RM0019
Reference manual

SPC56xx DSP function library 1

Introduction

The SPC56xx DSP function library 1 contains optimized functions for SPC56xx family of processors with Signal Processing Engine (SPE APU).

Contents

List of tables

1 Library functions

The library contains the following functions:

- **conv_enc_mux** - rate 1/2 constraint length 9 convolutional encoder
- **corr_mn** - cross-correlation
- **crc_calc** - 16-bit cyclic redundancy code
- **dct** - 2-D 8x8 DCT
- **fft256** - 256-point complex FFT, single precision floating point
- **fir20** - real 20TAP FIR filter
- **fir22sym** - real 22TAP FIR filter with symmetrical impulse response
- **fir32sym** - real 32TAP FIR filter with symmetrical impulse response
- **fir64sym** - real 64TAP FIR filter with symmetrical impulse response
- **iir_2nd_o** - direct form 2nd order IIR filter
- **lu_decomp** - LU decomposition

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
- **include** - contains library header file libdsp1.h
- **lib** - contains library files (.a), _cw for CodeWarrior, _ghs for Green Hills, _wr for Wind River
- **project** - contains project and make files to rebuild the library
- **src** - contains library source files for CodeWarrior and Green Hills compilers
- **src\src_WR** - contains library source files for Wind River compiler
- **test** - contains test projects (optional)

4 How to use the library in a project

- CodeWarrior
  - add libdsp1_cw.a into your project window
  - add path to library include file libdsp1.h to “Target Settings (Alt-F7)->Access Paths->User Paths”
  - include file libdsp1.h to your source file
- Green Hills
  - use -l and -L linker options -llibdsp1_ghs.a -L{path to libdsp1_ghs.a}
  - use -I compiler option -I{path to libdsp1.h},
  - include file libdsp1.h to your source file

The -l, -L and -I options can be set in MULTI Project Builder menu “Edit->Set Options...->Basic Options->Project”

- Wind River
  - use -l and -L linker options -llibdsp1_wr.a -L{path to libdsp1_wr.a}
  - use -I compiler option -I{path to libdsp1.h}
  - include file libdsp1.h to your source file

Code example:

```
#include "libdsp1.h"

#define N 4
#define M 4
```

Comments:

```
/* vec_x and vec_y must be double-word aligned */
int vec_x[N] = {1073741824,-429496729,1288490188,1717986918};
int vec_y[N] = {-1073741824,1073741824,1503238553,1717986918};
int vec_out[M];


void main(void)
{
        corr_mn(N, M, vec_x, vec_y, vec_out);
}
```

5 How to rebuild the library

The project files needed to rebuild the library are stored in the “project” directory.

- **CodeWarrior**, open project libdsp1.mcp in the CodeWarrior IDE and press F7
- **Green Hills**, open project libdsp1.gpj in the MULTI Project Builder and press Ctrl+F7
- **Wind River**, open command prompt window, change current directory to {libdsp1 folder}\project and run dmake.exe

6 Function API

6.1 Convolutional encoder

Function call:

```
void conv_enc_mux(int n, unsigned short *message_ptr,
          unsigned long *out_ptr);
```

Arguments:

Table 1. conv_enc_mux arguments

n	in	size of input message in half-words, must be multiple of 2
message_ptr	in	pointer to input sequence of bits, each half-word contains bits b(15) b(14) ... b(0), must be half-word aligned memory layout: message_ptr(0) message_ptr(1) ... message_ptr(n-1)
out_ptr	out	pointer to output sequence of bits, each word contains bits b(31) b(30) ... b(0), must be word aligned memory layout: out_ptr(0) out_ptr(1) ... out_ptr(n-1)

Description: Computes rate 1/2 constraint length 9 convolutional encoder. Input sequence of bits is stored in message_ptr. Output sequence of bits is written to out_ptr.

Algorithm:

Generator polynomials are fixed: g_0 = 561 (octal), g_1 = 753 (octal). These are the polynomials given in the 3G specification.

Encoder output sequences G_0 and G_1 are given by:

Equation 1 $G\_0(k) = x(k) \oplus x(k-2) \oplus x(k-3) \oplus x(k-4) \oplus x(k-8)$

Equation 2 $G\_1(k) = x(k) \oplus x(k-1) \oplus x(k-2) \oplus x(k-3) \oplus x(k-5) \oplus x(k-7) \oplus x(k-8)$

for each k = 0, 1, ..., 16n-1

where $x(0), x(1), ..., x(16n-1)$ is the input message (i.e., input sequence of bits). Note that $x(-15), x(-14), ..., x(-1)$ are zero for 3G.

Output bit sequence is obtained by interleaving bit sequences G_0 and G_1.

Performance: See *Section 7* and *Table 16*.

Example1: conv_enc_mux

```
#define  K  8
unsigned short message[K];
unsigned long out[K];

conv_enc_mux(K, message, out);
```

6.2 Cross-correlation

Function call:

```
void corr_mn(int N, int M, int *x_ptr, int  *y_ptr, int *out_ptr);
```

Arguments:

Table 2. corr_mn arguments

N	in	length of the input vectors x_ptr and y_ptr, must be multiple of 4
M	in	length of the output sequence out_ptr, must be <= N, must be multiple of 4
x_ptr	in	pointer to the first input vector of length N, data type of each vector item is signed 32-bit fractional in range -1 to $1-2^{-31}$, vector must be double word aligned memory layout: x_ptr(0) x_ptr(1) ... x_ptr(N-1)
y_ptr	in	pointer to the second input vector of length N, data type of each vector item is signed 32-bit fractional in range -1 to $1-2^{-31}$, vector must be double word aligned memory layout: y_ptr(0) y_ptr(1) ... y_ptr(N-1)
out_ptr	out	pointer to the output vector of length M, data type of each vector item is signed 32-bit fractional in range -1 to $1-2^{-31}$, vector must be word aligned, memory layout: out_ptr(0) out_ptr(1) ... out_ptr(M-1)

Description: Computes cross-correlation of input vectors x_ptr and y_ptr. The result is written to vector out_ptr. All data types signed 32-bit fractional in range -1 to $1-2^{-31}$.

Algorithm:

Correlation:

Equation 3 $$out(j) = x(j)y(0) + x(j+1)y(1) + ... + x(N-1)y(N-j-1)$$

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

Note: The input data must be appropriately scaled to prevent overflow.

Performance: See *Section 7* and *Table 17*.

Example 2. corr_mn

```
#define  N   4
#define  M   4
int vector_x[N] = {1073741824,-429496729,1288490188,1717986918};
int vector_y[N] = {-1073741824,1073741824,1503238553,1717986918};
int result[M];
corr_mn(N, M, vector_x, vector_y, result);
```

6.3 Reduced lookup table for 16-bit CRC

Function call:

```
void crc_rdc_table(unsigned short gen_poly,
     unsigned short *rdc_table_ptr);
```

Arguments:

Table 3. crc_rdc_table arguments

gen_poly	in	bits g0, g1, ..., g15 of generator polynomial where $g(x) = x^{16} + g15 \cdot x^{15} + g14 \cdot x^{14} + ... + g1 \cdot x^{1} + g0$ For example, gen_poly = 0xa001 corresponds to $g(x) = x^{16} + g15 \cdot x^{15} + g2 \cdot x^{2} + g0$
rdc_table_ptr	out	pointer to reduced lookup table of length 16 memory layout: rdc_table_ptr(0) rdc_table_ptr(1) ... rdc_table_ptr(15)

Description: Creates a reduced lookup table for 16-bit CRC computation (for a reference see T. V. Ramabadran and S. S. Gaitonde, *A Tutorial on CRC Computations*, IEEE Micro, Vol. 8, Issue 4, August 1988, pp. 62-75).

Algorithm:

Let g(x) be a 16-bit CRC generator polynomial.

We have

Equation 4 $$g(x) = x^{16} + g15 \cdot x^{15} + g14 \cdot x^{14} + ... + g1 \cdot x^{1} + 1 \ ,$$

where coefficients g15, g14,..., g0 are equal to 0 or 1.

The 16-entry reduced lookup table is created as follows: divide x^{i+16} by g(x) and let s_i(x) to be the remainder. Set table entry i equal to s_i(x).

The values s_i(x) are determined using the standard feedback shift register for polynomial division.

Performance: See *Section 7* and *Table 15*.

Example 3. crc_rdc_table

```
#define GEN_POLY 0xa001
unsigned short rdc_table[16];
crc_rdc_table(GEN_POLY, rdc_table);
```

6.4 Standard lookup table for 16-bit CRC

Function call:

```
void crc_table(unsigned short *rdc_table_ptr,
     unsigned short *table_ptr);
```

Arguments:

Table 4. crc_table arguments

rdc_table_ptr	in	pointer to reduced lookup table of length 16 created by the crc_rdc_table function, must be half-word aligned memory layout: rdc_table_ptr(0) rdc_table_ptr(1) ... rdc_table_ptr(15)
table_ptr	out	pointer to standard lookup table of length 256, must be word aligned, memory layout: table_ptr(0) table_ptr(1) ... table_ptr(255)

Description: Creates a standard 256-entry lookup table for byte-wise 16-bit CRC computation (for a reference see T. V. Ramabadran and S. S. Gaitonde, *A Tutorial on CRC Computations*, IEEE Micro, Vol. 8, Issue 4, August 1988, pp. 62-75).

Algorithm:

For any message byte (t0, t1, ..., t7), table entry corresponds to byte's 16-bit CRC value, i.e. divide $x^{16}(t0 + t1x + ... + t7x^7)$ by generator polynomial g(x) and set remainder equal to CRC. Table has 256 16-bit entries.

The standard table is created from the first 8-entries of reduced lookup table as follows:

For each input byte (t0, t1, ..., t7) determine it's CRC by XORing the entries of the reduced lookup table corresponding to the nonzero bits of (t0, t1, ..., t7). For example, if message byte is (t0, t1, ..., t7) = (0, 0, 0, 0, 1, 0, 0, 1) = 0x9, then t4 and t7 are nonzero. So XOR reduced table entries rdc_table_ptr(4) and rdc_table_ptr(7) to obtain CRC.

Performance: See *Section 7* and *Table 15*.

Example 4. crc_table

```
unsigned short rdc_table[16];
unsigned short table[256];
crc_table(rdc_table, table);
```

6.5 16-bit CRC

Function call:

```
unsigned short crc_calc(int K, unsigned short *table_ptr,
       unsigned char *message_ptr);
```

Arguments:

Table 5. crc_calc arguments

K	in	message size in bytes, must be a multiple of 4
table_ptr	in	pointer to standard lookup table created by the crc_table function, must be word aligned memory layout: table_ptr(0) table_ptr(1) ... table_ptr(255)
message_ptr	in	pointer to input message bytes, must be word aligned memory layout: message_ptr(0) message_ptr(1) ... message_ptr(K-1)

Description: Function returns 16-bit CRC (Cyclic Redundancy Code) computed for input message using the standard table lookup algorithm (for a reference see T. V. Ramabadran and S. S. Gaitonde, *A Tutorial on CRC Computations*, IEEE Micro, Vol. 8, Issue 4, August 1988, pp. 62-75).

Algorithm:

1. Initialize 16-bit CRC register (s0, s1, ..., s15) to 0x0000.
2. For input byte (b0, b1, ..., b7) compute vector t:
   (t0, t1, ..., t7) = (b0, b1, ..., b7) XOR (s8, s9, ..., s15).
3. Right shift CRC register eight bits.
4. Table lookup: Look up value corresponding to (t0, t1, ..., t7) and XOR with CRC register.
5. Repeat steps 2.-4. for each message byte.

Performance: See *Section 7* and *Table 18*.

Example 5. crc_calc

```
#define GEN_POLY 0xa001
#define K 8
unsigned short rdc_table[16];
unsigned short table[256];
unsigned char message[K];
unsigned short crc16;

crc_rdc_table(GEN_POLY, rdc_table);
crc_table(rdc_table, table);
crc16 = crc_calc(K, table, message);
```

6.6 2-D 8x8 DCT

Function call:

```
void dct(short *in_matrix_ptr, short *constant_ptr,
     short *out_matrix_ptr);
```

Arguments:

Table 6. dct arguments

in_matrix_ptr	in	pointer to the 8x8 input matrix, data type of each matrix item is 8-bit unsigned integer in range 0 to 255 stored as 16-bit value, matrix must be word aligned

constant_ptr	in	pointer to the constant array, array must be word aligned, use dct_const array which is defined inside the library and its extern definition is in the libdsp1.h header file
		The library defines the dct_const as follows:
		const short dct_const[10] = {
		6269, /* sqrt(2)[sin(3pi/8) - cos(3pi/8)]2^13 */
		-15137, /* sqrt(2)[sin(3pi/8) + cos(3pi/8)]2^13 */
		4434, /* sqrt(2)[cos(3pi/8)]2^13 */
		11585, /* sqrt(2)2^13 */
		6811, /* [cos(3pi/16)]2^13 */
		8035, /* [cos(pi/16)]2^13 */
		-2260, /* [sin(3pi/16) - cos(3pi/16)]2^13 */
		-6436, /* [sin(pi/16) - cos(pi/16)]2^13 */
		-11363, /* [-sin(3pi/16) - cos(3pi/16)]2^13 */
		-9633, /* [-sin(pi/16) - cos(pi/16)]2^13 */
		};

out_matrix_ptr	out	pointer to the 8x8 output matrix, data type of each matrix item is signed 16-bit integer, matrix must be word aligned

Description: This code performs a 1-D DCT (discrete cosine transform) on each row of an 8x8 input matrix. The result of each row operation is stored in the corresponding column of an 8x8 output matrix. The 1-D DCT is implemented using the method described in C. Loeffler, A. Lightenberf, and G. Moschytzi, *Practical Fast 1-D DCT Algorithms with 11 Multiplications*, Proceedings International Conference on Acoustics, Speech, and Signal Processing, 1989, pp. 988-991.

To compute 2-D 8x8 DCT, two subsequent calls of the dct function are needed. The first call computes a 1-D DCT on the rows of the input matrix. The results of the 1-D DCT row operations are written column-wise to an output matrix. The output matrix is used as input to the second call of the dct function. The second call to the dct function performs a 1-D DCT on the rows of the output matrix. The results are written column-wise over the input matrix.

Algorithm:

1-D DCT definition (there is added multiplication by $\sqrt{8}$ against the true 1-D DCT):

Equation 5

$$y(k) = \sqrt{8} \cdot w(k) \cdot \sum_{n=0}^{N-1} x(n) \cdot \cos\left[\frac{\pi}{N} k\left(n + \frac{1}{2}\right)\right] \quad \text{for each n from 0 to N-1}$$

Equation 6

$$w(k) = \frac{1}{\sqrt{N}} \quad \text{for k = 0}$$

Equation 7 $w(k) = \sqrt{\frac{2}{N}}$ for $1 \le k \le N - 1$

where N is the length of input sequence.

Note: The final results of the 2-D 8x8 DCT are 8 times larger than the true 2-D DCT results.

Performance: See *Section 7* and *Table 15*.

Example 6. dct

```
short x[8][8];
short temp[8][8];

/* calculate 2-D 8x8 DCT on input matrix x, results stored to x */
dct(&x[0][0], dct_const, &temp[0][0]);
dct(&temp[0][0], dct_const, &x[0][0]);
```

6.7 256-point complex FFT, float data

Function call:

```
void fft256(float *io_buffer1, float *io_buffer2, float
*w_table_ptr);
```

Arguments:

Table 7. fft256 arguments

io_buffer1	in/out	pointer to the input/output vector of length 512, vector must be double word aligned memory layout: x_Re(0) x_Im(0) x_Re(1) x_Im(1) ... x_Re(255) x_Im(255) (Re - real part, Im - imaginary part)
io_buffer2	in	pointer to a vector of length 512, the vector is used by the routine as a temporary buffer during computations, vector must be double word aligned
w_table_ptr	in	pointer to the vector of twiddle factors of length 256, vector must be double word aligned memory layout: w_Re(0) w_Im(0) w_Re(1) w_Im(1) ... w_Re(127) w_Im(127) Use w_table256 array which is defined inside the library and its extern definition is in the libdsp1.h header file. The w_table256 is computed according to this formula: $w\_Re(k) + i \cdot w\_Im(k) = w_N^k$ for k = 0, 1, ..., 127

Description: Computes the 256-point complex fast fourier transform (FFT), single precision floating point data types. Input data are stored in io_buffer1, output data are written over the input data into io_buffer1.

Algorithm:

Equation 8 $X(n) = \sum_{k=0}^{N-1} x(k) \cdot w_N^{n \cdot k}$ for each n from 0 to N-1

Equation 9 $w_N = e^{-\frac{2\pi i}{N}}$

where i is the imaginary unit with the property that: $i^2 = -1$, N = 256.

Note: There must be at least 16 bytes of readable memory before vector io_buffer1.

Performance: See *Section 7* and *Table 15*.

Example 7. fft256

```
float io_buffer_1[512];
float io_buffer_2[512];

fft256(io_buffer_1, io_buffer_2, w_table256);
```

6.8 20 TAP real FIR filter

Function call:

```
void fir20(int N, short int *x_ptr, short int *y_ptr, short int
*h_ptr);
```

Arguments:

Table 8. fir20 arguments

N	in	length of the output vector, must be multiple of 2
x_ptr	in	pointer to input vector of length N + M - 1, data type of each sample is signed 16 bit fractional, range -1 to $1\text{-}2^{-15}$, vector must be word aligned memory layout: x(-M+1) x(-M+2) ... x(0) x(1) ... x(N-1)
y_ptr	out	pointer to output vector of length N, data type of each sample is signed 16 bit fractional, range -1 to $1\text{-}1\text{-}2^{-15}$, vector must be word aligned memory layout: y(0) y(1) ... y(N-1)
h_ptr	in	pointer to vector of FIR coefficients of length M, data type of each coefficient is signed 16 bit fractional, range -1 to $1\text{-}2^{-15}$, vector must be half-word aligned memory layout: h(0) h(1) h(2) ... h(M-1)

Description: Computes 20 TAP real FIR filter (M = 20). The coefficients of the filter are stored in vector h. Input samples are stored in vector x, output samples in vector y. 16-bit fractional data types with saturation.

Algorithm:

Equation 10 $y(n) = \sum_{k=0}^{M-1} h(k) \cdot x(n-k)$ for each n from 0 to N-1, M = 20

Note: The coefficients stored in vector h must be scaled by user in order to prevent saturation, i.e. the sum of the absolute values of all coefficients must be lower than 1:

Equation 11

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

Performance: See *Section 7* and *Table 19*.

Example 8. fir20

```
#define N 256
#define M 20
short int xvec[N + M - 1];
short int yvec[N];
short int hvec[M];

fir20(N, xvec, yvec, hvec);
```

6.9 22 TAP real FIR filter with symmetrical impulse response

Function call:

```
void fir22sym(int N, short *x, int *y, short *h);
```

Arguments:

Table 9. fir22sym arguments

N	in	length of the output vector, must be multiple of 2
x	in	pointer to input vector of length N + M - 1, data type of each sample is signed 16 bit integer, range -32768 to 32767, vector must be word aligned memory layout: x(-M+1) x(-M+2) ... x(0) x(1) ... x(N-1)
y	out	pointer to output vector of length N, data type of each sample is signed 32 bit integer, range -2^{31} to 2^{31} -1, vector must be word aligned memory layout: y(0) y(1) ... y(N-1)
h	in	pointer to vector of FIR coefficients of length M/2, data type of each coefficient is signed 16 bit integer, range -32768 to 32767, vector must be word aligned memory layout: h(0) h(1) h(2) ... h(M/2-1)

Description: Computes 22 TAP real FIR filter with symmetrical impulse response (i.e. coefficients of the filter are symmetrical), M = 22. Half of the filter coefficients are stored in vector h. Input samples are stored in vector x, output samples in vector y.

Algorithm:

Equation 12 $$y(n) = \sum_{k=0}^{M-1} h(k) \cdot x(n-k)$$ for each n from 0 to N-1, M = 22

Note: The coefficients stored in vector h must be scaled by user in order to prevent overflow, i.e. the sum of the absolute values of all coefficients must be lower than 2^{15}:

Equation 13 $$\sum_{k=0}^{M-1} |h(k)| < 2^{15}$$

The filter must have symmetrical coefficients:

Equation 14 $$h(n) = h\left(\frac{M}{2} - n\right)$$ for each n from M/2 to M-1

Performance: See *Section 7* and *Table 20*.

Example 9. fir22sym

```
#define N 256
#define M 22
short x[N + M - 1];
int   y[N];
short h[M/2];

fir22sym(N, x, y, h);
```

6.10 32 TAP real FIR Filter with symmetrical impulse response

Function call:

```
void fir32sym(int N, short *x, int *y, short *h);
```

Arguments:

Table 10. fir32sym arguments

N	in	length of the output vector, must be multiple of 2
x	in	pointer to input vector of length N + M - 1, data type of each sample is signed 16 bit integer, range -32768 to 32767, vector must be word aligned memory layout: x(-M+1) x(-M+2) ... x(0) x(1) ... x(N-1)

Table 10. fir32sym arguments (continued)

y	out	pointer to output vector of length N, data type of each sample is signed 32 bit integer, range -2^{31} to 2^{31} -1, vector must be word aligned memory layout: y(0) y(1) ... y(N-1)
h	in	pointer to vector of FIR coefficients of length M/2, data type of each coefficient is signed 16 bit integer, range -32768 to 32767, vector must be word aligned memory layout: h(0) h(1) h(2) ... h(M/2-1)

Description: Computes 32 TAP real FIR filter with symmetrical impulse response (i.e. the coefficients of the filter are symmetrical), M = 32. Half of the filter coefficients are stored in vector h. Input samples are stored in vector x, output samples in vector y.

Algorithm:

Equation 15

$$y(n) = \sum_{k=0}^{M-1} h(k) \cdot x(n-k) \qquad \text{for each n from 0 to N-1, M = 32}$$

Note: The coefficients stored in vector h must be scaled by user in order to prevent overflow, i.e. the sum of the absolute values of all coefficients must be lower than 2^{15}:

Equation 16

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

The filter must have symmetrical coefficients:

Equation 17

$$h(n) = h\left(\frac{M}{2} - n\right) \qquad \text{for each n from M/2 to M-1}$$

Performance: See *Section 7* and *Table 21*.

Example 10. fir32sym

```
#define N 256
#define M 32
short x[N + M - 1];
int   y[N];
short h[M/2];

fir32sym(N, x, y, h);
```

6.11 64 TAP real FIR filter with symmetrical impulse response

Function call:

```
void fir64sym(int N, short *x, int *y, short *h);
```

Arguments:

Table 11. fir64sym arguments

N	in	length of the output vector, must be multiple of 2
x	in	pointer to input vector of length N + M - 1, data type of each sample is signed 16 bit integer, range -32768 to 32767, vector must be word aligned memory layout: x(-M+1) x(-M+2) ... x(0) x(1) ... x(N-1)
y	out	pointer to output vector of length N, data type of each sample is signed 32 bit integer, range -2^{31} to 2^{31} -1, vector must be word aligned memory layout: y(0) y(1) ... y(N-1)
h	in	pointer to vector of FIR coefficients of length M/2, data type of each coefficient is signed 16 bit integer, range -32768 to 32767, vector must be word aligned memory layout: h(0) h(1) h(2) ... h(M/2-1)

Description: Computes 64 TAP real FIR filter with symmetrical impulse response (i.e. the coefficients of the filter are symmetrical), M = 64. Half of the filter coefficients are stored in vector h. Input samples are stored in vector x, output samples in vector y.

Algorithm:

Equation 18

$$y(n) = \sum_{k=0}^{M-1} h(k) \cdot x(n-k) \qquad \text{for each n from 0 to N-1, M = 64}$$

Note: The coefficients stored in vector h must be scaled by user in order to prevent overflow, i.e. the sum of the absolute values of all coefficients must be lower than 2^{15}:

Equation 19

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

The filter must have symmetrical coefficients:

Equation 20

$$h(n) = h\left(\frac{M}{2} - n\right) \qquad \text{for each n from M/2 to M-1}$$

Performance: See *Section 7* and *Table 22*.

Example 11. fir64sym

```
#define N 256
#define M 64
short x[N + M - 1];
int   y[N];
short h[M/2];

fir64sym(N, x, y, h);
```

6.12 2nd-order IIR filter

Function call:

```
void iir_2nd_o(int N, short int *coeff_ptr, short int *x_ptr, short
int *y_ptr);
```

Arguments:

Table 12. iir_2nd_o arguments

N	in	length of the input and output vector, must be >= 8, must be multiple of 4
x_ptr	in	pointer to input vector of length N, data type of each sample is signed 16 bit integer, range -32768 to 32767, vector must be word aligned memory layout: x(0) x(1) ... x(N-1)
y_ptr	out	pointer to output vector of length N, data type of each sample is signed 16 bit integer, range -32768 to 32767, vector must be word aligned memory layout: y(0) y(1) ... y(N-1)
coeff_ptr	in	pointer to vector of filter coefficients, data type of each coefficient is signed 16 bit integer, range -32768 to 32767, vector must be half-word aligned coefficients must be stored in this order: b(0) b(1) b(2) a(1) a(2)

Description: Computes direct form second-order IIR filter (M = 2). The coefficients of the filter are stored in vector coeff_ptr. Input samples are stored in vector x_ptr, output samples in vector y_ptr. Signed 16-bit integer data computation with saturation.

Algorithm:

Equation 21

$$y(n) = \sum_{k=0}^{M} b(k) \cdot x(n-k) - \sum_{k=1}^{M} a(k) \cdot y(n-k)$$

for each n from 0 to N-1, M = 2

Note: Initial conditions are assumed to be zero, i.e. $x(n) = y(n) = 0$ *for each n < 0.*

Performance: See *Section 7* and *Table 23*.

Example 12. iir_2nd_o

```
#define N 256
short int coeff[5];
short int vec_x[N];
short int vec_y[N];

iir_2nd_o(N, coeff, vec_x, vec_y);
```

6.13 LU decomposition

Function call:

```
void lu_decomp(int n, float *matrix_ptr, int *row_perm_ptr, int *d,
float *row_scale_ptr);
```

Arguments:

Table 13. lu_decomp arguments

n	in	dimension of input/output square matrix, must be >= 6, must be multiple of 2
matrix_ptr	in/out	pointer to the input/output matrix of size nxn, data type of each matrix element is float, matrix must be double word aligned
row_perm_ptr	out	pointer to the permutation vector of length n, vector must be word aligned, to understand the meaning of this argument see Chapter 2.3 of “Numerical Recipes in C”
d	out	pointer to integer variable, there is returned value +/-1 in this variable indicating that an even/odd number of row interchanges were performed
row_scale_ptr	out	pointer to the row scale vector of length n, vector must be double word aligned, to understand the meaning of this argument see Chapter 2.3 of “Numerical Recipes in C”

Description: Computes LU decomposition. The input matrix is stored in matrix_ptr. The output matrix is written over the input matrix matrix_ptr. The method used for the LU decomposition is described in Chapter 2.3 of “Numerical Recipes in C”, which can be downloaded from http://www.library.cornell.edu/nr/bookcpdf.html.

Algorithm:

For input matrix X, the code finds matrices L and U such that

Equation 22 $PX = LU$

where L is a lower triangular matrix, U is an upper triangular matrix and P is a permutation matrix, i.e. a matrix of zeros and ones that has exactly one entry 1 in each row and column.

The output matrix contains the L and U matrices:

Equation 23

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ l_{21} & u_{22} & u_{23} & \cdots & u_{2n} \\ l_{31} & l_{32} & u_{33} & \cdots & u_{3n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ l_{n1} & l_{n2} & l_{n3} & \cdots & u_{nn} \end{bmatrix}$$

To get L from the output matrix - set the lower elements of L to the same values as the lower elements of the output matrix. Set the diagonal elements of L equal to one. The upper elements of L are all zero.

Equation 24 $L = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ l_{21} & 1 & 0 & \dots & 0 \\ l_{31} & l_{32} & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ l_{n1} & l_{n2} & l_{n3} & \dots & 1 \end{bmatrix}$

To get U from the output matrix - set the upper elements of U to the same values as the upper elements of the output matrix. Set the diagonal elements of U to the same values as the diagonal elements of the output matrix. Set the lower elements of U to zero.

Equation 25 $U = \begin{bmatrix} u_{11} & u_{12} & u_{13} & \dots & u_{1n} \\ 0 & u_{22} & u_{23} & \dots & u_{2n} \\ 0 & 0 & u_{33} & \dots & u_{3n} \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & 0 & \dots & u_{nn} \end{bmatrix}$

Performance: See *Section 7* and *Table 24*.

Example 13. lu_decomp

```
#define N 8
float matrix[N][N];
int d;
int row_perm[N];
float row_scale[N];

lu_decomp(N, &matrix[0][0], row_perm, &d, row_scale);
```

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

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 IPC column shows instructions per cycle.

Table 14. Code size

Function	Code size [bytes]
conv_enc_mux	444
corr_mn	580
crc_calc	168
crc_rdc_table	312
crc_table	240
dct	412
fft256	1112
fir20	436
fir22sym	348
fir32sym	500
fir64sym	984
iir_2nd_o	404
lu_decomp	1300

Table 15. Clock cycles

Function	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
crc_rdc_table(1)	170	89	89	1.9	0.87
crc_table	6342	6296	6296	-	0.96
dct(2)	696	589	589	3.2	0.92
fft256(3)	13259	12494	12465	5.2	0.95

1. gen_poly = 0xA001

2. dct_const in Internal Flash
3. w_table256 in Internal Flash

Table 16. conv_enc_mux function

n(1)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
2	182	126	126	6.3	0.88
4	279	223	223	7.0	0.93
6	379	323	323	7.2	0.94
8	475	420	420	7.4	0.95
32	1657	1595	1595	7.7	0.97
256	12677	12573	12573	7.8	0.98

1. Size of input message in half-words

Table 17. corr_mn function

N(1)	M	First function call [clock cycles]	Second Function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
4	4	159	95	95	4.5	0.60
8	8	290	184	184	6.7	0.79
12	12	436	317	317	7.2	0.84
16	16	615	495	490	7.7	0.86
32	32	1710	1583	1583	8.1	0.88
256	256	84317	84070	84070	8.3	0.86

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

Table 18. crc_calc function

K(1)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
4	107	64	64	11.1	0.66
8	143	100	100	13.9	0.71
12	178	136	136	15.3	0.74
32	362	318	318	17.3	0.77
256	2393	2335	2333	18.8	0.80

1. Message size in bytes

Table 19. fir20 function

N[1]	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
2	261	160	160	11.3	0.68
4	313	214	214	16.8	0.74
6	366	264	264	20.4	0.77
8	417	316	316	22.7	0.80
32	1044	941	941	30.3	0.88
256	6896	6764	6764	33.7	0.92

1. Length of the output vector

Table 20. fir22sym function

N[1]	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
2	211	124	124	7.9	0.73
4	256	174	174	11.1	0.80
6	309	227	227	12.7	0.84
8	360	280	280	13.7	0.86
32	1000	917	917	16.6	0.92
256	6972	6852	6852	17.7	0.94

1. Length of the output vector

Table 21. fir32sym function

N[1]	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
2	286	177	173	8.1	0.74
4	380	252	246	11.2	0.80
6	445	322	322	12.8	0.83
8	524	395	395	13.9	0.86
32	1396	1271	1271	17.2	0.93
256	9613	9447	9447	18.5	0.95

1. Length of the output vector

Table 22. fir64sym function

N(1)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
2	508	300	300	9.1	0.82
4	682	480	480	11.4	0.87
6	862	657	657	12.4	0.90
8	1038	833	833	13.1	0.92
32	3168	2958	2958	14.7	0.96
256	23014	22782	22782	15.2	0.98

1. Length of the output vector

Table 23. iir_2nd_o function

N(1)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
8	220	142	142	1.8	0.71
12	259	178	178	2.2	0.73
16	291	212	212	2.5	0.75
100	1055	968	968	3.5	0.79
256	2482	2371	2371	3.7	0.80

1. Length of the output vector

Table 24. lu_decomp function

n(1)	First function call [clock cycles]	Second function call [clock cycles]	Third function call [clock cycles]	Improvement to C function [-]	IPC [-]
6	2278	2070	2070	10.8	0.71
8	4428	4159	4159	8.6	0.72
10	7664	7439	7439	7.1	-
20	48343	47928	47928	4.3	-

1. Dimension of input/output square matrix

8 Revision history

Table 25. Document revision history

Date	Revision	Changes
29-May-2008	1	Initial release
24-Sep-2013	2	Updated Disclaimer.

Please Read Carefully:

Information in this document is provided solely in connection with ST products. STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, modifications or improvements, to this document, and the products and services described herein at any time, without notice.

All ST products are sold pursuant to ST’s terms and conditions of sale.

Purchasers are solely responsible for the choice, selection and use of the ST products and services described herein, and ST assumes no liability whatsoever relating to the choice, selection or use of the ST products and services described herein.

No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted under this document. If any part of this document refers to any third party products or services it shall not be deemed a license grant by ST for the use of such third party products or services, or any intellectual property contained therein or considered as a warranty covering the use in any manner whatsoever of such third party products or services or any intellectual property contained therein.

UNLESS OTHERWISE SET FORTH IN ST’S TERMS AND CONDITIONS OF SALE ST DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY WITH RESPECT TO THE USE AND/OR SALE OF ST PRODUCTS INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE (AND THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION), OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

ST PRODUCTS ARE NOT DESIGNED OR AUTHORIZED FOR USE IN: (A) SAFETY CRITICAL APPLICATIONS SUCH AS LIFE SUPPORTING, ACTIVE IMPLANTED DEVICES OR SYSTEMS WITH PRODUCT FUNCTIONAL SAFETY REQUIREMENTS; (B) AERONAUTIC APPLICATIONS; (C) AUTOMOTIVE APPLICATIONS OR ENVIRONMENTS, AND/OR (D) AEROSPACE APPLICATIONS OR ENVIRONMENTS. WHERE ST PRODUCTS ARE NOT DESIGNED FOR SUCH USE, THE PURCHASER SHALL USE PRODUCTS AT PURCHASER’S SOLE RISK, EVEN IF ST HAS BEEN INFORMED IN WRITING OF SUCH USAGE, UNLESS A PRODUCT IS EXPRESSLY DESIGNATED BY ST AS BEING INTENDED FOR “AUTOMOTIVE, AUTOMOTIVE SAFETY OR MEDICAL” INDUSTRY DOMAINS ACCORDING TO ST PRODUCT DESIGN SPECIFICATIONS. PRODUCTS FORMALLY ESCC, QML OR JAN QUALIFIED ARE DEEMED SUITABLE FOR USE IN AEROSPACE BY THE CORRESPONDING GOVERNMENTAL AGENCY.

Resale of ST products with provisions different from the statements and/or technical features set forth in this document shall immediately void any warranty granted by ST for the ST product or service described herein and shall not create or extend in any manner whatsoever, any liability of ST.

ST and the ST logo are trademarks or registered trademarks of ST in various countries.
Information in this document supersedes and replaces all information previously supplied.
The ST logo is a registered trademark of STMicroelectronics. All other names are the property of their respective owners.

© 2013 STMicroelectronics - All rights reserved

STMicroelectronics group of companies

Australia - Belgium - Brazil - Canada - China - Czech Republic - Finland - France - Germany - Hong Kong - India - Israel - Italy - Japan - Malaysia - Malta - Morocco - Philippines - Singapore - Spain - Sweden - Switzerland - United Kingdom - United States of America

www.st.com