Freescale Semiconductor
Application Note

Document Number: AN3807
Rev. 0, 10/2009

MPC5500 Linear Algebra Function Library 1

by: Abraham Tezmol , Application Engineer Guadalajara, Mexico
Ioseph Martinez, Application Engineer Guadalajara, Mexico
Roberto de Alba, Technical Information Center Engineer Guadalajara, Mexico
Rebeca Delgado, Fields Application Engineer Detroit, USA
Juan Ramon Ramos, Intern

1 Introduction

The MPC5500 Linear Algebra Function Library 1 contains optimized functions for the MPC5500 family of processors with a signal processing engine (SPE APU) for commonly used linear algebra operations.

Contents

2 Library Functions

The library contains the following functions:

matrix_add_int — Addition of two 16-bit signed integer square matrices

matrix_subtract_int — Subtraction of two 16-bit signed integer square matrices

matrix_mult_int — Multiplication of two 16-bit signed integer square matrices

matrix_copy_int — Copy of a 16-bit signed integer square matrix

matrix_transpose_int — Transpose of a 16-bit signed integer square matrix

matrix_scalar_add_int — Addition of a 16-bit signed integer square matrix and a scalar

matrix_scalar_mult_int — Multiplication of a 16-bit signed integer square matrix and a scalar

matrix_inverse_udu_decomp_float — Product of a single precision floating-point column vector and inverse of a square positive-definite matrix by UD decomposition method

matrix_inverse_crout_decomp_float — Inverse of a single precision floating-point square matrix. Using Crout's decomposition method

matrix_mult_float — Multiplication of two single precision floating-point square matrices

matrix_conv_int16_to_float — Conversion of a 16-bit signed integer square matrix into a single precision floating-point square matrix

matrix_conv_int32_to_float — Conversion of a 32-bit signed integer square matrix into a single precision floating-point square matrix

matrix_conv_float_to_int16 — Conversion of a single precision floating-point square matrix into a 16-bit signed integer square matrix

matrix_conv_float_to_int32 — Conversion of a single precision floating-point square matrix into a 32-bit signed integer square matrix

3 Supported Compilers

The library was built and tested using the following compilers:

- CodeWarrior for the MPC5500 V2.1
- Green Hills MULTI for PowerPC v5.0.5

4 Directory Structure

This document contains a library user's manual.

CodeWarrior

- cw — Contains project files to rebuild the library.
  - bin — Contains the library file SPE_Linear_Algebra_cw.a.

- lcf — Linker command files.
- SPE_Linear_Algebra_Library_cw_Data — Contains project data relative to library build.
- src — Contains library source files for the CodeWarrior compiler.
  - include\ — Contains the function definitions header file SPE_Linear_Algebra.h.

Green Hills

- ghs — Contains the project and makes files to rebuild the library.
  - lib — Contains the library file SPE_Linear_Algebra_ghs.a.
  - src — Contains the library source files for the Green Hills compiler.
    - include — Contains the function definitions header file SPE_Linear_Algebra.h.

5 How to Use the Library in a Project

CodeWarrior

- Add the SPE_Linear_Algebra_cw.a into your project window.
- Add a path to the library and include the file SPE_Linear_Algebra.h to target settings (Alt-F7)->Access Paths->User Paths.
- Include the file SPE_Linear_Algebra.h to your source file

Green Hills

- Use –l and –L linker options –lSPE_Linear_Algebra_ghs.a and –L{path SPE_Linear_Algebra_ghs.a}
- Use –I compiler option, QPI{path to SPE_Linear_Algebra.h},
- Include the file SPE_Linear_Algebra.h to your source file
- The –l, –L, and –I options can be set in the MULTI Project Builder menu Edit->Set Options...->Basic
- Options->Project.

Code Example

```
#include "SPE_Linear_Algebra.h"
#define M 3
#define N 3
#define P 3

/* A and B must be word aligned */
short A[M][N] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
short B[N][P] = {10,11,12,13,14,15,16,17,18};

/* C must be double-word aligned */
int C[M][P] = {0};

void main(void)
{
    matrix_add_int(*A,*B,*C,3);
}
```

6 How to Rebuild the Library

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

- CodeWarrior — Open the project SPE_Linear_Algebra_Library_cw.mcp in the CodeWarrior IDE and press F7.
- Green Hills — Open the project SPE_Linear_Algebra_Library_ghs.gpj in the MULTI Project Builder and press Ctrl+F7.

7 API Function

7.1 Integer Matrix Addition

Function call — void matrix_add_int (int* Aptr, int* Bptr, long* Cptr, int N)

Arguments:

Aptr	in	• Pointer to input Matrix A. • A is an N x N square matrix of the 16-bit signed integers. • A has a memory alignment of 4 bytes.
Bptr	in	• Pointer to input Matrix B. • B is an N x N square matrix of the 16-bit signed integers. • B has a memory alignment of 4 bytes.
Cptr	out	• Pointer to output Matrix C. • C is an N x N square matrix of the 32-bit signed integers. • C has a memory alignment of 8 bytes.
N	in	• Size of input and output matrices given by N x N. • N supported = 2, 3, 4, 5, 6, 7, 8, 9, and 10.

Description — Addition of the 16-bit signed integer matrices *A* and *B*. The result is stored into the 32-bit signed integer matrix *C*.

Algorithm — If *A* and *B* are matrices of identical dimensions *m*-by-*n*, then their sum is an *m*-by-*n* matrix denoted as $C = A + B$. The addition is given by:

$$c_{i,j} = (a_{i,j} + b_{i,j}) \qquad \textbf{\textit{Eqn. 1}}$$

For each pair of *i* and *j* with $1 \le i \le m$ and $1 \le j \le n$.

See Section 8, “Performance.”

Example 1. matrix_add_int

```
short A[2][2] = {1,2,3,4};
short B[2][2] = {5,6,7,8};
int C[2][2] = {0,0,0,0};

matrix_add_int(*A,*B,*C,2);
```

7.2 Integer Matrix Subtraction

Function call — void matrix_subtract_int (int* Aptr, int* Bptr, long* Cptr, int N)

Arguments:

Aptr	in	• Pointer to input Matrix A. • A is an N x N square matrix of the 16-bit signed integers. • A has a memory alignment of 4 bytes.
Bptr	in	• Pointer to input Matrix B. • B is an N x N square matrix of the 16-bit signed integers. • B has a memory alignment of 4 bytes.
Cptr	out	• Pointer to output Matrix C. • C is an N x N square matrix of the 32-bit signed integers. • C has a memory alignment of 8 bytes.
N	in	• Size of input and output matrices given by N x N. • N supported = 2, 3, 4, 5, 6, 7, 8, 9, and 10.

Description — This is a subtraction of the 16-bit signed integer matrices *A* and *B*. The result is stored into the 32-bit signed integer matrix *C*.

Algorithm — If *A* and *B* are matrices of identical dimensions *m*-by-*n*, then their subtraction is an *m*-by-*n* matrix denoted by $C = A - B$. The subtraction is given by:

$$c_{i,j} = (a_{i,j} - b_{i,j}) \qquad \textit{Eqn. 2}$$

For each pair of *i* and *j* with $1 \le i \le m$ and $1 \le j \le n$.

See Section 8, “Performance.”

Example 2. matrix_subtract_int

```
short A[2][2] = {1,2,3,4};
short B[2][2] = {5,6,7,8};
int C[2][2] = {0,0,0,0};

matrix_subtract_int(*A,*B,*C,2);
```

7.3 Integer Matrix Multiplication

Function call — void matrix_mult_int (int* Aptr, int* Bptr, long* Cptr, int N)

Arguments:

Aptr	in	• Pointer to input Matrix A. • A is an N x N square matrix of the 16-bit signed integers. • A has a memory alignment of 4 bytes.
Bptr	in	• Pointer to input Matrix B. • B is an N x N square matrix of the 16-bit signed integers. • B has a memory alignment of 4 bytes.
Cptr	out	• Pointer to output Matrix C. • C is an N x N square matrix of the 32-bit signed integers. • C has a memory alignment of 8 bytes.
N	in	• Size of input and output matrices given by N x N. • N supported = 2, 3, 4, 5, 6, 7, 8, 9, and 10.

Description — This is a multiplication of the 16-bit signed integer matrices *A* and *B*. The result is stored into the 32-bit signed integer matrix *C*.

Algorithm — If *A* is an *m*-by-*n* matrix and *B* is an *n*-by-*p* matrix, then their product is an *m*-by-*p* matrix denoted by $C = A \cdot B$. The product is given by:

$$c_{i,j} = \sum_{r=1}^{n} a_{i,r} b_{i,r} = a_{i,1} b_{1,j} + a_{i,2} b_{2,j} + \ldots + a_{i,n} b_{n,j} \qquad \textbf{Eqn. 3}$$

For each pair of *i* and *j* with $1 \le i \le m$ and $1 \le j \le p$.

See Section 8, “Performance.”

Overflow notice — Due to the accumulative nature of the matrix multiplication operation, make sure that all the expected values can be stored in a 32-bit signed integer to avoid overflow (final result inaccuracy). A good rule of thumb to avoid overflow for a matrix multiplication is as follows:

$$|a_{i,j}| < \sqrt{\frac{2^{31}}{N}}, \quad |b_{i,j}| < \sqrt{\frac{2^{31}}{N}} \qquad \textbf{Eqn. 4}$$

For each pair of *i* and *j* with $1 \le i \le m$ and $1 \le j \le p$, where N is the size of the input and output matrices given by N x N.

The matrix multiplication code does not check for overflow conditions. It is your responsibility to check for the status of overflow high (OVH), overflow low (OV), summary overflow high (SOVH), and summary overflow low (SOV) flags in the signal processing/embedded floating-point status and control register (SPEFSCR) to determine if execution of this operation caused overflow. These flags indicate that a minimum of one element from the matrix C is not accurate.

Example 3. matrix_mult_int

```
short A[2][2] = {1,2,3,4};
short B[2][2] = {5,6,7,8};
int C[2][2] = {0,0,0,0};

matrix_mult_int(*A,*B,*C,2);
```

7.4 Integer Matrix Copy

Function call — void matrix_copy_int (int* Aptr, int* Bptr, int N)

Arguments:

Aptr	in	• Pointer to input Matrix A. • A is an N x N square matrix of the 16-bit signed integers. • A has a memory alignment of 8 bytes.
Bptr	out	• Pointer to output Matrix B. • B is an N x N square matrix of the 16-bit signed integers. • B has a memory alignment of 8 bytes.
N	in	• Size of input and output matrices given by N x N. • N supported = 2, 3, 4, 5, 6,7, 8, 9, and 10.

Description — Copy of the 16-bit signed integer matrix *A*. The result is stored into the 16-bit signed integer matrix *B*.

Algorithm — If *A* is a matrix of dimensions *m*-by-*n*, then its copy is denoted by B. The copy is given by:

$$b_{i,j} = a_{i,j} \tag{Eqn. 5}$$

For each pair of *i* and *j* with $1 \le i \le m$ and $1 \le j \le n$.

See Section 8, "Performance."

Example 4. matrix_copy_int

```
short A[2][2] = {1,2,3,4};
short B[2][2] = {0,0,0,0};

matrix_copy_int(*A,*B,2);
```

7.5 Integer Matrix Transpose

Function call — void matrix_transpose_int (int* Aptr, int* Bptr, int N)

Arguments:

Aptr	in	• Pointer to input Matrix A. • A is an N x N square matrix of the 16-bit signed integers. • A has a memory alignment of 4 bytes.
Bptr	out	• Pointer to output Matrix B. • B is an N x N square matrix of the 16-bit signed integers. • B has a memory alignment of 4 bytes.
N	in	• Size of input and output matrices given by N x N. • N supported = 2, 3, 4, 5, 6, 7, 8, 9, and 10.

Description — Transpose of the 16-bit signed integer matrix *A*. The result is stored into the 16-bit signed integer matrix *B*.

Algorithm — If *A* is a matrix of dimensions *m*-by-*n*, then its transpose is an *n*-by-*m* matrix denoted by $B = A^T$. The transpose is given by:

$$b_{i,j} = a_{j,i} \qquad \textbf{\textit{Eqn. 6}}$$

For each pair of *i* and *j* with $1 \le i \le m$ and $1 \le j \le n$.

See Section 8, “Performance.”

Example 5. matrix_transpose_int

```
short A[2][2] = {1,2,3,4};
short B[2][2] = {0,0,0,0};

matrix_transpose_int(*A,*B,2);
```

7.6 Integer Matrix Scalar Addition

Function call — void matrix_scalar_add_int (int* Aptr, int* Bptr, long* Cptr, int N)

Arguments:

Aptr	in	• Pointer to input Matrix A. • A is an N x N square matrix of the 16-bit signed integers. • A has a memory alignment of 4 bytes.
Bptr	in	• Pointer to input scalar b. • B is an 1 x 1 array of 16-bit signed integers. • B has a memory alignment of 4 bytes.
Cptr	out	• Pointer to output Matrix C. • C is an N x N square matrix of the 32-bit signed integers. • C has a memory alignment of 8 bytes.
N	in	• Size of input and output matrices given by N x N. • N supported = 2, 3, 4, 5, 6, 7, 8, 9, and 10.

Description — The addition of the 16-bit signed integer matrix *A* and a 16-bit signed integer scalar *b*. The result is stored in the 32-bit signed integer matrix *C*.

Algorithm — If *A* is an *m*-by-*n* matrix and *b* a scalar, then their sum is an *m*-by-*n* matrix denoted by $C = A + b$. The scalar addition is given by:

$$c_{i,j} = (a_{i,j} + b) \qquad \textbf{\textit{Eqn. 7}}$$

For each pair of *i* and *j* with $1 \le i \le m$ and $1 \le j \le n$.

See Section 8, “Performance.”

Example 6. matrix_scalar_add_int

```
short A[2][2] = {1,2,3,4};
short b[1][1] = {5};
int C[2][2] = {0,0,0,0};

matrix_scalar_add_int(*A,*b,*C,2);
```

7.7 Integer Matrix Scalar Multiplication

Function call — void matrix_scalar_mult_int (int* Aptr, int* Bptr, long* Cptr, int N)

Arguments:

Aptr	in	• Pointer to input Matrix A. • A is an N x N square matrix of the 16-bit signed integers. • A has a memory alignment of 4 bytes.
Bptr	in	• Pointer to input scalar b. • B is a 1 x 1 array of the 16-bit signed integers. • B has a memory alignment of 4 bytes.
Cptr	out	• Pointer to output Matrix C. • C is an N x N square matrix of the 32-bit signed integers. • C has a memory alignment of 8 bytes.
N	in	• Size of input and output matrices given by N x N. • N supported = 2, 3, 4, 5, 6, 7, 8, 9, and 10.

Description — The multiplication of the 16-bit signed integer matrix *A* and a 16-bit signed integer scalar *b*. The result is stored in the 32-bit signed integer matrix *C*.

Algorithm — If *A* is an *m*-by-*n* matrix and *b* a scalar, then their product is an *m*-by-*n* matrix denoted by $C = A \cdot b$. The scalar multiplication is given by:

$$c_{i,j} = (a_{i,j} \cdot b) \qquad \textbf{Eqn. 8}$$

for each pair of *i* and *j* with $1 \le i \le m$ and $1 \le j \le n$.

See Section 8, "Performance."

Example 7. matrix_scalar_mult_int

```
short A[2][2] = {1,2,3,4};
short b[1][1] = {5};
int C[2][2] = {0,0,0,0};

matrix_scalar_mult_int(*A,*b,*C,2);
```

7.8 Floating Point Matrix Inverse by UD Decomposition

Function call — void matrix_inverse_udu_decomp_float (float* Aptr, float* Hptr, float* Uprt, float* Dptr, float* X1ptr, float* X2ptr, float* X3ptr, int N)

Arguments:

Aptr	in	• Pointer to input Matrix A. • A is an N x N square positive definite matrix of the single precision floating point numbers. • A has a memory alignment of 4 bytes.
Hptr	in	• Pointer to input column vector H (named output vector or output matrix). • H is an N x 1 column vector of the single precision floating point numbers limited to [2^16, -2^16]. • H has a memory alignment of 4 bytes.
Uptr	out	• Pointer to output Matrix U. • U is an N x N empty square matrix of single precision floating point numbers. • U is a component of matrix A, based on UDU' decomposition algorithm. • U has a memory alignment of 4 bytes.
Dptr	out	• Pointer to output Matrix D. • D is an N x N empty square matrix of single precision floating point numbers. • D is a component of matrix A, based on UDU' decomposition algorithm. • D has a memory alignment of 4 bytes.

X1ptr	out	• Pointer to output column vector X1. • X1 holds the first stage solution, computed by using forward substitution. • X1 is an N x 1 empty column vector of single precision floating point numbers. • X1 has a memory alignment of 4 bytes.
X2ptr	out	• Pointer to output column vector X2. • X2 holds the second stage solution, computed by using scalar substitution. • X2 is an N x 1 empty column vector of single precision floating point numbers. • X2 has a memory alignment of 4 bytes.
X3ptr	out	• Pointer to output column vector X3. • X3 holds the product of inv(A)*H, computed by using backward substitution. • X3 is an N x 1 empty column vector of single precision floating point numbers. • X3 has a memory alignment of 4 bytes.
N	in	• Size of matrices A, U, and D given by N x N. • Number of rows of column vectors H, X1, X2, and X3. • N supported = 2, 3, 4, 5, 6, 7, 8, 9, and 10.

Description — Computes the product of $A^{-1}*H$ by means of the U-D factorization approach, where A^{-1} is the inverse matrix of A and H is a column vector. A is a single precision floating-point positive definite square matrix and H is a single precision floating-point column vector.

Final product of $A^{-1}*H$ is stored into single precision floating point vector *X3*. Intermediate stage solutions are stored into single precision floating point vectors *X1* and *X2*.

Algorithm —

See Section 8, “Performance.”

Example 8. matrix_inverse_udu_decomp_float

```
float A[2][2] = {5, 11, 11, 25};
float H[2][1] = {1, 2};
float U[2][2] = {0, 0, 0, 0};
float D[2][2] = {0, 0, 0, 0};
float X1[2][1] = {0, 0};
float X2[2][1] = {0, 0};
float X3[2][1] = {0, 0};

matrix_inverse_udu_decomp_float(*A,*H,*U,*D,*X1,*X2,*X3,2);
```

7.9 Floating Point Matrix Inverse by Crout Decomposition

Function call — void matrix_inverse_crout_decomp_float (float* Aptr, float* Uprt, float* Lptr, float* Zptr, float* A_inv_ptr, int N)

Arguments:

Aptr	in	• Pointer to input Matrix A. • A is an N x N square non-singular matrix of the single precision floating point numbers. • A has a memory alignment of 4 bytes.
Uptr	out	• Pointer to output Matrix U. • U is an N x N empty square matrix of the single precision floating point numbers. • U is a component of matrix A, based on the Crout UL decomposition algorithm. • U has a memory alignment of 4 bytes.
Lptr	out	• Pointer to output Matrix L. • L is an N x N empty square matrix of single precision floating point numbers. • L is a component of matrix A, based on the Crout UL decomposition algorithm. • L has a memory alignment of 4 bytes.
Zptr	out	• Pointer to output column vector Z. • Z holds the first stage auxiliary output, computed by using forward substitution. • Z is an N x 1 empty column vector of single precision floating point numbers. • Z has a memory alignment of 4 bytes.
A_inv_ptr	out	• Pointer to output matrix A_inv. • A_inv holds the inverse of input matrix A, computed by using Crout decomposition. • A_inv is an N x N empty square matrix of single precision floating point numbers. • A_inv has a memory alignment of 4 bytes.
N	in	• Size of matrices A, U, L, and A_inv given by N x N. • Number of rows of column vectors Z. • N supported = 2, 3, 4, 5, 6, 7, 8, 9, and 10.

Description — Computes inverse matrix (A^{-1}) of the single precision floating-point non-singular square matrix *A* by means of the Crout LU decomposition approach.

A^{-1} is stored into single precision floating point matrix *A_inv*. The intermediate auxiliary stage solution is stored into the single precision floating point vector *Z*.

Algorithm —

See Section 8, "Performance."

Example 9. matrix_inverse_crout_decomp_float

```
float A[2][2] = {-107.59, 5254.1, -110, -1932.3};
float U[2][2] = {0, 0, 0, 0};
float L[2][2] = {0, 0, 0, 0};
float Z[2][1] = {0, 0};
float A_INV[2][2] = {0, 0, 0, 0};

matrix_inverse_crout_decomp_float(*A,*U,*L,*Z,*A_INV,2);
```

7.10 Floating Point Matrix Multiplication

Function call — void matrix_mult_float (float* Aptr, float* Bptr, float* Cptr, int N)

Arguments:

Aptr	in	• Pointer to input Matrix A. • A is an N x N square matrix of single precision floating point numbers. • A has a memory alignment of 8 bytes.
Bptr	in	• Pointer to input Matrix B. • B is an N x N square matrix of single precision floating point numbers. • B has a memory alignment of 8 bytes.
Cptr	out	• Pointer to output Matrix C. • C is an N x N square matrix of single precision floating point numbers. • C has a memory alignment of 8 bytes.
N	in	• Size of input and output matrices given by N x N. • N supported = 2, 3, 4, 5, 6, 7, 8, 9, and 10.

Description — The multiplication of single precision floating point matrices *A* and *B*. The result is stored into the single precision floating point matrix *C*.

Algorithm — If *A* is an *m*-by-*n* matrix and *B* is an *n*-by-*p* matrix, then their product is an *m*-by-*p* matrix denoted by $C = A \cdot B$. The product is given by:

$$c_{i,j} = \sum_{r=1}^{n} a_{i,r} b_{i,r} = a_{i,1} b_{1,j} + a_{i,2} b_{2,j} + \ldots + a_{i,n} b_{n,j} \qquad \textit{Eqn. 9}$$

For each pair of *i* and *j* with $l \le i \le m$ and $l \le j \le m$.

See Section 8, “Performance.”

Example 10. matrix_mult_float

```
float A[2][2] = {1.2, 2.8, 3.24, 4.96};
float B[2][2] = {5.73, 6.33, 7.0, 8};
float C[2][2] = {0, 0, 0, 0};

matrix_mult_float(*A,*B,*C,2);
```

7.11 Matrix Conversion from the 16-bit Integer to Floating Point

Function call — void matrix_conv_int16_to_float (int* Aptr, float* Bptr, int N)

Arguments:

Aptr	in	• Pointer to input Matrix A. • A is an N x N square matrix of the 16-bit signed integers. • A has a memory alignment of 8 bytes.
Bptr	out	• Pointer to output Matrix B. • B is an N x N square matrix of the single precision floating point numbers. • B has a memory alignment of 8 bytes.
N	in	• Size of input and output matrices given by N x N. • N supported = 2, 3, 4, 5, 6, 7, 8, 9, and 10.

Description — The conversion of the 16-bit signed integer matrix *A* into the single precision floating point matrix *B*.

Algorithm — Each 16-bit signed integer element of *A* is converted to the nearest single-precision floating-point value using the current rounding mode. The results are then placed into the corresponding element of *B*.

See Section 8, “Performance.”

Example 11. matrix_conv_int16_to_float

```
short A[2][2] = {32766, -32766, 32767, -32767};
float B[2][2] = {0, 0, 0, 0};

matrix_conv_int16_to_float(*A,*B,2);
```

7.12 Matrix Conversion from the 32-bit Integer to Floating Point

Function call — void matrix_conv_int32_to_float (long* Aptr, float* Bptr, int N)

Arguments:

Aptr	in	• Pointer to input Matrix A. • A is an N x N square matrix of the 32-bit signed integers. • A has a memory alignment of 8 bytes.
Bptr	out	• Pointer to output Matrix B. • B is an N x N square matrix of the single precision floating point numbers. • B has a memory alignment of 8 bytes.
N	in	• Size of input and output matrices given by N x N. • N supported = 2, 3, 4, 5, 6, 7, 8, 9, and 10.

Description — The conversion of the 32-bit signed integer matrix *A* into the single precision floating point matrix *B*.

Algorithm — Each 32-bit signed integer element of *A* is converted to the nearest single-precision floating-point value using the current rounding mode and the results are placed into the corresponding element of *B*.

See Section 8, “Performance.”

Example 12. matrix_conv_int32_to_float

```
int     A[2][2] = {132766, -32766, 62767, -82767};
float B[2][2] = {0, 0, 0, 0};

matrix_conv_int32_to_float(*A,*B,2);
```

7.13 Matrix Conversion from the Floating Point to the 16-bit Integer

Function call — void matrix_conv_float_to_int16 (float* Aptr, int* Bptr, int N)

Arguments:

Aptr	in	• Pointer to input Matrix A. • A is an N x N square matrix of the single precision floating point numbers. • A has a memory alignment of 8 bytes.
Bptr	out	• Pointer to output Matrix B. • B is an N x N square matrix of the 16-bit signed integers. • B has a memory alignment of 8 bytes.
N	in	• Size of input and output matrices given by N x N. • N supported = 2, 3, 4, 5, 6, 7, 8, 9, and 10.

Description — The conversion of a single precision floating point matrix *A* into the 16-bit signed integer matrix *B*.

Algorithm — Each single-precision floating-point element in *A* is converted to a signed integer using the current rounding mode and the result is saturated if it cannot be represented in a 16-bit integer. NaNs are converted as if they were zero.

The default rounding mode is round to the nearest. Other rounding modes can be selected by changing the value of the embedded floating-point rounding mode control (FRMC) in the signal processing and embedded floating-point status and control register (SPEFSCR).

See Section 8, "Performance."

Example 13. matrix_conv_float_to_int16

```
float A[2][2] = {36.03, 0.8491, -93.5473, 100.2266};
short B[2][2] = {0, 0, 0, 0};

matrix_conv_float_to_int16(*A,*B,2);
```

7.14 Matrix Conversion from Floating Point to 32-bit Integer

Function call — void matrix_conv_float_to_int32 (float* Aptr, long* Bptr, int N)

Arguments:

Aptr	in	• Pointer to input Matrix A. • A is an N x N square matrix of the single precision floating point numbers. • A has a memory alignment of 8 bytes.
Bptr	out	• Pointer to output Matrix B. • B is an N x N square matrix of the 32-bit signed integers. • B has a memory alignment of 8 bytes.
N	in	• Size of input and output matrices given by N x N. • N supported = 2, 3, 4, 5, 6, 7, 8, 9, and 10.

Description — Conversion of the single precision floating point matrix *A* into the 32-bit signed integer matrix *B*.

Algorithm — Each single-precision floating-point element in *A* is converted to a signed integer using the current rounding mode, the result is also saturated if it cannot be represented in a 32-bit integer. NaNs are converted as if they were zero.

Default rounding mode is round to the nearest. Other rounding modes can be selected by changing the value of the embedded *FRMC* in the *SPEFSCR*.

See Section 8, “Performance.”

Example 14. matrix_conv_float_to_int32

```
float A[2][2] = {36.03, 0.8491, -93.5473, 100.2266};
int   B[2][2] = {0, 0, 0, 0};

matrix_conv_float_to_int32(*A,*B,2);
```

8 Performance

Table 1. Code size per function

		Code Size (Bytes)								
	Function Name	**2x2**	**3x3**	**4x4**	**5x5**	**6x6**	**7x7**	**8x8**	**9x9**	**10x10**
1	matrix_add_int	204	252	228	292	240	308	260	308	276
2	matrix_subtract_int	204	252	300	380	244	292	260	292	276
3	matrix_mult_int	216	340	276	364	300	408	260	452	400
4	matrix_copy_int	180	196	204	216	216	216	224	232	232
5	matrix_transpose_int	204	244	224	232	240	248	256	264	272
6	matrix_scalar_add_int	200	236	220	232	232	264	244	264	256
7	matrix_scalar_mult_int	200	236	220	232	232	264	244	264	256
8	matrix_inverse_udu_decomp_float	240	340	572	880	1380	1948	572	3540	572
9	matrix_inverse_crout_decomp_float	520	860	956	956	956	956	956	956	956
10	matrix_mult_float	228	388	288	332	320	388	352	420	392
11	matrix_conv_int16_to_float	196	232	216	240	228	248	240	260	252
12	matrix_conv_int32_to_float	196	232	216	248	228	248	240	260	252
13	matrix_conv_float_to_int16	196	232	216	240	228	248	240	260	252
14	matrix_conv_float_to_int32	196	232	216	240	228	240	240	260	252

Table 2. Flash execution time enabled by code cache

			Execution Time (Clock Cycles)								
	Iteration	**Function Name**	**2x2**	**3x3**	**4x4**	**5x5**	**6x6**	**7x7**	**8x8**	**9x9**	**10x10**
1	1	matrix_add_int	22	45	78	118	150	189	238	298	360
	2		12	28	62	90	126	167	214	274	326
	3		12	28	62	90	126	167	214	274	326
2	1	matrix_subtract_int	22	44	76	122	138	193	231	293	340
	2		12	28	44	66	114	164	206	264	306
	3		12	28	44	66	114	164	206	264	306
3	1	matrix_mult_int	31	82	177	405	395	791	651	1422	1126
	2		17	56	148	356	360	734	594	1354	1066
	3		17	56	148	356	360	734	594	1354	1066
4	1	matrix_copy_int	12	29	21	51	49	81	85	115	116
	2		6	14	14	37	41	67	67	97	101
	3		6	14	14	37	41	67	67	97	101
5	1	matrix_transpose_int	24	60	86	122	153	215	270	319	389
	2		18	38	67	103	137	192	243	294	365
	3		18	38	67	103	137	192	243	294	365
6	1	matrix_scalar_add_int	24	45	68	93	109	140	185	212	271
	2		13	28	49	75	90	118	159	190	247
	3		13	28	49	75	90	118	159	190	247
7	1	matrix_scalar_mult_int	28	53	69	100	109	142	185	214	271
	2		17	38	51	82	90	120	159	192	247
	3		17	38	51	82	90	120	159	192	247
	3		22	47	69	102	134	184	223	285	341
8	1	matrix_inverse_udu_decomp_float	79	153	888	548	883	1269	3463	2342	5570
	2		65	147	816	520	826	1190	3363	2172	5475
	3		65	147	816	520	826	1190	3363	2172	5475
9	1	matrix_inverse_crout_decomp_float	253	410	1972	3106	4668	6653	9203	12272	16033
	2		184	313	1822	2966	4533	6509	9036	12120	15856
	3		184	313	1822	2966	4533	6509	9036	12120	15856
10	1	matrix_mult_float	35	118	221	494	577	1054	1187	1932	2195
	2		26	98	197	456	544	1009	1143	1877	2132
	3		26	98	197	456	544	1009	1143	1877	2132
11	1	matrix_conv_int16_to_float	20	46	55	83	107	146	168	213	264

Table 2. Flash execution time enabled by code cache (continued)

	2		16	37	50	68	89	130	158	191	243
	3		16	37	50	68	89	130	158	191	243
12	1	matrix_conv_int32_to_float	25	43	58	80	104	136	161	216	266
	2		16	37	52	68	88	124	148	203	254
	3		16	37	52	68	88	124	148	203	254
13	1	matrix_conv_float_to_int16	25	47	65	82	107	135	165	189	267
	2		16	37	52	66	88	121	148	210	244
	3		16	37	52	66	88	121	148	210	244
14	1	matrix_conv_float_to_int32	22	58	61	87	102	143	166	226	268
	2		16	37	52	64	88	120	148	201	254
	3		16	37	52	64	88	120	148	201	254

9 References

G. J. Bierman, *Measurement Updating Using the U-D Factorization* — Proceedings of the 1975 IEEE Conference on Decision and Control, pp. 337-346.

Press, W. H.; Flannery, B. P.; Teukolsky, S. A.; and Vetterling, W. T *Numerical Recipes in FORTRAN: The Art of Scientific Computing, 2nd edition* — Cambridge University Press. pp. 36-38, 1992.

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3807
Rev. 0
10/2009