Freescale Semiconductor
Application Note

Document Number: AN4786
Rev. 0, 08/2013

MEPL Quick Start Guide

Using the Mentor Embedded Performance Library for Freescale AltiVec Technology (MEPL)

This document explains how to download and use the functions of the Mentor Embedded Performance Library for Freescale AltiVec Technology (MEPL), which can be downloaded from the Mentor Graphics website.

By using AltiVec technology, most functions execute much faster than other libraries and standard C code. This can be most helpful in time sensitive applications, such as in aerospace and military fields. The acceleration created by AltiVec technology when using the MEPL has been proven.

This guide demystifies the MEPL functions and the requirements and data types for each function's input parameters. The guide describes how to initialize input parameters, how to pass in the variables, and the steps needed to create the desired output. There is a section on signal processing functions and a section filled with example code using several of the functions provided in this library.

For easy reference, the function topics in this guide appear in the same order as in the Mentor Embedded Performance Library Reference Manual.

Contents

1 Downloading and linking

1.1 Downloading

To find everything you need to download, go to mentor.com then search for ‘Mentor Embedded Performance Library.’ The library should then be downloaded to a file titled ‘install directory.’ Then the appropriate links can be made, which are explained next.

1.2 Linking

The linking necessary for your code to compile goes as follows:

1. In your c source code:

```
#include <mepl.h>
```

This should cover everything provided in the MEPL, because this header file contains:

```
#include <mepl/types.h>
#include <mepl/complex.h>
#include <mepl/math.h>
#include <mepl/misc.h>
#include <mepl/signal.h>
#include <mepl/image.h>
#include <mepl/random.h>
#include <mepl/vector.h>
```

2. In your gcc compiler command line:

```
-I/<install directory>/MEPL/bin/mepl-1.0/e6500-32/include
```

or

```
-I/<install directory>/MEPL/bin/mepl-1.0/e6500-64/include
```

3. In your gcc linker command line:

```
-L/<install directory>/MEPL/bin/mepl-1.0/e6500-32/lib -lcblas -latlas –lmepl
```

or

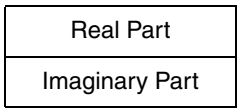
```
-L/<install directory>/MEPL/bin/mepl-1.0/e6500-64/lib -lcblas -latlas –lmepl
```

2 Input parameters for common data types

The data types used in the MEPL are a bit confusing. In the MEPL header files, the most common input parameters are defined as follows:

```
typedef signed long int mepl_index;
typedef unsigned long int mepl_length;
typedef unsigned long int mepl_stride;
typedef float complex mepl_cfloat;
typedef double complex mepl_cdouble;
```

These data types are straight forward, except when it comes to the complex data types, `mepl_cfloat` and `mepl_cdouble`. Both data types are interleaved complex numbers. In other words, one number will have two elements: one real, one imaginary. One interleaved complex number in memory will resemble the following structure:



When declaring a vector or matrix with the type `mepl_cfloat` or `mepl_cdouble`, they can be declared using the following macros supplied by the MEPL library. These are given in the reference manual, but they are worth repeating:

```
/* Construct a mepl_cfloat from two floats */
#define mepl_cfloat(r,i) ...
/* Construct a mepl_cdouble from two doubles */
#define mepl_cdouble(r,i) ...

/* Return the real component of a complex number */
float mepl_real_cf(mepl_cfloat v);
double mepl_real_cd(mepl_cdouble v);

/* Return the imaginary component of a complex number */
float mepl_imag_cf(mepl_cfloat v);
double mepl_imag_cd(mepl_cdouble v);

/* Return the magnitude of a complex number */
float mepl_cmag_f(mepl_cfloat v);
double mepl_cmag_d(mepl_cdouble v);

/* Return the conjugate of a complex number */
mepl_cfloat mepl_conj_f(mepl_cfloat v);
mepl_cdouble mepl_conj_d(mepl_cdouble v);
```

3 Naming conventions

The MEPL reference manual explains the library naming conventions. In this section, the MEPL naming conventions are explained in further detail. Once the conventions are understood, it is much easier to decipher what inputs are needed.

Each function name has the same format, unless noted otherwise. The format is given below:

`mepl_`*`function_p`*

The *_p* at the end of the function name determines the primary input data type. This convention is used through this entire document, so when an italicized P is seen it is referring to a data type. In addition, any italicized text should be taken as a placeholder. If any text is italicized, it should be replaced with the appropriate function name or data type. For example:

> When transposing a matrix filled with floating point numbers, the function name `mepl_`*`function_p`* will become `mepl_transpose_f`.

The different data types used in this library are given in this table.

Table 1. MEPL data types

Suffix	Meaning
`_i`	Signed integer (`int`)
`_u`	Unsigned integer (`unsigned int`)
`_f`	Single-precision floating point (`float`)
`_d`	Double-precision floating point (`double`)
`_cf`	Interleaved single-precision complex (`mepl_cfloat`)
`_cd`	Interleaved double-precision complex (`mepl_cdouble`)
`_zf`	Split single-precision complex (represented as a pair of `float` arrays)
`_zd`	Split double-precision complex (represented as a pair of `double` arrays)

Therefore, anytime one of these suffixes is seen, it is referring to one of the types given above. The first two, meaning `int` and `unsigned int`, are not to be confused with `mepl_index`, `mepl_length`, or `mepl_stride`. The latter data types were defined as those specific names because they correspond to their uses. In other words, when a parameter is passed to a function as `mepl_index`, the user will know the parameter associates with the index of the input vector or matrix. Likewise, `mepl_length` corresponds to the length of an input vector or matrix and `mepl_stride` corresponds to the stride of an input vector or matrix.

Another confusing aspect of the data types is the difference between interleaved complex values and split complex values. An interleaved element would contain both the real and imaginary part of a complex number, while a split complex value will have two separate elements for the real and imaginary part of a complex number. When there is a ‘z’ in the suffix, it can be thought of as a zipper separating the complex number into two parts. Here’s an example:

Interleaved complex array with *k* elements

Real [0]
Imaginary [0]
Real [1]
Imaginary [1]
...
Real [k]
Imaginary [k]

Split-Complex arrays with *k* elements

Real Array
Real [0]
Real [1]
Real [2]
...
Real[k]

Imaginary Array
Imaginary [0]
Imaginary[1]
Imaginary [2]
...
Imaginary[k]

In addition, any names used for variables correlate to the ones used in the reference manual. Therefore, if you were to look from the reference manual to this guide, the names will correspond. This is meant to create less confusion and aid in the process of using the MEPL.

4 Signal processing functions

Transformations, impulse responses, and convolution, which make up a majority of the signal processing functions, are all computed with three smaller functions: create, compute, and destroy. These three functions are used in that order, but will be explained in the same order as discussed in the MEPL reference manual:

1. Create
2. Destroy
3. Compute

The remaining signal processing functions fall under windowing functions and miscellaneous functions and will be explained in this order:

4. Windowing Functions
5. Miscellaneous Functions

4.1 Create functions

Create functions construct objects used in computing transformations, impulse responses, and convolution. These objects are data structures containing pertinent information needed to compute the signal processing functions. The names for each of the create functions are as follows:

`mepl_`*`function`*`_create_`*`p`*

The create functions are the first to be called out of the three functions for transformations, impulse responses, and convolution. In the next section, we see how to properly declare and initialize the input parameters for create functions, then describe how to use the create function. Below is a table of the inputs.

Table 2. Create functions with inputs

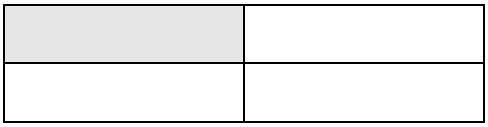
Create Functions	Inputs
1D Discrete Fourier Transform (DFT)	`mepl_length l, mepl_dft_direction d`
Multiple 1D DFT	`mepl_length rows, mepl_length cols, mepl_dftm_direction d, mepl_axis a`

Table 2. Create functions with inputs (continued)

Create Functions	Inputs
2D DFT	`mepl_length rows, mepl_length cols, mepl_dft2d_direction d`
Complex-to-Real or Real-to-Complex DFT	`mepl_length l`
Multiple 1D Complex-to-Real or Real-to-Complex DFT	`mepl_length rows, mepl_length cols, mepl_axis a`
2D Complex-to-Real or Real-to-Complex DFT	`mepl_lenth rows, mepl_length cols`
1D Discrete Cosine Transform (DCT)	`mepl_length l`
1D Discrete Sine Transform (DST)	`mepl_length l`
Multiple 1D DCT or DST	`mepl_length rows, mepl_length cols, mepl_axis a`
2D DCT or DST	`mepl_length rows, mepl_length cols`
Finite Impulse Response (FIR)	`P const* kernel, mepl_length K, mepl_length N, mepl_length D`
Infinite Impulse Response (IIR)	`P const* A, mepl_stride A_stride, P const* B, mepl_stride B_stride, mepl_length num_filters, mepl_length N`
Convolution	`P const* H, mepl_length H_length, mepl_length X_length, mepl_length decimation`
2D Convolution	`P const* H, mepl_length H_rows, mepl_length H_cols, mepl_length X_rows, mepl_length X_cols, mepl_length decimation`

4.1.1 Declaring and initializing create functions

Each of the inputs in the table above describes the dimensions of the input vector(s) or matrix. When the input is a matrix, the `P const* H` passed in will be a pointer to the first element in the top corner of the matrix. For a 4x4 matrix, it points to the highlighted element below:



The other variables involved are the length of the matrix (`l`), the direction in which to compute a transform (`d`), the number of rows in the matrix (`rows`), the number of columns in the matrix (`cols`), the stride between rows (`H_stride`) and the axis on which to perform a transform (`a`). When calling a function that requires a direction or axis specification, use the following type-definitions:

```
typedef enum { MEPL_DFT_FORWARD = -1, MEPL_DFT_INVERSE = 1
     }mepl_dft_direction;
typedef enum { MEPL_BY_ROW = 0, MEPL_BY_COL = 1 }mepl_axis;
```

For a vector with N elements, an example declaration would be:

```
mepl_length l = N;
mepl_dft_direction d = MEPL_DFT_FORWARD;                    //or MEPL_DFT_INVERSE
float* H[N];
```

For an MxN matrix, an example declaration would be:

```
mepl_dft_direction d = MEPL_DFT_FORWARD;                    //or MEPL_DFT_INVERSE
mepl_length rows = M;
mepl_length cols = N;
mepl_length H_stride = N;
mepl_axis a = MEPL_BY_ROW;                                       //or MEPL_BY_COL
float* H[rows][cols];
```

For an NxN matrix, an example declaration would be:

```
mepl_length row = N, cols = N, H_stride = N;
mepl_dft_direction d = MEPL_DFT_FORWARD;                    //or MEPL_DFT_INVERSE
mepl_axis a = MEPL_BY_ROW;                                       //or MEPL_BY_COL
float* H[rows][cols];
```

The only task left is to fill the matrix (`H`) or input vector, which is up to the user.

The inputs in the table above are very similar for all create functions needed for transforms, but the create functions for impulse responses and convolutions differ slightly. These differences are explained in the next sections:

- Section 4.1.1.1, “Finite impulse response (FIR)”
- Section 4.1.1.2, “Infinite Impulse Response (IIR)”
- Section 4.1.1.3, “Convolution”

4.1.1.1 Finite impulse response (FIR)

For the FIR, the `kernel` is initialized similarly to the input vector for transforms, but it serves a different purpose. In addition, `K` acts as the length of the kernel, `N` is the length of the imaginary plane of the input vector, and `D` is the stride of the input vector. Example declaration:

```
/* Kernel with M elements */
mepl_length K = M;
mepl_length N;
mepl_length D;
float kernel[K];
```

4.1.1.2 Infinite Impulse Response (IIR)

For the IIR, there are two input vectors (`A` and `B`). If the vectors have contiguous elements, the strides (`A_stride` and `B_stride`) will equal one. Otherwise, the stride will be the number of memory addresses in between each element. The length of both vectors is given by `num_filters` and the variable `N` is the length of input data vectors for computations. Example declaration:

```
/* Vectors with M contiguous elements */
mepl_length num_filters = M;
mepl_length N;
```

```
float A[num_filters];
float B[num_filters];
mepl_stride A_stride = 1;
mepl_stride B_stride = 1;
```

4.1.1.3 Convolution

For convolution, `P const* H` is the convolution kernel but will be declared and initialized the same as if it were a regular matrix. `H_length`, the length of the convolution kernel, and `X_length`, the length of the input, are identical to the length of a matrix or vector, so declaration and initialization are the same. The last input parameter needed to create an object for convolution is decimation. The decimation factor, which is the rate at which you wish to sample the data, is initialized just as a length would be initialized. Example declaration:

```
/* 1D Convolution of Vector with M contiguous elements */
mepl_length H_length = M;
float H[H_length];
mepl_length X_length = M;
mepl_length decimation;


/* 2D Convolution of an MxN Matrix */
mepl_length H_rows = M, X_rows = M;
mepl_length H_cols = N, X_cols = N;
float H[H_rows][H_cols];
mepl_length decimation;
```

As described earlier under naming conventions, when a function ends with `_zf`, it means the input is split-complex. In other words, there will be two input vectors, one containing the real part of a number and the other containing the imaginary part. When the input is split-complex, the pointer for an interleaved vector (`H*`) will become two pointers (`H_real*` and `H_imag*`). The same happens to the input variables pertaining to the dimensions. When declaring and initializing your input data, keep this in mind. Whether you choose to use interleaved vectors or split vectors makes a difference in what and how many input parameters are needed.

4.1.2 Summary of declaring and initializing create functions

By now, you should be able to declare and initialize input parameters and use the create functions. If you are still confused about the parameters, never fear! Here is a table summarizing what was just explained. While you are looking at a specific function in the reference manual and you don't know what to do with it, look at this table. First, find the input parameter you are having trouble with. Then, you can see what it stands for and how to declare and initialize the variable. For example declarations, the question mark means it is up to the user to set a value.

Table 3. Summary of create functions

When you see:	What it means:	Example declarations:
`mepl_length l`	Length of vector/matrix	`mepl_length l = ?;`
`mepl_direction d`	Desired direction of computation	`mepl_direction d = MEPL_DFT_FORWARD;` or `mepl_direction d = MEPL_DFT_INVERSE;`
`mepl_length rows`	Number of rows in matrix	`mepl_length rows = ?;`
`mepl_length cols`	Number of columns in matrix	`mepl_length cols = ?;`
`mepl_axis a`	Desired axis to approach for computation	`mepl_axis a = MEPL_BY_ROW;` or `mepl_axis a = MEPL_BY_COL;`
P `const* kernel`	Kernel for FIR	*P* `Kernel[K];`
`mepl_length K`	Length of input vector	`mepl_length K = ?;`
`mepl_length N`	Plane on which FIR is computed	`mepl_length N = ?;`
`mepl_length D`	Stride of input data	`mepl_length D = ?;`
`mepl_length num_filters`	Length of A & B vectors	`mepl_length num_filters = ?;`
P `const* A`	Input vector	*P* `A[num_filters];`
`mepl_stride A_stride`	Stride of vector A	`mepl_stride A_stride = ?;`
P `const* B`	Input vector	*P* `B[num_filters];`
`mepl_stride B_stride`	Stride of vector B	`mepl_stride B_stride = ?;`
`mepl_length N`	Length of input data vectors	`mepl_length N = ?;`
P `const* H`	Input vector for convolution	*P* `H[H_length];`
`mepl_length H_length`	Length of vector H	`mepl_length H_length = ?;`
`mepl_length X_length`	Length of input vector	`mepl_length X_length = ?;`
`mepl_length decimation`	Decimation factor for convolution	`mepl_length decimation = ?;`
P `const*H`	Input matrix for convolution	*P* `H[H_rows][H_cols];`
`mepl_length H_rows`	Number of rows in matrix H	`mepl_length H_rows = ?;`
`mepl_length H_cols`	Number of columns in matrix H	`mepl_length H_cols = ?;`
`mepl_length X_rows`	Number of rows in input matrix	`mepl_length X_rows = ?;`
`mepl length X_cols`	Number of columns in input matrix	`mepl_length X_cols = ?;`

4.1.3 How to use create functions

The vast majority of signal processing functions need create functions to supply a pointer to the object used to compute the signal. For each function, the data type for the object pointer is:

`mepl_`*`function_p`*`*`

At the beginning of each function in the Mentor Graphics reference manual, these object types are defined. The structure of this object does not need to be understood in order to carry out this function. For example, if an object was needed for a 1D DFT using interleaved complex floating point inputs, then the pointer would be:

```
mepl_dft_cf*
```

A pointer should be declared and then set equal to the function. After calling the create function, this pointer can be used further for the signal computation.

4.1.4 Example of create functions

All together, the code for declaring variables and calling the create function for an NxN matrix would go as follows:

```
/* Variable Declarations */
mepl_length l = N;
mepl_dft_direction d = MEPL_DFT_FORWARD;                    //or MEPL_DFT_INVERSE
mepl_length stride = N;
mepl_cfloat const* X[l];

/* Object Pointer for 1D DFT */
mepl_dft_cf* dft;

/* Creating the Object */
dft = mepl_dft_create_cf(l, d);
```

4.2 Destroy functions

Just as the name would suggest, these functions destroy the objects that are used in computing transforms, impulse responses, and convolution when they are no longer needed. The names for each of these functions are:

```
mepl_function_destroy_p
```

The only input parameter for destroy functions is the object pointer. The code to call the function would be:

```
mepl_dft_destroy_cf(dft);
```

4.3 Compute functions

The final function to solve transformations, impulse responses, and convolutions is the compute function. The compute function is the second to be performed out of the three. There are two different types of compute functions: in-place computation and out-of-place computation. In-place means the results are written over the input vector or matrix, i.e. in the place of the input values. Out-of-place means there is a separate vector or matrix for the output values, so there will be an extra parameter passed into the compute function. Compute functions have two different function calls, given below:

```
mepl_function_compute_ip_p (in-place computation)
```

or

```
mepl_function_compute_op_p (out-of-place computation)
```

The inputs for compute functions differ depending on the dimension and type of function. The table below gives the parameters passed to each compute function.

Table 4. Compute functions with inputs

Compute Functions	Inputs
1D Discrete Fourier Transform (DFT)	
in-place computation	`mepl_dft_cf const* dft, mepl_cfloat* X, mepl_stride stride`
out-of-place computation	`mepl_dft_cf const* dft, mepl_cfloat const* X, mepl_stride X_stride, mepl_cfloat* Y, mepl_stride Y_stride`
Multiple 1D DFT	
in-place computation	`mepl_dftm_cf const* dft, mepl_cfloat* X, mepl_stride stride`
out-of-place computation	`mepl_dftm_cf const* dft, mepl_cfloat const* X, mepl_stride X_stride, mepl_cfloat* Y, mepl_stride Y_stride`
2D DFT	
in-place computation	`mepl_dft2d_cf const* dft, mepl_cfloat* X, mepl_stride stride`
out-of-place computation	`mepl_dft2d_cf const* dft, mepl_cfloat const* X, mepl_stride X_stride, mepl_cfloat* Y, mepl_stride Y_stride`
Complex-to-Real DFT	
in-place computation	`Not applicable`
out-of-place computation	`mepl_dft_c2r_cf const* dft, mepl_cfloat const* X, mepl_stride X_stride, mepl_cfloat* Y, mepl_stride Y_stride`
Real-to-Complex DFT	
in-place computation	`Not applicable`
out-of-place computation	`mepl_dft_r2c_cf const* dft, mepl_cfloat const* X, mepl_stride X_stride, mepl_cfloat* Y, mepl_stride Y_stride`
Multiple 1D Complex-to-Real	
in-place computation	`Not applicable`
out-of-place computation	`mepl_dftm_cf const* dft, mepl_cfloat const* X, mepl_stride X_stride, mepl_cfloat* Y, mepl_stride Y_stride`

Table 4. Compute functions with inputs (continued)

Compute Functions	Inputs
Multiple 1D Real-to-Complex DFT	
in-place computation	`Not applicable`
out-of-place computation	`mepl_dftm_cf const* dft, mepl_cfloat const* X, mepl_stride X_stride, mepl_cfloat* Y, mepl_stride Y_stride`
2D Complex-to-Real	
in-place computation	`Not applicable`
out-of-place computation	`mepl_dft2d_cf const* dft, mepl_cfloat const* X, mepl_stride X_stride, mepl_cfloat* Y, mepl_stride Y_stride`
2D Real-to-Complex DFT	
in-place computation	`Not applicable`
out-of-place computation	`mepl_dft2d_cf const* dft, mepl_cfloat const* X, mepl_stride X_stride, mepl_cfloat* Y, mepl_stride Y_stride`
1D Discrete Cosine Transform - Type 1 (DCT1)	
in-place computation	`mepl_dct1_f const* dct, mepl_cfloat const* X, mepl_stride X_stride, mepl_cfloat* Y, mepl_stride Y_stride`
out-of-place computation	`mepl_dct1_f const* dct, float const *X, mepl_stride X_stride, float* Y, mepl_stride Y_stride`
1D Discrete Sine Transform - Type 1 (DST1)	
in-place computation	`mepl_dst1_cf const* dst, mepl_cfloat const* X, mepl_stride X_stride, mepl_cfloat* Y, mepl_stride Y_stride`
out-of-place computation	`mepl_dst1_f const *dst, float const* X, mepl_stride X_stride, float* Y, mepl_stride Y_stride`
Multiple 1D DCT1	
in-place computation	`mepl_dct1m_cf const* dct, mepl_cfloat const* X, mepl_stride X_stride, mepl_cfloat* Y, mepl_stride Y_stride`
out-of-place computation	`mepl_dct1m_f const* dct, float const* X, mepl_stride X_stride, float* Y, mepl_stride Y_stride`

Table 4. Compute functions with inputs (continued)

Compute Functions	Inputs
Multiple 1D DST1	
in-place computation	`mepl_dst1m_cf const* dct, mepl_cfloat const* X, mepl_stride X_stride, mepl_cfloat* Y, mepl_stride Y_stride`
out-of-place computation	`mepl_dst1m_f const* dst, float const* X, mepl_stride X_stride, float* Y, mepl_stride Y_stride`
2D DCT1	
in-place computation	`mepl_dct12d_cf const* dct, mepl_cfloat const* X, mepl_stride X_stride, mepl_cfloat* Y, mepl_stride Y_stride`
out-of-place computation	`mepl_dct12d_f const *dct, float const* X, mepl_stride X_stride, float* Y, mepl_stride Y_stride`
2D DST1	
in-place computation	`mepl_dst12d_cf const* dct, mepl_cfloat const* X, mepl_stride X_stride, mepl_cfloat* Y, mepl_stride Y_stride`
out-of-place computation	`mepl_dst12d_f const* dct, float const* X, mepl_stride X_stride, float* Y, mepl_stride Y_stride`
Finite Impulse Response (FIR)	`mepl_fir_state_p* fir, P const* A, mepl_stride A_stride, P* Z, mepl_stride Z_stride`
Infinite Impulse Response (IIR)	`mepl_iir_state_p* iir, P const* A, mepl_stride A_stride, P* Z, mepl_stride Z_stride`
Convolution	`mepl_conv_state_p const* conv, P const* X, mepl_stride X_stride, P* Z, mepl_stride Z_stride`
2D Convolution	`mepl_conv2d_min_state_p* conv, P const* X, mepl_stride X_stride, P* Z, mepl_stride Z_stride`

4.3.1 Declaring and initializing compute functions

For compute functions, the first parameter will be a pointer to the object generated with the create functions. The input parameters following are the input vector(s) or matrix and output vector(s) or matrix. The parameters with the data type of `mepl_cfloat` are interleaved complex floating points.

When initializing the vectors or matrices, use the appropriate macros given in the MEPL reference manual and given in the Section 2, “Input parameters for common data types.” Here’s an example initialization:

```
/* Initialization of Vector with k Elements */
mepl_length l = k;
```

```
mepl_cfloat X[l];


/* Fill Vector with Desired Values */
float real = 0.0;
float imag = 0.0;
int i;
for(i=0;i<1;i++){
    X[i] = mepl_cfloat(real,imag)          ;//calling macro
}
```

In addition, the initialized value could be hardcoded, which doesn't require using a macro. An example of this is given below.

```
/* Initialization of Vector with k Elements */
mepl_length l = k;
mepl_cfloat X[l];


/* Fill Vector with Value Init */
mepl_cfloat Init = {0.0,0.0};
int i;
for(i=0;i<1;i++){
    X[i] = Init;
}
```

Another option is to use split-complex vectors and matrices. These will be initialized as described under the create functions above with two separate variables.

4.3.2 Summary of compute functions

Below is a table to help recap the compute function. If you are looking back and forth between the reference manual and this guide, you can easily find the input value giving you trouble, discover its meaning, and see an example declaration. Anytime a question mark is seen, the value is to be set by the user.

Table 5. Summary of compute functions

When you see:	What it means:	Example declaration
`mepl_function_p*`	Object pointer	Output for create function
`mepl_cfloat* X`	Input vector for transform	`mepl_cfloat X[l] = mepl_cfloat(r, i);`
`mepl_length stride`	Amount of memory between each element	`mepl_length stride = 1;`

Table 5. Summary of compute functions (continued)

When you see:	What it means:	Example declaration
`P* X_real`	For a split-complex input, this is the vector containing the real part of each element	`P X_real[l];` `X_real[j] = ?;`
`P* X_imag`	For a split-complex input, this is the vector containing the imaginary part of each element	`P X_imag[l];` `X_imag[j] = ?;`
`mepl_length X_stride`	Amount of memory between each element; stride for X vector when using out-of-place computation	`mepl_length X_stride = 1;`
`mepl_cfloat* Y`	Output vector needed for out-of-place computation	`mepl_cfloat Y[l] = mepl_cfloat(r, i);`
`mepl_length Y_stride`	Amount of memory between each element; stride for Y vector when using out-of-place computation	`mepl_length Y_stride = 1;`
`P* Y_real`	When using out-of-place computation and split-complex output, this is the vector containing the real part of each element	`P Y_real[l];` `Y_real[j] = ?;`
`P* Y_imag`	When using out-of-place computation and split-complex output, this is the vector containing the imaginary part of each element	`P Y_imag[l];` `Y_imag[j] = ?;`

4.3.3 How to use compute functions

After all the necessary elements are initialized, the compute function simply needs to be called. For example, for a 1D DFT function, the function call would resemble this:

```
mepl_dft_compute_ip_cf(dft, X, stride);
```

or

```
mepl_dft_compute_op_cf(dft, X, X_stride, Y, Y_stride);
```

The results can then be reviewed by printing the appropriate vector and using the return macros supplied in the MEPL. They are also stated in the Section 2, “Input parameters for common data types.” Below is an example for printing an element of type `mepl_cfloat`.

```
/* Print Vector X with k Elements */
int i;
for(i=0;i<k;i++){
    printf(“%f + i %f”, mepl_real_cf(X[i]), mepl_imag_cf(X[i]));
}
```

Both macros used in the `printf` statement above return the desired part of element `X[i]`. For the in-place computation, the `X` vector would be printed and the `Y` vector would be printed for the out-of-place computation.

4.3.4 Example of compute functions

Now that create, destroy, and compute functions are understood, here is an example for a 1D DFT using all three:

```
/* Initialization for Vector with 4 Elements */
mepl_length l = 4;
mepl_dft_direction d = MEPL_DFT_FORWARD;
mepl_length stride = 1;
mepl_cfloat const* X[l]

/* Object Pointer for 1D DFT */
mepl_dft_cf* dft;

/* Creating the Object */
dft = mepl_dft_create_cf(l, d);

/* Compute the 1D DFT */
mepl_dft_compute_ip_cf(dft, X, stride);

/* Destroy the object */
mepl_dft_destroy_cf(dft);
```

For actual results, please read AN2115, *Complex Floating Point Fast Fourier Transform*, which gives the Fourier Transform of an 8x8 matrix.

That concludes the explanation of transformations, impulse responses, and convolutions. The remaining signal processing functions include windowing functions and other miscellaneous functions, which will be explained in that order respectively.

4.4 Windowing functions

The windowing functions are used to create a vector within a certain window. The inputs are given in the following table.

Table 6. Windowing functions with inputs

Windowing Functions	Inputs
Blackman window	`P *Z, mepl_stride Z_stride, mepl_length Z_length`
Dolph-Chebyshev window	`P ripple, P *Z, mepl_stride Z_stride, mepl_length Z_length`
Hanning window	`P *Z, mepl_stride Z_stride, mepl_length Z_length`
Kaiser window	`P beta, P *Z, mepl_stride Z_stride, mepl_length Z_length`

4.4.1 Declaring and initializing windowing functions

Each of the windowing functions has an output vector (Z), with dimensions given by the stride and length variables. For the Dolph-Chebyshev and Kaiser windows, there is an extra parameter for each that is passed in first. Both should be initialized as a scalar, given the values corresponding to the user's desired output. An example declaration is given below:

```
/* Initialization for Vector with k Elements */
mepl_stride Z_stride = 1;
mepl_length Z_length = k;
P Z[Z_length];
P ripple;                          //Dolph-Chebyshev window only
P beta;                            //Kaiser window only
```

4.4.2 Summary of windowing functions

A recap of the windowing functions is given below, which is useful when looking for a specific input parameter. In the example declaration column, the question marks means the set value is under the discretion of the user.

Table 7. Summary of windowing functions

When you see:	What it means:	Example Declaration
`mepl_length Z_length`	Desired length for output vector	`mepl_length Z_length = ?;`
`P *Z`	Pointer to output vector	`P Z[Z_length];` `Z[j] = ?;`
`mepl_length Z_stride`	Stride of output vector	`mepl_length Z_stride = 1;`

Table 7. Summary of windowing functions (continued)

When you see:	What it means:	Example Declaration
`P* ripple`	Desired ripple for the Dolph-Chebyshev window	`P ripple = ?;`
`P* beta`	Transition width parameter for Kaiser window	`P beta = ?;`

4.4.3 How to use windowing functions

Once the parameters are initialized, the function can then be called. For instance, to create a floating point vector initialized with a Dolph-Chebyshev window, the call would be:

```
mepl_chebyshev_f(ripple, Z, Z_stride, Z_length);
```

The output would then be available to print in the `Z` vector.

4.4.4 Example of windowing functions

All together, the code should resemble the following example for a Dolph-Chebyshev window with floating point data types. The values given are not set values and should be changed to match the user's task at hand.

```
/* Initialization for Vector with k Elements */
mepl_stride Z_stride = 1;
mepl_length Z_length = k;
float Z[Z_length];

/* Fill Input Vector */
int i;
for(i=0;i<Z_length;i++){
    Z[i] = i;
}

/* Ripple of size 4 for Window */
float ripple = 4.0;

/* Function Call */
mepl_chebyshev_f(ripple, Z, Z_stride, Z_length);
```

4.5 Miscellaneous functions

The rest of the signal processing functions are given below, with their corresponding input parameters. Primarily, they deal with the limits of a vector.

Table 8. Miscellaneous functions with inputs

Miscellaneous Functions	Inputs
Clip within outside	 `P low, P high, P* A, mepl_stride A_stride, P* Z, mepl_stride Z_stride, mepl_length length` `P low, P high, P* A, mepl_stride A_stride, P* Z, mepl_stride Z_stride, mepl_length length`
Limit	`P limit, P* A, mepl_stride A_stride, P* Z, mepl_stride Z_stride, mepl_length length`
Threshold	`P limit, P* A, mepl_stride A_stride, P* Z, mepl_stride Z_stride, mepl_length length`
Unwrap	`P const* A, mepl_stride A_stride, P* Z, mepl_stride Z_stride, mepl_length length`
Histogram	`P const* X, mepl_stride X_stride, mepl_length X_length, P min, P max, unsigned int* Y, mepl_stride Y_stride, mepl_length Y_length`

4.5.1 Declaring and initializing miscellaneous functions

The declarations and initializations for the inputs above are the same as described in previous sections. Therefore, this section will be excluded.

4.5.2 Summary of miscellaneous functions

Below is a summary table with example declarations for the miscellaneous signal processing functions.

Table 9. Summary of miscellaneous functions

When you see:	What it means:	Example Declaration
`P low`	Lower threshold that defines the area needed to be clipped	`P low = ?;`
`P high`	Upper threshold that defines the area needed to be clipped	`P high = ?;`
`P limit`	Single-sided clip	`P limit = ?;`
`mepl_length length`	Length of vectors A and Z	`mepl_length length = ?;`
`P const* A`	Input vector for clip, threshold, or unwrap	`P A[length];` `A[j] = ?;`
`mepl_stride A_stride`	Stride of input vector	`mepl_stride A_stride = 1;`
`P* Z`	Output vector for clip, threshold, or unwrap	`P Z[length];` `Z[j] = ?;`
`mepl_stride Z_stride`	Stride of output vector	`mepl_stride Z_stride = 1;`
`mepl_length X_length`	Length of input vector	`mepl_length X_length = ?;`

Table 9. Summary of miscellaneous functions (continued)

When you see:	What it means:	Example Declaration
`P const* X`	Input vector for histogram	`P X[X_length];` `X[j] = ?;`
`mepl_stride X_stride`	Stride of input vector	`mepl_stride X_stride = ?;`
`P min`	Edge of first histogram bin	`P min = ?;`
`P max`	Edge of last histogram bin	`P max = ?;`
`mepl_length Y_length`	Length of output vector for histogram	`mepl_length Y_length = ?`
`unsigned int* Y`	Output vector for histogram	`unsigned int Y[Y_length];` `Y[j] = ?;`
`mepl_stride Y_stride`	Stride of output vector	`mepl_stride Y_stride = ?;`

4.5.3 How to use miscellaneous functions

These miscellaneous functions are fairly simple to execute. It is just a matter of declaring the appropriate variables and passing them through the function.

4.5.4 Example of miscellaneous functions

An example of the function that clips values to lie outside the given values is given below. The primary data type used here will be floating points.

```
/* Initialization for Vector with k Elements */
mepl_stride A_stride = 1;
mepl_stride Z_stride = 1;
mepl_length length = k;
float A[length];
float Z[length];

/* Fill the Input Vector */
int i;
for(i=0;i<length;i++){
    A[i] = i;
}
/* Lower and Upper Thresholds */
float low = 6;
float high = 24;

/* Function Call */
mepl_iclip_f(low, high, A, A_stride, Z, Z_stride, length);
```

5 Examples

From this point on, there will be examples of code using functions from the MEPL. The functions used are given in this order:

1. Scalar multiply
2. Multiply-Add
3. Maximum, returned by index
4. Root mean square
5. 1D discrete fourier transform on 2D matrix
6. Fill vector with random values.
7. Linear interpolation

The order also correlates to the order they appear in the MEPL Reference Manual.

5.1 Scalar multiply

Using complex floating points, the function name for scalar multiply is:

```
mepl_vmul_s_cf
```

Example Code:

```
#include <stdio.h>
#include <math.h>
#include <mepl.h>

void main(void){

    /* Initialization for 8 Element Vector */
    mepl_stride stride = 1;
    mepl_length length = 8;
    mepl_cfloat A[length];
    mepl_cfloat b = 2.5;             //scalar to multiply by

    /* Fill Input Vector using Macro */
    float real = 1.0;
    float imag = 0.0;
    int i;
    for(i+0;i<length;i+=1){
        A[i] = mepl_cfloat(real,imag);
    }

    /* Result Vector */
    mepl_cfloat Z[length];
```

```
    /* Function Call */
    mepl_vmul_s_cf(A, stride, b, Z, stride, length);

    /* Print Result from Vector Z */
    int i;
    for(i=0;i<length;i++){
    printf("%f + i %f", mepl_real_cf(Z[i]), mepl_imag_cf(Z[i]));
    }
}
```

5.2 Multiply-Add

Using floating points, the function name for multiply-add is:

```
mepl_vmadd_f
```

Example Code:

```
#include <stdio.h>
#include <math.h>
#include <mepl.h>

void main(void){

    /* Initialization for 8 Element Vector */
    mepl_stride stride = 1;
    mepl_length length = 8;
    float A[N];
    float B[N];
    float C[N];

    /* Fill Input Vectors with Init Value */
    float Init = 1.0;
    int i;
    for(i=0;i<length;i+=1){
        A[i] = Init;
        B[i] = Init*2;
        C[i] = Init;
        Init += 1;
    }
    /* Result Vector */
    float Z[length];

    mepl_vmadd_f(A, stride, B, stride, C, stride, Z, stride, length);
}
```

When adding a scalar instead of a vector, the function name is:

```
mepl_vaxpsy_f
```

Example Code:

```
#include <stdio.h>
#include <math.h>
#include <mepl.h>

void main(void){

    /* Initialization for 8 Element Vectors */
    mepl_stride stride = 1;
    mepl_length length = 8;
    float A[length];
    float B[length];
    float c = 2.0;                    //scalar to add with

    /* Fill Input Vectors with Value Init */
    float Init = 1.0;
    int i;
    for(i=0;i<length;i+=1){
        A[i] = B[i] = Init;
        Init += 1.0;
    }

    /* Result Vector*/
    float Z[length];

    /* Function Call */
    mepl_vaxpsy_f(A, stride, B, stride, c, Z, stride, length);
}
```

5.3 Maximum, returned by index

Using floating points, the function name for maximum, returned by index is:

```
mepl_maxidx_f
```

In this example, malloc and free are used for the index variable. Without malloc and free, the code will not compile because of an incompatible declaration error. Because malloc and free are used, the file `stdlib.h` needs to be linked as well.

Example Code:

```
#include <stdio.h>
#include <math.h>
#include <mepl.h>
#include <stdlib.h>

void main(void){

    /* Initialization for 8 Element Vector */
    mepl_stride stride = 1;
    mepl_length length = 8;
    float A[length];
    /* Fill Input Vector with Value Init */

    float Init = 1.0;
    int i;
    for(i=0;i<length;i+=1){
        A[i] = Init;
        Init += 1;
    }

    /* Result Variable */
    float result;

    /* Pointer to Index */
    mepl_index *idx = (mepl_index*) malloc(1);

    /* Function Call */
    result = mepl_maxidx_f(A, stride, length, idx);

    /* Free Index Pointer */
    free(idx);
}
```

5.4 Root mean square

Using floating points, the function name for root mean square is:

```
mepl_rms_f
```

This particular function doesn't have the optimization expected with AltiVec technology. If simpler functions are used to compute the root mean square, then optimization can be seen. The functions used were a summation with absolute value and square-root. Together, the root mean square can be computed.

Example Code:

```
#include <stdio.h>
#include <math.h>
#include <mepl.h>


void main(void){
    /* Initialization for 8 element Vector */
    mepl_stride stride = 1;
    mepl_length length = 8;
    float X[length];
    float sum;                  //extra variable used for
    sum = 0.0;                  //summation
    /* Fill Input Vector with Value Init */
    float Init = 1.0;
    int i;
    for(i=0;i<length;i+=1){
        X[i] = Init;
        Init += 1;
    }
    /* Result Vector */
    float Z;

    /* Call to Summation-Absolute Value Function */
    sum = mepl_sumsq_f(X, stride, length);

    /* Divide Summation by Vector Length */
    sum /= length;

    /* Call to Square Root Function */
    mepl_vsqrt_f(&sum, stride, &Z, stride, stride);
}
```

5.5 1D discrete fourier transform on 2D matrix

Using floating point and split-complex values, the function name for 1D discrete fourier transform on 2D matrix is:

```
mepl_dft_compute_ip_zf
```

The example given below uses the 1D DFT to get a result for a 2D Matrix. Therefore, the 1D DFT function is computed in a loop on each column. The easier option would be to use the 2D DFT function.

Example Code:

```
#include <stdio.h>
#include <math.h>
#include <mepl.h>

/* To prevent errors, the arrays need to be global variables */
#define length 8              //number of rows and columns
#define log2_length 3
mepl_cfloat X[length][length];

void main(void){
    /* Fill Input Matrix with Values Init1 & Init2 */
    int i, j, k;
    mepl_cfloat Init1 = {0.0,0.0};
    mepl_cfloat Init2 = {1.0,0.0};

    for(i=0;i<length;i+=1){
        for(k=0;k<length;k+=1){
            //the if statement below creates a pulse
            if((i>=2)&&(i<=5)&&(k>=2)&&(k<=5)){
                X[i][k] = Init2;
            }else{  X[i][k] = Init1; }
        }
    }

    /* Create DFT Object */
    mepl_dft_cf* plan;
    plan = mepl_dft_create_cf(length,MEPL_DFT_FORWARD);

    /* Compute the 1D DFT on 2D Matrix */
    for(i=0;i<length;i++){
        mepl_dft_compute_ip_cf(plan, X[i], length);
```

```
    }

    /* Print Results */
    for(i=0;i<length;i++){
        printf("%f + i %f", mepl_real_cf(X[i]), mepl_imag_cf(X[i]));
    }
    /* Destroy DFT Object */
    mepl_dft_destroy_cf(plan);
}
```

5.6 Fill vector with random values

Using floating points, the function name for fill vector with random values is:

```
mepl_vrandu_f
```

Example Code:

```
#include <stdio.h>
#include <math.h>
#include <mepl.h>

void main(void){
    /* Initialization for 8 Element Vector */
    mepl_stride stride = 1;
    mepl_length length = 8;
    mepl_index seed = 16;
    /* Result Vector */
    float a[length];
    /* Create RNG Object using the Seed */
    mepl_randstate_f* r;
    r = mepl_randcreate_f(seed);
    /* Function Call */
    mepl_vrandu_f(r, a, stride, N);
    /* Destroy Object */
    mepl_randdestory_f(r);
}
```

5.7 Linear interpolation

Using floating points, the function name for linear interpolation is:

```
mepl_interp_linear_f
```

Example Code:

```
#include <stdio.h>
#include <math.h>
#include <mepl.h>

void main(void){
    /* Initialization for 8 Element Vectors */
    mepl_stride stride = 1;
    mepl_length length = 8;
    float x_data[length];
    float y_data[length];
    float xp[length];

    /* Fill Input Vectors with Values Init1 & Init2 */
    float Init1 = 3.1;
    float Init2 = 3.0;
    int i;
    for(i=0;i<length;i+=1){
        x_data[i] = y_data[i] = Init1;
        xp[i] = Init2;
        if(i%2!=0){ Init1 += 2.2; }
        else{ Init1 += 1.0; }
        Init2 += 1.0;
    }
    /* Result Vector */
    float yp[length];
    mepl_interp_linear_f(x_data, stride, y_data, stride, length, xp, stride, yp, stride,
    length);
}
```

6 MEPL and AltiVec technology

After reading this guide, hopefully you will be more comfortable using the Mentor Embedded Performance Library. This library takes advantage of the AltiVec technology on products with an e6500 core, such as the T4240 chip, accelerating the computation time needed to execute the functions. A few of the functions have been tested against other libraries, proving the speed-up time. The proof can be seen in the data provided for some of the example functions.

Table 10. Performance improvements with MEPL and AltiVec technology

Function	Input Size	Times Faster (approximate)	Percent Increase in Computation Speed
Scalar Multiply	Vector w/8 elements	0.2	– 80%
	Vector w/256 elements	1.0	– 4%
	Vector w/8192 elements	5.0	402%
Vector Multiply-Vector Add	Vector w/8 elements	0.1	– 87%
	Vector w/256 elements	1.4	42%
	Vector w/8192 elements	3.3	227%
Vector Multiply-Scalar Add	Vector w/8 elements	0.1	– 89%
	Vector w/256 elements	0.7	– 32%
	Vector w/8192 elements	3.9	294%
Maximum, returned by index	Vector w/8 elements	0.1	– 93%
	Vector w/256 elements	0.4	– 93%
	Vector w/8192 elements	3.3	227%
Root Mean Square (RMS)	Vector w/8 elements	7.4	638%
	Vector w/256 elements	6.1	507%
	Vector w/8192 elements	8.0	696%
1D Discrete Fourier Transform on 2D Matrix	8x8 Matrix	0.5	– 48%
	256x256 Matrix	3.6	263%
Fill Vector with Random Values	Vector w/8 elements	1.9	93%
	Vector w/256 elements	5.5	453%
	Vector w/8192 elements	6.5	549%

For each function and input size, there is an approximate result describing how much faster the function from the MEPL is compared to an alternative function or method. This was calculated by dividing the number of clock cycles it took to complete the alternative computation by the number of clock cycles it took to complete the MEPL function. The last column is then the percent increase in speed that AltiVec technology creates for the MEPL functions. As seen in the table above, AltiVec technology optimization tends to be greater with large sizes of data. Overall, the MEPL is extremely useful for anyone who wishes to make faster calculations.

7 Revision history

This table provides a revision history for this document.

Table 11. Document revision history

Rev. number	Date	Substantive change(s)
0	08/2013	Initial release.

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: AN4786
Rev. 0
08/2013