x, w);
}
```

6.13 Real float windowing function

Function call:

```
void window_apply_real_float(unsigned int N, float *x, float *y,
                float *w);
```

Arguments:

Table 13. window_apply_real_float arguments

N	in	length of vectors, must be multiple of 16
$x^{(1)}$	in	pointer to the input vector of length N
$y^{(1)}$	out	pointer to the output vector of length N, y vector can be the same as x vector (in-place windowing)
$w^{(1)}$	in	pointer to the window vector of length N

1. All vectors must be double word aligned. All vectors have the following memory layout: *x*(0) *x*(1)... *x*(N-1).

Description: Windowing function for real float data. Input data are stored in x vector, output data are written into y vector. The y vector can be the same as x vector (in-place windowing).

Algorithm:

Equation 27

$$y(k) = x(k)w(k)$$

for each k = 0,1,...N-1

Note: There must be 16 bytes of readable memory behind input vector x and 8 bytes of readable memory behind window vector w.

Performance: See *Chapter 7: Performance* and *Table 44*.

Example 13. window_apply_real_float

```
#include "libdsp2.h"
float x[64+4];    /* +4 to ensure there is 16 bytes of readable
memory behind buffer */
float w[64+2];    /* +2 to ensure there is  8 bytes of readable
memory behind buffer */

void main(void)
{
    unsigned int i, N;

    /* length of vectors is 64 */
    N = 64;
    /* prepare example input data */
    for (i = 0; i < N; i++) x[i] = i+1;
    for (i = 0; i < N; i++) w[i] = i+N;
    /* windowing function on real float data, x[i] = x[i]*w[i] */
    window_apply_real_float(N, x, x, w);
}
```

6.14 Real frac16 windowing function

Function call:

```
void window_apply_real_frac16(unsigned int N, short *x, short *y,
                short *w);
```

Arguments:

Table 14. window_apply_real_frac16 arguments

N	in	length of complex vectors, must be multiple of 16
x(1)	in	pointer to the input vector of length N
y(1)	out	pointer to the output vector of length N, y vector can be the same as x vector (in-place windowing)
w(1)	in	pointer to the window vector of length N

1. All vectors must be word aligned. All vectors have the following memory layout: *x*(0) *x*(1)... *x*(N-1).

Description: Windowing function for real frac16 data. All the data are handled as signed fractional 16-bit (range -1 to 1-2^-15). Input data are stored in x vector, output data are written into y vector. The y vector can be the same as x vector (in-place windowing).

Algorithm:

Equation 28

$$y(k) = x(k)w(k)$$

for each k = 0,1,...N-1

Note: There must be 16 bytes of readable memory behind input vector x and 8 bytes of readable memory behind window vector w.

Performance: See *Chapter 7: Performance* and *Table 45*.

Example 14. window_apply_real_frac16

```
#include "libdsp2.h"
short x[64+4];    /* +4 to ensure there is 8 bytes of readable
memory behind buffer, must be aligned to 4 bytes */
short w[64+2];    /* +2 to ensure there is 4 bytes of readable
memory behind buffer, must be aligned to 4 bytes */

void main(void)
{
    unsigned int i, N;

    /* length of vectors is 64 */
    N = 64;
    /* prepare example input data */
    for (i = 0; i < N; i++) x[i] = (i+1)<<5;
    for (i = 0; i < N; i++) w[i] = (i+N)<<5;
    /* windowing function on real float data, x[i] = x[i]*w[i] */
    window_apply_real_frac16(N, x, x, w);
}
```

6.15 mfb50_pmh - pressure sensing function 1

Function call:

```
float mfb50_pmh(unsigned int N, float *P_MFB, float *V, float K,
float *PMH);
```

Arguments:

Table 15. mfb50_pmh arguments

N	in	length of pressure vector P_MFB and volume vector V, N-2 must be multiple of 8, N must be greater than or equal to 18
P_MFB(1)	in/out	pointer to the input pressure vector of length N, input memory layout is P(0) P(1) ... P(N-1), output memory layout is MFB(1) MFB(2) ... MFB(N-2) MFB(N-1) 0
V(1)	in	pointer to the input volume vector of length N, memory layout is V(0) V(1) ... V(N-1)

Table 15. mfb50_pmh arguments (continued)

K	in	constant related to engine setup
PMH	out	pointer to variable where PMH index is returned

1. Both pressure and volume vector must be double word aligned.

Description: Pressure sensing function 1. The function is used for in-cylinder pressure sensor management. For each cylinder and every engine revolution, there is calculated the mass fraction burned function (MFB), MFB50 value and PMH torque index. As inputs there are used vector of in-cylinder pressure values P_MFB, vector of cylinder volume values V, and constant K. The function returns the MFB50 value, mass fraction burned function (MFB) is returned in P_MFB vector and PMH index in a variable addresses by PMH pointer.

Algorithm:

Calculation of MFB and MFB50 value:

1. Aquisition of in-cylinder pressure at any sampling position θ[i] (aquisition window ranges from -180° before TDC to 180° after TDC): P[0] P[1] ... P[N-1]
2. Calculation of cylinder volume V[i] at any sampling position θ[i], V[i] can be precalculated and stored in a calibration vector: V[0] V[1] ... V[N-1]
3. Calculation of heat release rate (QIST), (supposed delta θ is constant e.g. 0,5°), for each i = 1,2 ... N-2:

Equation 29

$$QV = K \cdot P[i] \cdot \frac{V[i+1] - V[i-1]}{(K-1) \cdot 2 \cdot \Delta\theta}$$

$$QP = V[i] \cdot \frac{P[i+1] - P[i-1]}{(K-1) \cdot 2 \cdot \Delta\theta}$$

$$QIST[i] = QV + QP$$

Note: The function calculates the heat release rate without multiplication with constant $\frac{1}{(K-1) \cdot 2 \cdot \Delta\theta}$ *in both QV and QP equations.*

4. Calculation of mass fraction burned MFB (integration of heat release rate), initial condition is MFB[1] = 0, then for each i = 2,3...N-2:

Equation 30

$$MFB[i] = MFB[i-1] + (QIST[i] + QIST[i-1]) \cdot \frac{\Delta\theta}{2}$$

Note: The function calculates the mass fraction burned without multiplication with constant $\frac{\Delta\theta}{2}$.

Also note that the function writes calculated MFB[1] at the P_MFB array position 0.

5. Find maximum *MFBmax* and minimum *MFBmin* value from MFB[i], i = 1,2...N-2
6. Calculate MFB50 value as:

Equation 31

$$MFB50 = \frac{MFBmax + MFBmin}{2}$$

Calculation of PMH index:

7. Starting with PMH_old = 0, calculate for each i = 1,2...N-1:

Equation 32

$$DVA = V[i] - V[i-1]$$

$$PMA = \frac{P[i] + P[i-1]}{2}$$

$$PMH = PMH\_old + PMA \cdot DVA$$

Note: There must be 8 bytes of readable memory behind input vector P_MFB and V.

Performance: See *Chapter 7: Performance* and *Table 46*.

Example 15. mfb50_pmh

```
#include "libdsp2.h"
float P[740];
float V[740];
float K = 1.4F;
unsigned int N = 714;
float PMH;
unsigned int MFB50_index;
float MFB50;

void main(void)
{
    /* from pressure vector P and volume vector V, calculate MFB50
    value, PMH index and mass fraction burned (MFB), MFB
    returned back in P vector, MFB1 at P array position 0 */
    MFB50 = mfb50_pmh(N, P, V, K, &PMH);
    /* search for MFB50 value in mass fraction burned (P) vector and
    return index at which the MFB50 value was found */
    MFB50_index = mfb50_index(N, P, MFB50);
}
```

6.16 mfb50_index - pressure sensing function 2

Function call:

```
unsigned int mfb50_index(unsigned int N, float *MFB, float MFB50);
```

Arguments:

Table 16. mfb50_index arguments

N	in	length of mass fraction burned vector MFB, N-2 must be multiple of 4, N must be greater than or equal to 10
MFB(1)	in	pointer to the input mass fraction burned vector of length N, layout is MFB(1) MFB(2) ...MFB(N-2) MFB(N-1) 0, use vector calculated by mfb50_pmh function
MFB50	in	MFB50 value, use value returned by mfb50_pmh function

1. The MFB vector must be double word aligned.

Description: Pressure sensing function 2. The function is used for in-cylinder pressure sensor management. This function should follow the mfb50_pmh function call. The function searches the mass fraction burned (MFB) vector and returns index at which the MFB50 value was found. The returned index is called MFB50% position index.

Algorithm:

1. For each i = 2,3 ... N-2, calculate absolute value of difference MFB[i] and MFB50:

Equation 33

$$diff[i] = |MFB[i] - MFB50|$$

2. Find index *i* at which the *diff*[*i*] has minimal value.

Note: There must be 8 bytes of readable memory behind input vector MFB.

Performance: See *Chapter 7: Performance* and *Table 47*.

Example 16. mfb50_index

```
#include "libdsp2.h"
float P[740];
float V[740];
float K = 1.4F;
unsigned int N = 714;
float PMH;
unsigned int MFB50_index;
float MFB50;

void main(void)
{
    /* from pressure vector P and volume vector V, calculate MFB50
    value, PMH index and mass fraction burned (MFB), MFB
    returned back in P vector, MFB1 at P array position 0 */
    MFB50 = mfb50_pmh(N, P, V, K, &PMH);
    /* search for MFB50 value in mass fraction burned (P) vector and
    return index at which the MFB50 value was found */
    MFB50_index = mfb50_index(N, P, MFB50);
}
```

6.17 conv3x3 - 2-D convolution with 3x3 kernel

Function call:

```
void conv3x3(unsigned char *a, unsigned char *b, signed char *c,
             unsigned short M, unsigned short N, unsigned short
shift);
```

Arguments:

Table 17. conv3x3 arguments

a(1)	in	input image of size MxN
b(1)	out	output image of size MxN
c(2)	in	3x3 convolution kernel (mask)
M	in	number of rows of a and b, must be >= 3
N	in	number of columns of a and b, must be multiple of 4, must be >= 8
shift	in	number of bits result is shifted down

1. a and b must be word aligned.
2. c must be half word aligned and there must be 1 byte of readable memory behind c.

Description: Computes the 2-D convolution on MxN input image using a 3x3 convolution kernel. The computed convolution value is shifted by shift bits to the right and then range limited to 0..255. The output image is of the same size as input image and has one pixel border that is not computed. The first and the last row of output image is not changed, the first column is zeroed and the last column will contain not meaningful results. The last column can be zeroed when ZERO_LASTCOL is defined to 1 for the penalty of one more instruction inside the inner loop. The ZERO_LASTCOL is a symbol defined in function source file.

Algorithm:

a - input matrix, c - 3x3 convolution kernel, b - output matrix

$$a = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0N-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1N-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2N-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ a_{(M-1)0} & a_{(M-1)1} & a_{(M-1)2} & \cdots & a_{M-1N-1} \end{bmatrix} \quad c = \begin{bmatrix} c_{00} & c_{01} & c_{02} \\ c_{10} & c_{11} & c_{12} \\ c_{20} & c_{21} & c_{22} \end{bmatrix} \quad b = \begin{bmatrix} b_{00} & b_{01} & b_{02} & \cdots & b_{0N-1} \\ b_{10} & b_{11} & b_{12} & \cdots & b_{1N-1} \\ b_{20} & b_{21} & b_{22} & \cdots & b_{2N-1} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ b_{(M-1)0} & b_{(M-1)1} & b_{(M-1)2} & \cdots & b_{M-1N-1} \end{bmatrix}$$

1. Compute 2-D convolution, i.e. for each i = 1,2 ... M-2, j = 1,2 ... N-2 compute b_{ij}:

Equation 34

$$b_{ij} = \sum_{k=0}^{2} \sum_{p=0}^{2} a_{(i+k-1)(j+p-1)} \cdot c_{(2-k)(2-p)}$$

2. Shift each computed bij by shift bits to the right: $b_{ij} = b_{ij} / 2^{shift}$
3. Range limit to 0..255:
   if (b_{ij} < 0) then b_{ij} = 0,
   if (b_{ij} > 255) then b_{ij} = 255

Note: The input/output images and convolution kernel must be properly aligned in memory as described in Table 17. And there must be 1 byte of readable memory behind the convolution kernel c.

Performance: See *Chapter 7: Performance* and *Table 48*.

Example 17. conv3x3

```
#include "libdsp2.h"

/* #pragma and ALIGN macros used to align to 4 bytes */
#ifdef __ghs__
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
unsigned char a[16*12];

/* #pragma and ALIGN macros used to align to 2 bytes */
#ifdef __ghs__
#pragma alignvar (2)
#endif
ALIGN_DCC(2)
signed char c[3*3] ALIGN_MWERKS(2) = {
    -1,   -2,   -1,
     0,    0,    0,
     1,    2,    1,
};

/* #pragma and ALIGN macros used to align to 4 bytes */
#ifdef __ghs__
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
unsigned char b[16*12] ALIGN_MWERKS(4);


unsigned short M, N, shift;

void main(void)
{
    /******** 2-D convolution with 3x3 kernel *********/
    M = 16;
    N = 12;
    shift = 4;
    conv3x3(a, b, c, M, N, shift);
}
```

6.18 sobel3x3 - sobel filter

Function call:

```
void sobel3x3(unsigned char *a, unsigned char *b, unsigned short M,
      unsigned short N);
```

Arguments:

Table 18. sobel3x3 arguments

a(1)	in	input image of size MxN
b(1)	out	output image of size MxN
M	in	number of rows of a and b, must be >= 3
N	in	number of columns of a and b, must be multiple of 4, must be >= 8

1. a and b must be word aligned.

Description: Computes the sobel filter on MxN input image. The function adds together absolute values of horizontal and vertical sobel filters and then limits output to 0..255. The output image is of the same size as input image and has one pixel border that is not computed. The first and the last row of output image is not changed, the first column is zeroed and the last column will contain not meaningful results. The last column can be zeroed when ZERO_LASTCOL is defined to 1 for the penalty of one more instruction inside the inner loop. The ZERO_LASTCOL is a symbol defined in function source file.

Algorithm:

1. Compute horizontal sobel filter, i.e. compute 2-D convolution with the following kernel:

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 0 \end{bmatrix}$$

2. Compute vertical sobel filter, i.e. compute 2-D convolution with the following kernel:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

3. Add absolute values of horizontal and vertical sobel filters
4. Range limit to 0..255

Note: The input/output images must be properly aligned in memory as described in Table 18.

Performance: See *Chapter 7: Performance* and *Table 49*.

Example 18. sobel3x3

```
#include "libdsp2.h"

/* #pragma and ALIGN macros used to align to 4 bytes */
#ifdef __ghs__
```

```
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
unsigned char a[16*12];

/* #pragma and ALIGN macros used to align to 4 bytes */
#ifdef __ghs__
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
unsigned char b[16*12] ALIGN_MWERKS(4);


unsigned short M, N, shift;

void main(void)
{
    /******** Sobel filter *********/
    M = 16;
    N = 12;
    sobel3x3(a, b, M, N);
}
```

6.19 sobel3x3_horizontal - horizontal sobel filter

Function call:

```
void sobel3x3_horizontal(unsigned char *a, unsigned char *b,
              unsigned short M, unsigned short N);
```

Arguments:

Table 19. sobel3x3_horizontal arguments

a(1)	in	input image of size MxN
b(1)	out	output image of size MxN
M	in	number of rows of a and b, must be >= 3
N	in	number of columns of a and b, must be multiple of 4, must be >= 8

1. a and b must be word aligned.

Description: Computes the horizontal sobel filter on MxN input image. The absolute value of horizontal sobel filter result is range limited to 0..255. The output image is of the same size as input image and has one pixel border that is not computed. The first and the last row of output image is not changed, the first column is zeroed and the last column will contain not meaningful results. The last column can be zeroed when ZERO_LASTCOL is defined to 1 for the penalty of one more instruction inside the inner loop. The ZERO_LASTCOL is a symbol defined in function source file.

Algorithm:

1. Compute horizontal sobel filter, i.e. compute 2-D convolution with the following kernel:

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 0 \end{bmatrix}$$

2. Get absolute value of result
3. Range limit to 0..255

Note: The input/output images must be properly aligned in memory as described in Table 19.

Performance: See *Chapter 7: Performance* and *Table 50*.

Example 19. sobel3x3_horizontal

```
#include "libdsp2.h"

/* #pragma and ALIGN macros used to align to 4 bytes */
#ifdef __ghs__
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
unsigned char a[16*12];

/* #pragma and ALIGN macros used to align to 4 bytes */
#ifdef __ghs__
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
unsigned char b[16*12] ALIGN_MWERKS(4);


unsigned short M, N, shift;

void main(void)
{
    /******** Horizontal sobel filter *********/
    M = 16;
    N = 12;
    sobel3x3_horizontal(a, b, M, N);
}
```

6.20 sobel3x3_vertical - vertical sobel filter

Function call:

```
void sobel3x3_vertical(unsigned char *a, unsigned char *b,
              unsigned short M, unsigned short N);
```

Arguments:

Table 20. sobel3x3_vertical arguments

a(1)	in	input image of size MxN
b(1)	out	output image of size MxN
M	in	number of rows of a and b, must be >= 3
N	in	number of columns of a and b, must be multiple of 4, must be >= 8

1. a and b must be word aligned.

Description: Computes the vertical sobel filter on MxN input image. The absolute value of vertical sobel filter result is range limited to 0..255. The output image is of the same size as input image and has one pixel border that is not computed. The first and the last row of output image is not changed, the first column is zeroed and the last column will contain not meaningful results. The last column can be zeroed when ZERO_LASTCOL is defined to 1 for the penalty of one more instruction inside inner loop. The ZERO_LASTCOL is a symbol defined in function source file.

Algorithm:

1. Compute vertical sobel filter, i.e. compute 2-D convolution with the following kernel:

$$G_X = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

2. 2. Get absolute value of result
3. 3. Range limit to 0..255

Note: The input/output images must be properly aligned in memory as described in Table 20.

Performance: See *Chapter 7: Performance* and *Table 51*.

Example 20. sobel3x3_vertical

```
#include "libdsp2.h"

/* #pragma and ALIGN macros used to align to 4 bytes */
#ifdef __ghs__
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
unsigned char a[16*12];

/* #pragma and ALIGN macros used to align to 4 bytes */
#ifdef __ghs__
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
unsigned char b[16*12] ALIGN_MWERKS(4);
```

```
unsigned short M, N, shift;

void main(void)
{
    /******** Vertical sobel filter *********/
    M = 16;
    N = 12;
    sobel3x3_vertical(a, b, M, N);
}
```

6.21 corr_frac16 - frac16 correlation function

Function call:

```
void corr_frac16(unsigned short N, unsigned short M, short *x,
        short *y, short *r);
```

Arguments:

Table 21. corr_frac16 arguments(1)

N	in	length of input vectors x and y (must be multiple of 4)
M	in	length of output vector r (must be multiple of 4, M <= N)
x	in	first input vector of length N
y	in	second input vector of length N
r	out	correlation of vectors x and y

1. The x,y,r data are in fractional 16-bit format in range -1 to 1. The arrays x,r must be word aligned, the array y must be half-word aligned (implicit alignment of short data).

Description: Computes the correlation of the arrays x and y. The result is written to the array r. All data types are signed 16-bit fractional in range -1 to $1-2^{-15}$. There is used the arithmetic with saturation so the output will never overflow.

Algorithm:

Correlation:

Equation 35

$$out(j) = x(j)y(0) + x(j+1)y(1) + ... + x(N-1)y(N-j-1)$$

for each j = 0, 1, ..., M-1

Note: The x and y can be the same array, in that case the auto-correlation is computed.

Performance: See *Chapter 7: Performance* and *Table 52*.

Example 21. corr_frac16

```
#include "libdsp2.h"

/* #pragma and ALIGN macros used to align to 4 bytes */
#ifdef __ghs__
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
short x[8] ALIGN_MWERKS(4) = {6554,10393,269,12670,2496,6233,9945,-
303};
short y[8]                 = {8192,-6113,3558,4066,-4590,10723,-
3612,6019};

/* #pragma and ALIGN macros used to align to 4 bytes */
#ifdef __ghs__
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
short r [8] ALIGN_MWERKS(4);

unsigned short M, N;

void main(void)
{
    /* compute correlation of x and y */
    N = 16;
    M = 16;
    corr_frac16(N, M, x, y, r);
}
```

6.22 fir_float - float FIR filter

Function call:

```
void fir_float (unsigned short N, unsigned short ntaps, float *x,
       float *y, float *h);
```

Arguments:

Table 22. fir_float arguments arguments(1)

N	in	length of output array y (must be multiple of 2)
ntaps	in	number of filter coefficients (must be >= 4)
x	in	array of input samples of length N+ntaps-1, x(-ntaps+1) x(-ntaps+2) ... x(0) x(1) ... x(N-1)
y	out	array of output samples of length N, y(0) y(1) ... y(N-1)
h	in	array of filter coefficients stored in reversed order, h(ntaps-1) ... h(1) h(0)

1. The arrays x, y, h must be double-word aligned. There must be 8 bytes of readable memory behind the arrays h and x.

Description: Computes the real FIR filter on float data. The input samples are stored in the array x, output samples are written to the array y.

Algorithm:

Equation 36 $$y(n) = \sum_{k=0}^{ntaps-1} h(k) \cdot x(n-k)$$ for each n from 0 to N-1

Note: The filter coefficients must be stored in reversed order.

Performance: See *Chapter 7: Performance* and *Table 53*.

Example 22. fir_float

```
#include "libdsp2.h"

/* #pragma and ALIGN macros used to align to 8 bytes */
#ifdef __ghs__
#pragma alignvar (8)
#endif
ALIGN_DCC(8)
float x[288/*+2*/] ALIGN_MWERKS(8); /* +2 to ensure there are 8
bytes of readable memory behind x */

#ifdef __ghs__
#pragma alignvar (8)
#endif
ALIGN_DCC(8)
float y [256] ALIGN_MWERKS(8);
#ifdef __ghs__
#pragma alignvar (8)
#endif
ALIGN_DCC(8)
float hr[33/*+2*/] ALIGN_MWERKS(8) = { /* +2 to ensure there are 8
bytes of readable memory behind hr */
  -0.0015288448F,-0.0019041377F,-0.0025120102F,-0.0031517229F,-
0.0033815748F,-
0.0025689987F,0.0000000000F,0.0049703708F,0.0127558533F,0.023399631
1F,0.0364901802F,0.0511553726F,0.0661431077F,0.0799794722F,0.091179
2726F,0.0984722196F,0.1010036179F,0.0984722196F,0.0911792726F,0.079
9794722F,0.0661431077F,0.0511553726F,0.0364901802F,0.0233996311F,0.
0127558533F,0.0049703708F,0.0000000000F,-0.0025689987F,-
0.0033815748F,-0.0031517229F,-0.0025120102F,-0.0019041377F,-
0.0015288448F,
};

unsigned short N, ntaps;

void main(void)
{
    /* filter the samples in x with FIR filter of 32nd order, */
```

```
    /* write result into y, hr are filter coefficients stored */
    /* in reversed order */
    N = 256;
    ntaps = 33;
    fir_float(N, ntaps, x, y, hr);
}
```

6.23 fir_frac16 - frac16 FIR filter

Function call:

```
void fir_frac16(unsigned short N, unsigned short ntaps, short *x,
      short *y, short *h);
```

Arguments:

Table 23. fir_frac16 arguments(1)

N	in	length of output array y (must be multiple of 2)
ntaps	in	number of filter coefficients (must be >= 4)
x	in	array of input samples of length N+ntaps-1, x(-ntaps+1) x(-ntaps+2) ... x(0) x(1) ... x(N-1)
y	out	array of output samples of length N, y(0) y(1) ... y(N-1)
h	in	array of filter coefficients stored in reversed order, h(ntaps-1) ... h(1) h(0)

1. The x,y,h data are in fractional 16-bit format in range -1 to 1. The arrays x, y must be word aligned, the array h must be half-word aligned (implicit alignment of short data). There must be 2 bytes of readable memory behind the array h and 4 bytes of readable memory behind the array x.

Description: Computes the real FIR filter on frac16 data. The input samples are stored in the array x, output samples are written to the array y. There is used arithmetic with saturation.

Algorithm:

Equation 37

$$y(n) = \sum_{k=0}^{ntaps-1} h(k) \cdot x(n-k) \quad \text{for each n from 0 to N-1}$$

Note: The coefficients stored in the array h must be scaled to prevent a saturation, i.e. the sum of the absolute values of all coefficients must be lower than 2^{15}:

Equation 38

$$\sum_{k=0}^{ntaps-1} |h(k)| < 2^{15}$$

The filter coefficients must be stored in reversed order.

Performance: See *Chapter 7: Performance* and *Table 54*.

Example 23. fir_frac16

```
#include "libdsp2.h"

/* #pragma and ALIGN macros used to align to 4 bytes */
#ifdef __ghs__
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
short x[288/*+2*/] ALIGN_MWERKS(4); /* +2 to ensure there are 4
bytes of readable memory behind x */
#ifdef __ghs__
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
short y[256] ALIGN_MWERKS(4);
short hr[33/*+1*/] = { /* +1 to ensure there are 2 bytes of readable
memory behind hr */
  -50,-62,-82,-103,-111,-
84,0,163,418,767,1196,1676,2167,2621,2988,3227,3310,3227,2988,2621,
2167,1676,1196,767,418,163,0,-84,-111,-103,-82,-62,-50,
};

unsigned short N, ntaps;

void main(void)
{
    /* filter the samples in x with FIR filter of 32nd order, */
    /* write result into y, hr are filter coefficients stored */
    /* in reversed order */
    N = 256;
    ntaps = 33;
    fir_frac16(N, ntaps, x, y, hr);
}
```

6.24 iir_float_1st - float first-order IIR filter

Function call:

```
void iir_float_1st (unsigned short N, float *x, float *y, float *c,
            float *s);
```

Arguments:

Table 24. iir_float_1st arguments(1)

N	in	number of input and output samples (N must be multiple of 4, must be >= 8)
x	in	array of input samples of length N, x(0) x(1) ... x(N-1)
y	out	array of output samples of length N, y(0) y(1) ... y(N-1)

Table 24. iir_float_1st arguments(1) (continued)

c	in	array of filter coefficients, c[3] = {b0, b1, a1}
s	in/out	array of previous input and output values (state vector), s[2] = {x(-1), y(-1)}

1. The arrays x,y,s must be double-word aligned, the array c must be word aligned (implicit alignment of float data). There must be 16 bytes of readable memory behind the array x. There are returned new values of x(-1), y(-1) in the array s to be ready for next function call.

Description: Computes the first-order IIR filter on float data. The input samples are stored in the array x, output samples are written to the array y.

Algorithm:

Equation 39

$$y(n) = b(0)x(n) + b(1)x(n-1) - a(1)y(n-1)$$

for each n from 0 to N-1

Performance: See *Chapter 7: Performance* and *Table 55*.

Example 24. iir_float_1st

```
#include "libdsp2.h"

/* #pragma and ALIGN macros used to align to 8 bytes */
#ifdef __ghs__
#pragma alignvar (8)
#endif
ALIGN_DCC(8)
float x[256/*+4*/] ALIGN_MWERKS(8); /* +4 to ensure there are 16
bytes of readable memory behind x */

#ifdef __ghs__
#pragma alignvar (8)
#endif
ALIGN_DCC(8)
float y [256] ALIGN_MWERKS(8);

#ifdef __ghs__
#pragma alignvar (8)
#endif
ALIGN_DCC(8)
float s [2] ALIGN_MWERKS(8) = {0,0};
float c [3]                 =
{0.5157131330F,0.5157131330F,0.0314262660F};

unsigned short N;
```

```
void main(void)
{
    /* filter the samples in x with first-order IIR filter, */
    /* write result into y */
    N = 256;
    iir_float_1st(N, x, y, c, s);
}
```

6.25 iir_float_2nd - float second-order IIR filter

Function call:

```
void iir_float_2nd (unsigned short N, float *x, float *y, float *c,
            float *s);
```

Arguments:

Table 25. iir_float_2nd arguments(1)

N	in	number of input and output samples (N must be multiple of 4, must be >= 8)
x	in	array of input samples of length N, x(0) x(1) ... x(N-1)
y	out	array of output samples of length N, y(0) y(1) ... y(N-1)
c	in	array of filter coefficients, c[5] = {b0, b1, b2, a1, a2}
s	in/out	array of previous input and output values (state vector), s[4] = {x(-2), x(-1), y(-2), y(-1)}

1. The arrays x,y,s must be double-word aligned, the array c must be word aligned (implicit alignment of float data). There must be 16 bytes of readable memory behind the array x. There are returned new values of x(-2), x(-1), y(-2), y(-1) in the array s to be ready for next function call.

Description: Computes the second-order IIR filter on float data. The input samples are stored in the array x, output samples are written to the array y.

Algorithm:

Equation 40

$$y(n) = b(0)x(n) + b(1)x(n-1) + b(2)x(n-2) - a(1)y(n-1) - a(2)y(n-2)$$

for each n from 0 to N-1

Performance: See *Chapter 7: Performance* and *Table 56*.

Example 25. iir_float_2nd

```
#include "libdsp2.h"

/* #pragma and ALIGN macros used to align to 8 bytes */
#ifdef __ghs__
#pragma alignvar (8)
#endif
```

```
ALIGN_DCC(8)
float x[256/*+4*/] ALIGN_MWERKS(8); /* +4 to ensure there are 16
bytes of readable memory behind x */

#ifdef __ghs__
#pragma alignvar (8)
#endif
ALIGN_DCC(8)
float y [256] ALIGN_MWERKS(8);

#ifdef __ghs__
#pragma alignvar (8)
#endif
ALIGN_DCC(8)
float s[4] ALIGN_MWERKS(8) = {0,0,0,0};
float c[5]                 =
{0.3115387742F,0.6230775485F,0.3115387742F,0.0736238464F,0.17253125
05F};

unsigned short N;

void main(void)
{
    /* filter the samples in x with second-order IIR filter, */
    /* write result into y */
    N = 256;
    iir_float_2nd(N, x, y, c, s);
}
```

6.26 iir_float_casc - cascade of float second-order IIR filters

Function call:

```
#define iir_float_casc    iir_float_casc_c
void iir_float_casc_c(unsigned short N, float *x, float *y, float
*c,
            float *s, unsigned short m);
```

Arguments:

Table 26. iir_float_casc arguments(1)

N	in	number of input and output samples (N must be multiple of 4, must be >= 8)
x	in	array of input samples of length N, x(0) x(1) ... x(N-1)
y	out	array of output samples of length N, y(0) y(1) ... y(N-1)
c	in	array of filter coefficients for each second-order IIR filter, c[m*5] = {b10, b11, b12, a11, a12, b20, b21, b22, a21, a22, ..., bm0, bm1, bm2, am1, am2}

Table 26. iir_float_casc arguments(1) (continued)

s	in/out	array of previous input and output values (state vector), s[m*4] = {x1(-2), x1(-1), y1(-2), y1(-1), x2(-2), x2(-1), y2(-2), y2(-1), ..., xm(-2), xm(-1), ym(-2), ym(-1)}
m	in	number of second-order IIR filters in the cascade

1. The arrays x,y,s must be double-word aligned, the array c must be word aligned (implicit alignment of float data). There must be 16 bytes of readable memory behind the array x and y. There are returned new values of x(-2), x(-1), y(-2), y(-1) in the array s to be ready for next function call.

Description: Computes the cascade of second-order IIR filters on float data. The input samples are stored in the array x, output samples are written to the array y. The function calls the iir_float_2nd function for each second-order IIR filter in the cascade.

Algorithm:

The filter equation for the i-order IIR filter (i = 2*m):

Equation 41

$$y(n) = b(0)x(n) + b(1)x(n-1) + \ldots + b(i)x(n-i) - a(1)y(n-1) - \ldots - a(i)y(n-i)$$

for each n = 0,1...N-1

The i-order filter is implemented using the cascade of m IIR filters of second-order:

Equation 42

$$y_1(n) = b(10)x_1(n) + b(11)x_1(n-1) + b(12)x_1(n-2) - a(11)y_1(n-1) - a(12)y_1(n-2)$$

Equation 43

$$y_2(n) = b(20)y_1(n) + b(21)y_1(n-1) + b(22)y_1(n-2) - a(21)y_2(n-1) - a(22)y_2(n-2)$$

…

Equation 44

$$y_m(n) = b(m0)y_{m-1}(n) + b(m1)y_{m-1}(n-1) + b(m2)y_{m-1}(n-2) - a(m1)y_m(n-1) - a(m2)y_m(n-2)$$

for each n = 0,1...N-1

Performance: See *Chapter 7: Performance* and *Table 57*.

Example 26. iir_float_casc

```
#include "libdsp2.h"

/* #pragma and ALIGN macros used to align to 8 bytes */
#ifdef __ghs__
#pragma alignvar (8)
#endif
ALIGN_DCC(8)
float x[256/*+4*/] ALIGN_MWERKS(8); /* +4 to ensure there are 16
bytes of readable memory behind x */

#ifdef __ghs__
#pragma alignvar (8)
#endif
ALIGN_DCC(8)
float y [256/*+4*/] ALIGN_MWERKS(8); /* +4 to ensure there are 16
bytes of readable memory behind y */

#ifdef __ghs__
#pragma alignvar (8)
#endif
ALIGN_DCC(8)
float s[3*4] ALIGN_MWERKS(8) = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
float c[3*5]                 = {0.3284466238F,   0.6596805377F,
0.3312418003F,   0.0639408215F,   0.0183196767F,
                                  0.2531886900F,   0.5063763109F,
0.2531936959F,   0.0736238464F,   0.1725312505F,
                                  0.4280607669F,   0.8524906977F,
0.4244401939F,   0.0998014847F,   0.5894355624F,};

unsigned short N;

void main(void)
{
    /* filter the samples in x with sixth-order IIR filter, */
    /* write result into y */
    N = 256;
    iir_float_casc(N, x, y, c, s, 3);
}
```

6.27 iir_frac16_1st - frac16 first-order IIR filter

Function call:

```
void iir_frac16_1st(unsigned short N, short *x, short *y, short *c,
            short *s);
```

Arguments:

Table 27. iir_frac16_1st arguments(1)

N	in	number of input and output samples (N must be multiple of 4)
x	in	array of input samples of length N, x(0) x(1) ... x(N-1)
y	out	array of output samples of length N, y(0) y(1) ... y(N-1)
c	in	array of filter coefficients, c[3] = {b0, b1, a1}
s	in/out	array of previous input and output values (state vector), s[2] = {x(-1), y(-1)}

1. The x,y,c,s data are in fractional 16-bit format in range -1 to 1. The arrays x,y,s must be word aligned, the array c must be half-word aligned (implicit alignment of short data). There must be 4 bytes of readable memory behind the array x. There are returned new values of x(-1), y(-1) in the array s to be ready for next function call.

Description: Computes the first-order IIR filter on frac16 data. The input samples are stored in the array x, output samples are written to the array y. There is used the arithmetic with saturation so the output will never overflow.

Algorithm:

Equation 45

$$y(n) = b(0)x(n) + b(1)x(n-1) - a(1)y(n-1)$$

for each n from 0 to N-1

Performance: See *Chapter 7: Performance* and *Table 58*.

Example 27. iir_frac16_1st

```
#include "libdsp2.h"

/* #pragma and ALIGN macros used to align to 4 bytes */
#ifdef __ghs__
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
short x[256/*+2*/] ALIGN_MWERKS(4); /* +2 to ensure there are 4
bytes of readable memory behind x */

#ifdef __ghs__
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
short y[256] ALIGN_MWERKS(4);

#ifdef __ghs__
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
short s[2] ALIGN_MWERKS(4) = {0,0};
```

```
short c[3]                  = {16899,16899,1030};

unsigned short N;

void main(void)
{
    /* filter the samples in x with first-order IIR filter, */
    /* write result into y */
    N = 256;
    iir_frac16_1st(N, x, y, c, s);
}
```

6.28 iir_frac16_2nd - frac16 second-order IIR filter

Function call:

```
void iir_frac16_2nd(unsigned short N, short *x, short *y, short
*c, short *s);
```

Arguments:

Table 28. iir_frac16_2nd arguments(1)

N	in	number of input and output samples (N must be multiple of 4)
x	in	array of input samples of length N, x(0) x(1) ... x(N-1)
y	out	array of output samples of length N, y(0) y(1) ... y(N-1)
c	in	array of filter coefficients, c[5] = {b0, b1, b2, a1, a2}
s	in/out	array of previous input and output values (state vector), s[4] = {x(-2), x(-1), y(-2), y(-1)}

1. The x,y,c,s data are in fractional 16-bit format in range -1 to 1. The arrays x,y,s must be word aligned, the array c must be half-word aligned (implicit alignment of short data). There must be 4 bytes of readable memory behind the array x. There are returned new values of x(-2), x(-1), y(-2), y(-1) in the array s to be ready for next function call.

Description: Computes the second-order IIR filter on frac16 data. The input samples are stored in the array x, output samples are written to the array y. There is used the arithmetic with saturation so the output will never overflow.

Algorithm:

Equation 46

$$y(n) = b(0)x(n) + b(1)x(n-1) + b(2)x(n-2) - a(1)y(n-1) - a(2)y(n-2)$$

for each n from 0 to N-1

Performance: See *Chapter 7: Performance* and *Table 59*.

Example 28. iir_frac16_2nd

```
#include "libdsp2.h"
```

```
/* #pragma and ALIGN macros used to align to 4 bytes */
#ifdef __ghs__
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
short x[256/*+2*/] ALIGN_MWERKS(4); /* +2 to ensure there are 4
bytes of readable memory behind x */

#ifdef __ghs__
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
short y[256] ALIGN_MWERKS(4);

#ifdef __ghs__
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
short s[4] ALIGN_MWERKS(4) = {0,0,0,0};
short c[5]                 = {10209,20417,10209,2413,5654};

unsigned short N;

void main(void)
{
    /* filter the samples in x with second-order IIR filter, */
    /* write result into y */
    N = 256;
    iir_frac16_2nd(N, x, y, c, s);
}
```

6.29 iir_frac16_casc - cascade of frac16 second-order IIR filters

Function call:

```
#define iir_frac16_casc   iir_frac16_casc_c
void iir_frac16_casc_c(unsigned short N, short *x, short *y, short
*c,
              short *s, unsigned short m);
```

Arguments:

Table 29. iir_frac16_casc arguments(1)

N	in	number of input and output samples (N must be multiple of 4)
x	in	array of input samples of length N, x(0) x(1) ... x(N-1)
y	out	array of output samples of length N, y(0) y(1) ... y(N-1)

Table 29. iir_frac16_casc arguments(1) (continued)

c	in	array of filter coefficients for each second-order IIR filter, c[m*5] = {b10, b11, b12, a11, a12, b20, b21, b22, a21, a22, ..., bm0, bm1, bm2, am1, am2}
s	in/out	array of previous input and output values (state vector), s[m*4] = {x1(-2), x1(-1), y1(-2), y1(-1), x2(-2), x2(-1), y2(-2), y2(-1), ..., xm(-2), xm(-1), ym(-2), ym(-1)}
m	in	number of second-order IIR filters in the cascade

1. The x,y,c,s data are in fractional 16-bit format in range -1 to 1. The arrays x,y,s must be word aligned, the array c must be half-word aligned (implicit alignment of short data). There must be 4 bytes of readable memory behind the array x and y. There are returned new values of x(-2), x(-1), y(-2), y(-1) in the array s to be ready for next function call.

Description: Computes the cascade of second-order IIR filters on frac16 data. The input samples are stored in the array x, output samples are written to the array y. The function calls the iir_float_2nd function for each second-order IIR filter in the cascade. There is used the arithmetic with saturation so the output will never overflow.

Algorithm:

The filter equation for the i-order IIR filter (i = 2*m):

Equation 47

$$y(n) = b(0)x(n) + b(1)x(n-1) + \ldots + b(i)x(n-i) - a(1)y(n-1) - \ldots - a(i)y(n-i)$$

for each n = 0,1...N-1

The i-order filter is implemented using the cascade of m IIR filters of second-order:

Equation 48

$$y_1(n) = b(10)x_1(n) + b(11)x_1(n-1) + b(12)x_1(n-2) - a(11)y_1(n-1) - a(12)y_1(n-2)$$

Equation 49

$$y_2(n) = b(20)y_1(n) + b(21)y_1(n-1) + b(22)y_1(n-2) - a(21)y_2(n-1) - a(22)y_2(n-2)$$

…

Equation 50

$$y_m(n) = b(m0)y_{m-1}(n) + b(m1)y_{m-1}(n-1) + b(m2)y_{m-1}(n-2) - a(m1)y_m(n-1) - a(m2)y_m(n-2)$$

for each n = 0,1...N-1

Performance: See *Chapter 7: Performance* and *Table 60*.

Example 29. iir_frac16_casc

```
#include "libdsp2.h"

/* #pragma and ALIGN macros used to align to 4 bytes */
#ifdef __ghs__
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
short x[256/*+2*/] ALIGN_MWERKS(4); /* +2 to ensure there are 4
bytes of readable memory behind x */

#ifdef __ghs__
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
short y[256/*+2*/] ALIGN_MWERKS(4); /* +2 to ensure there are 4
bytes of readable memory behind y */

#ifdef __ghs__
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
short s[3*4] ALIGN_MWERKS(4) = {0,0,0,0,0,0,0,0,0,0,0,0};
short c[3*5]                 = {10763,21616,10854,2095,600,
                                8296,16593,8297,2413,5654,
                                14027,27934,13908,3270,19315,};

unsigned short N;

void main(void)
{
    /* filter the samples in x with sixth-order IIR filter, */
    /* write result into y */
    N = 256;
    iir_frac16_casc(N, x, y, c, s, 3);
}
```

6.30 iir_frac16_2nd_hc - frac16 second-order IIR filter with half coefficients

Function call:

```
void iir_frac16_2nd_hc(unsigned short N, short *x, short *y, short
*c,
            short *s);
```

Arguments: dkkdkdkd

Returns:

Table 30. iir_frac16_2nd_hc arguments(1)

N	in	number of input and output samples (N must be multiple of 4, N must be >= 8)
x	in	array of input samples of length N, x(0) x(1) ... x(N-1)
y	out	array of output samples of length N, y(0) y(1) ... y(N-1)
c	in	array of filter coefficients divided by 2, c[5] = {b(0)/2, b(1)/2, b(2)/2, a(1)/2, a(2)/2}
s	in/out	array of previous input and output values (state vector), s[4] = {x(-2), x(-1), y(-2), y(-1)}

1. The x,y,c,s data are in fractional 16-bit format in range -1 to 1. The arrays x,y,s must be word aligned, the array c must be half-word aligned (implicit alignment of short data). There must be 4 bytes of readable memory behind the array x. There are returned new values of x(-2), x(-1), y(-2), y(-1) in the array s to be ready for next function call.

Description: Computes the second-order IIR filter on frac16 data. The input samples are stored in the array x, output samples are written to the array y. The filter coefficients are stored in the array c divided by 2. There is used the arithmetic with saturation so the output will never overflow.

Algorithm:

Equation 51

$$y(n) = 2\left(\frac{b(0)}{2}x(n) + \frac{b(1)}{2}x(n-1) + \frac{b(2)}{2}x(n-2) - \frac{a(1)}{2}y(n-1) - \frac{a(2)}{2}y(n-2)\right)$$

for each n from 0 to N-1

Note: The routine requires these conditions are met:
*-1 =< b1/2-a1*b0/2 < 1*
*-1 =< b2/2-a1*b1/2 < 1*
*-1 =< -a1*b2/2 < 1*
*-1 =< a2/2-a1*a1/2 < 1*
*-1 =< -a1*a2/2 < 1*

Performance: See *Chapter 7: Performance* and *Table 61*.

Example 30. iir_frac16_2nd_hc

```
#include "libdsp2.h"

/* #pragma and ALIGN macros used to align to 4 bytes */
#ifdef __ghs__
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
short x[256] ALIGN_MWERKS(4);

#ifdef __ghs__
#pragma alignvar (4)
```

```
#endif
ALIGN_DCC(4)
short y[256] ALIGN_MWERKS(4);

#ifdef __ghs__
#pragma alignvar (4)
#endif
ALIGN_DCC(4)
short s[4] ALIGN_MWERKS(4) = {9830,13107,-3932,11665};
short c[5]                 = {10209,20417,10209,2413,5654};

unsigned short N;

void main(void)
{
    /* filter the samples in x with second-order IIR filter, */
    /* write result into y, coefficients in c are stored */
    /* divided by 2 */
    N = 256;
    iir_frac16_2nd_hc(N, x, y, c, s);
}
```

7 Performance

This section shows the code size and clock cycles for each function. The clock cycles were measured for the following conditions:

- Code Memory (Internal Flash) - cache on
- Data Memory (Internal RAM) - cache on
- Stack Memory - locked in cache
- Data acquired on the MPC5554 with SIU_MIDR = 0x55540011
- Branch Target Buffer (BTB) enabled
- BIUCR = 0x00094BFD

The “Improvement to C function” column shows performance increase comparing the assembly code to a respective C function, the value is calculated as a ratio of the number_of_clock_cycles_of_C_ function/number_of_clock_cycles_of_optimized_ library_ function. The number of clock cycles are taken for the third function call. The C-code was compiled in CodeWarrior for PowerPC V1.5 beta2, Global Optimizations set to level 4, Optimize for Faster Execution Speed, Instruction Scheduling and Peephole Optimization turned on.

The IPC column shows instructions per cycle.

Table 31. Code size

Function	Code size [bytes]
bitrev_table_16bit	168
bitrev_table_32bit	168
fft_radix4_float	1216
fft_radix4_frac32 - scaling off	1224
fft_radix4_frac32 - scaling on	1344
fft_quad_float	1268
fft_quad_frac32 - scaling off	1276
fft_quad_frac32 - scaling on	1388
fft_radix2_last_stage_float	324
fft_radix2_last_stage_frac32 - scaling off	328
fft_radix2_last_stage_frac32 - scaling on	368
fft_real_split_float	452(1)
fft_real_split_frac32 - scaling off	452(1)
fft_real_split_frac32 - scaling on	460(2)
window_apply_complex_float	260
window_apply_complex_frac16	244(3)
window_apply_real_float	172
window_apply_real_frac16	172
mfb50_pmh	1100

Table 31. Code size (continued)

Function	Code size [bytes]
mfb50_index	180
conv3x3	672[4]
sobel3x3	492[4]
sobel3x3_horizontal	348[4]
sobel3x3_vertical	352[4]
corr_frac16	440
fir_float	776
fir_frac16	568
iir_float_1st	324
iir_float_2nd	416
iir_float_casc (iir_float_casc_c)	124[5]
iir_frac16_1st	164
iir_frac16_2nd	208
iir_frac16_casc (iir_frac16_casc_c)	124[5]
iir_frac16_2nd_hc	284

1. N2_REALPART configured to 0, for N2_REALPART configured to 1 code size is 460 bytes.
2. The code size is the same for N2_REALPART configured to 0 or 1.
3. Valid for both scaling on and off.
4. ZERO_LASTCOL configured to 0, if configured to 1 the size is 4 bytes greater for conv3x3 function and 8 bytes greater for the sobel functions.
5. The iir_float_casc (iir_frac16_casc) function calls function iir_float_2nd (iir_frac16_2nd). The code size in the table is valid for the CodeWarrior compiler with optimizations set to level 4, Instruction Scheduling and Peephole Optimization turned on.

Table 32. Radix-4 complex to complex float in-place FFT[1]

N[2]	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
64	2546	2274	2259	2.57	0.97
256	12362	11653	11637	2.4	0.98
1024	59917	57374	56911	2.3	0.99
4096	307499	305244	304521	2.1	0.89

1. The clock cycles include bit reversing by bitrev_table_32bit and FFT computation by fft_radix4_float function. w_table in internal flash, seed_table in internal flash.
2. N is FFT length.

Table 33. Quad radix-2 complex to complex float in-place FFT(1)

N(2)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
64	2683	2421	2415	2.2	0.96
256	12843	12196	12173	2.1	0.98
1024	62713	60186	60075	2.0	0.99
4096	320423	316501	316011	1.9	0.90

1. The clock cycles include bit reversing by bitrev_table_32bit and FFT computation by fft_quad_float function. w_table in internal flash, seed_table in internal flash.
2. N is FFT length.

Table 34. Radix-2 complex to complex float in-place FFT(1)

N(2)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
128	6197	5749	5739	2.1	0.98
512	29801	28519	28473	2.1	0.99
2048	144575	138638	137365	2.0	0.99

1. The clock cycles include bit reversing by bitrev_table_32bit and FFT computation by fft_quad_float function and fft_radix2_last_stage_float function. w_table in internal flash, seed_table in internal flash.
2. N is FFT length.

Table 35. Radix-4 complex to complex frac16/frac32 in-place FFT, scaling on(1)

N(2)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
64	2746	2480	2474	-	0.97
256	13411	12773	12740	-	0.98
1024	65051	62715	62334	-	0.99
4096	326464	324470	324053	-	0.91

1. The clock cycles include bit reversing by bitrev_table_16bit and FFT computation by fft_radix4_frac32 function. w_table in internal flash, seed_table in internal flash.
2. N is FFT length.

Table 36. Quad radix-2 complex to complex frac16/frac32 in-place FFT, scaling on(1)

N(2)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
64	2901	2627	2620	-	0.97
256	13913	13234	13234	-	0.98

Table 36. Quad radix-2 complex to complex frac16/frac32 in-place FFT, scaling on[1] (continued)

N[2]	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
1024	67669	65418	65328	-	0.99
4096	340105	337619	337101	-	0.92

1. The clock cycles include bit reversing by bitrev_table_16bit and FFT computation by fft_quad_frac32 function. w_table in internal flash, seed_table in internal flash.
2. N is FFT length.

Table 37. Radix-2 complex to complex frac32 in-place FFT, scaling on[1]

N[2]	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
128	6763	6270	6261	-	0.98
512	32312	31060	31043	-	0.99
2048	156208	151037	149960	-	0.99

1. The clock cycles include bit reversing by bitrev_table_16bit and FFT computation by fft_quad_frac32 function and fft_radix2_last_stage_frac32 function. w_table in internal flash, seed_table in internal flash.
2. N is FFT length.

The number of clock cycles for third function call of the frac16/frac32 FFTs without scaling is the same as for respective float FFT plus N/8.

Table 38. Real to complex float in-place FFT (N odd power of two)[1]

N[2]	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
128	3659	3258	3233	2.3	0.97
512	16522	15384	15251	2.3	0.98
2048	76073	72880	71331	2.2	0.99

1. The clock cycles include bit reversing by bitrev_table_32bit and FFT computation by fft_radix4_float function and fft_real_split_float function. w_table in internal flash, seed_table in internal flash, wa_table and wb_table in internal flash. N2_REALPART set to 0.
2. N is FFT length.

Table 39. Real to complex float in-place FFT (N even power of two)(1)

N(2)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
256	8288	7615	7593	2.1	0.98
1024	37929	35826	35607	2.0	0.99
4096	176668	173279	172939	1.9	0.95

1. The clock cycles include bit reversing by bitrev_table_32bit and FFT computation by fft_quad_float function, fft_radix2_last_stage_float function and fft_real_split_float function. w_table in internal flash, seed_table in internal flash, wa_table and wb_table in internal flash. N2_REALPART set to 0.
2. N is FFT length.

Table 40. Real to complex frac16/frac32 in-place FFT, scaling on (N odd power of two)(1)

N(2)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
128	3881	3477	3450	-	0.97
512	17495	16492	16356	-	0.99
2048	81210	78233	76899	-	0.99

1. The clock cycles include bit reversing by bitrev_table_16bit and FFT computation by fft_radix4_frac32 function and fft_real_split_frac32 function. w_table in internal flash, seed_table in internal flash, wa_table and wb_table in internal flash. N2_REALPART set to 0.
2. N is FFT length.

Table 41. Real to complex frac16/frac32 in-place FFT, scaling on (N even power of two)(1)

N(2)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
256	8887	8172	8117	-	0.98
1024	40376	38527	38179	-	0.99
4096	188384	185207	184761	-	0.96

1. The clock cycles include bit reversing by bitrev_table_16bit and FFT computation by fft_quad_frac32 function, fft_radix2_last_stage_frac32 function and fft_real_split_frac32 function. w_table in internal flash, seed_table in internal flash, wa_table and wb_table in internal flash. N2_REALPART set to 0.
2. N is FFT length.

The number of clock cycles for third function call of the frac16/frac32 real to complex in-place FFTs without scaling is the same as for respective float FFT plus N/16.

Table 42. Complex float windowing function(1)

N(2)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
64	815	666	666	2.0	0.99
128	1545	1295	1290	2.0	0.99
256	2991	2541	2538	2.0	0.99
512	5882	5078	5034	2.0	0.99
1024	11652	10192	10026	2.0	0.99
2048	23189	21928	21427	1.9	0.93

1. w vector in internal flash.
2. N is window length.

Table 43. Complex frac16 windowing function(1)

N(2)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
64	691	608	602	-	0.99
128	1291	1162	1162	-	0.99
256	2502	2282	2282	-	0.99
512	4910	4522	4522	-	0.99
1024	9724	9022	9002	-	0.99
2048	19329	18061	17962	-	0.99

1. w vector in internal flash. The table is valid for both scaling on and off.
2. N is window length.

Table 44. Real float windowing function(1)

N(2)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
64	298	188	188	3.1	0.86
128	524	360	360	3.1	0.85
256	965	704	704	3.1	0.84
512	1843	1398	1392	3.1	0.84
1024	3609	2801	2768	3.1	0.84
2048	7159	5658	5520	3.1	0.84
4096	14289	13145	12564	2.8	0.74

1. w vector in internal flash.
2. N is window length.

Table 45. Real frac16 windowing function(1)

N(2)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
64	264	188	188	-	0.86
128	469	360	360	-	0.85
256	862	704	704	-	0.84
512	1644	1392	1392	-	0.84
1024	3207	2777	2768	-	0.84
2048	6365	5540	5520	-	0.84
4096	12644	11143	11024	-	0.84

1. w vector in internal flash.
2. N is window length.

Table 46. mfb50_pmh - pressure sensing function 1

N(1)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
18	390	308	301	2.9	0.96
26	488	407	407	3.1	0.96
34	598	507	507	3.3	0.97
42	693	609	609	3.3	0.97
362	4999	4702	4689	3.7	1.00
722	9812	9301	9279	3.7	1.00

1. N is length of P_MFB and V vectors.

*Note: The number of clock cycles for third function call for N greater than or equal 42 is num_of_cycles = 609+((N-2)/8-5)*102 (609 is number of clock cycles for N = 42 and 102 is number of instructions in the loop).*

Table 47. mfb50_index - pressure sensing function 2

N(1)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
18	128	91	91	2.5	0.95
26	157	123	123	2.7	0.96
34	191	155	155	2.9	0.97
42	225	187	187	3.0	0.97
362	1585	1467	1467	3.4	1.00
722	3116	2907	2907	3.5	1.00

1. N is length of MFB vector.

*Note: The number of clock cycles for third function call for N greater than or equal 18 is num_of_cycles = 91+((N-2)/4-4)*16 (91 is number of clock cycles for N = 18 and 16 is number of instructions in the loop).*

Table 48. conv3x3 - 2-D convolution with 3x3 kernel(1)

MxN(2)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
16x8	1453	1401	1399	3.0	0.97
8x16	1367	1320	1318	3.0	0.96
10x16	1767	1719	1717	3.1	0.97
32x32	12584	12484	12477	3.3	0.99

1. All measurements executed with convolution kernel c placed in internal RAM.
2. MxN is size of input and output matrix (M is number of rows).

Note: The number of instructions executed is 76+(M-2)(39+53*(N-4)/4).*

Table 49. sobel3x3 - sobel filter(1)

MxN(2)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
16x8	1304	1256	1248	3.0	0.98
8x16	1209	1162	1162	3.1	0.97
10x16	1569	1528	1528	3.1	0.97
32x32	11419	11318	11314	3.3	0.99

1. The number of instructions executed is 38+(M-2)*(37+48*(N-4)/4).
2. MxN is size of input and output matrix (M is number of rows).

Table 50. sobel3x3_horizontal - horizontal sobel filter(1)

MxN(2)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
16x8	904	871	871	2.7	0.98
8x16	852	812	812	2.8	0.96
10x16	1100	1066	1066	2.8	0.96
32x32	7910	7820	7820	2.9	0.99

1. The number of instructions executed is 28+(M-2)*(26+33*(N-4)/4).
2. MxN is size of input and output matrix (M is number of rows).

Table 51. sobel3x3_vertical - vertical sobel filter(1)

MxN(2)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
16x8	935	907	907	2.9	0.96
8x16	841	815	815	3.1	0.96
10x16	1099	1071	1071	3.1	0.97
32x32	7932	7847	7847	3.3	0.99

1. The number of instructions executed is 28+(M-2)*(27+33*(N-4)/4).
2. MxN is size of input and output matrix (M is number of rows).

Table 52. corr_frac16 - correlation function

N(1)	M(1)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
4	4	97	57	57	4.0	0.86
16	16	400	368	355	4.6	0.92
32	32	1168	1115	1115	4.8	0.96
128	128	15161	15083	15083	4.7	0.99
256	256	58938	58808	58795	4.6	1.00

1. N - length of input vectors, M - length of output vector.

Table 53. fir_float - FIR filter for float data

N(1)	ntaps(1)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
256	20	12974	12856	12862	3.0	0.98
256	22	13762	13633	13633	3.2	0.98
256	32	19119	19008	19006	3.1	0.99
256	64	35516	35392	35390	2.9	0.99

1. N - number of output samples, ntaps - number of filter coefficients.

Table 54. fir_frac16 - FIR filter for frac16 data

N(1)	ntaps(1)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
256	20	10631	10544	10550	3.8	0.97
256	22	11294	11196	11193	4.0	0.97
256	32	16018	15926	15926	3.8	0.98
256	64	30358	30262	30262	3.5	0.99

1. N - number of output samples, ntaps - number of filter coefficients.

Table 55. iir_float_1st - first-order IIR filter for float data

N(1)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
128	1013	935	935	2.2	0.97
256	1946	1831	1831	2.2	0.97

1. N - number of input/output samples.

Table 56. iir_float_2nd - second-order IIR filter for float data

N(1)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
128	1265	1171	1171	2.5	1.00
256	2403	2291	2291	2.5	1.00

1. N - number of input/output samples.

Table 57. iir_float_casc - cascade of second-order IIR filters for float data

N(1)	m(1)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
256	1	2461	2339	2338	-	0.99
256	2	4799	4649	4640	-	0.99
256	3	7084	6936	6936	-	0.99

1. N - number of input/output samples, m - number of second-order filters.

Table 58. iir_frac16_1st - first-order IIR filter for frac16 data

N(1)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
128	758	703	703	2.5	1.00
256	1451	1375	1375	2.5	1.00

1. N - number of input/output samples.

Table 59. iir_frac16_2nd - second-order IIR filter for frac16 data

N(1)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
128	890	838	838	3.3	1.00
256	1719	1638	1638	3.3	1.00

1. N - number of input/output samples.

Table 60. iir_frac16_casc - cascade of second-order IIR filters for frac16 data

N(1)	m(1)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
256	1	1777	1688	1685	-	0.99
256	2	3454	3340	3334	-	0.99
256	3	5099	4977	4977	-	0.99

1. N - number of input/output samples, m - number of second-order filters.

Table 61. iir_frac16_2nd_hc - second-order IIR filter for frac16 data with half coefficients

N(1)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
128	970	919	919	3.1	1.00
256	1855	1783	1783	3.2	1.00

1. N - number of input/output samples.

8 Revision history

Table 62. Document revision history

Date	Revision	Changes
30-Jun-2008	1	Initial release.
18-Sep-2013	2	Updated Disclaimer.
22-Apr-2015	3	Editorial and formatting changes throughout reference manual. Added the RPNs RPC56xx for defence application in cover page.