Example Library API Reference

Version 3.0

Printed in the United States of America March 2007

IBM Systems and Technology Group
2070 Route 52, Bldg. 330
Hopewell Junction, NY 12533-6351

The IBM home page can be found at **ibm.com**

The IBM semiconductor solutions home page can be found at **ibm.com**/chips

September 14, 2007

Preface

The document provides Application Programming Interface (API) descriptions of the example libraries included in the Software Development Kit (SDK) for Multicore Acceleration. These examples have been provided to assist in the education of Cell Broadband Engine (CBE) programming through code examples. These code examples have been provided in the form of libraries to assist in the development of CBE applications.

See the *Revision Log* on page 199 for a list of changes to this document.

Who Should Read This Manual

This document is intended for use by software engineers that are developing applications for use with the Cell Broadband Engine (CBE).

1. Contents

2. List of Figures

3. Overview

This document contains user documentation for the example libraries prototype SDK samples libraries. This document has been organized into the following sections, each section corresponding to one of the functional example libraries.

Document Section	Description
Section 4 SPU Software Managed Cache Library on page 13	Describes the services provided in the example *SPU Software Managed Cache Library*
Section 5 FFT Library on page 27	Describes the subroutines in the example *FFT Library*.
Section 6 Game Math Library on page 33	Describes the subroutines in the example *Game Math Library*.
Section 7 Image Library on page 47	Describes the subroutines in the example *Image Library*.
Section 8 Large Matrix Library on page 55	Describes the subroutines in the example *Large Matrix Library*.
Section 9 Matrix Library on page 79	Describes the subroutines in the example small *Matrix Library*.
Section 10 Misc Library on page 97	Describes the subroutines in the example *Misc Library*.
Section 11 Multi-Precision Math Library on page 121	Describes the subroutines in the example *Multi-Precision Math Library*
Section 12 Sync Library on page 147	Describes the subroutines in the example *Sync Library*.
Section 13 Vector Library on page 173	Describes the subroutines in the example short *Vector Library*.
Revision Log on page 199	Provides a listing of the changes for each version of this document.

These example libraries have been provided to assist software developers with Cell Broadband Engine programming by:

- Providing reusable and optimized subroutines specifically targeted for the processor / architecture.
- Providing the foundation for the development of CBE applications.
- Providing libraries/subroutines that abstract HW features and functions.
- Providing libraries that address the primary target processor applications.
- Providing both vectored and scalar subroutines. Vectored PPE-targeted routines exploit the Vector/SIMD multimedia extension.
- Providing functions in both callable and inlineable library subroutines.
- Providing readable and well documented source code that can be easily customized and tailored to the end users needs and/or data formats.

Note: The example libraries provide no special handling of erroneous inputs or conditions (e.g. divide by zero; out of supported range inputs).

4. SPU Software Managed Cache Library

To facilitate applications with non-predicatable data access patterns, a set of utilities are provided for the construction and access of a software managed cache in the SPU's local store.

Name(s)

none

Header File(s)

<cache-api.h>
<cache-dir.h>
<cache-4way.h>
<cache-stats.h>

4.1 Overview

A software managed cache can be constructed by defining a couple of required attributes and including the cache header file. Several other attributes may optionally be defined which determine the cache topology and behavior. Multiple caches may be defined in the same program by re-defining these attributes and re-including the header file (cache-api.h). The only restriction is that the CACHE_NAME must be different for each cache.

Table 4-1. Cache Attributes

Symbol	Description	Possible Values	Default Value
CACHED_TYPE	Specifies the type of data being cached.	any valid data type	none (required)
CACHE_NAME	Specifies the unquie string associated with the cache.	any	none (required)
CACHELINE_LOG2SIZE	Specifies the log base 2 of cache line size in bytes.	4-12	7 (128 bytes)
CACHE_LOG2NSETS	Specifies the log base 2 of the number of sets in the cache.	0-12	6 (64 sets)
CACHE_LOG2NWAY	Specifies the associativity of the cache.	0 or 2	2 (4-way)
CACHE_TYPE	Specifies the type of cache. 0 specifies that the cache is a read only cache, 1 specifies that the cache is read/write.	0 or 1	1 (read/write)
CACHE_SET_TAGID(set)	Specifies the tag ID for the given set.	any (0 thu 31)	set & 0x1F
CACHE_READ_X4	When the cached type is an integral type that fits in a 32-bit word, this define enables the cache_rd_x4() service which caches and returns four data items at a time in a vec_uint4.	-	not defined
CACHE_STATS	When this is defined, the cache code maintains metrics on cache activity. These can be displayed by calling the cache_pr_stats() service from within the program using the cache.	-	not defined

Note: Only CACHED_TYPE and CACHE_NAME are required to be defined by the programmer. It is recommended to use a previously typedef-ed type for CACHED_TYPE to avoid any potential operator precedence issues that might arise. CACHE_NAME can be any string that would be suitable for a C function name, and is the same string which must be used to reference the cache using the supported interaces.

The interfaces in the SPE cache layer are implemented as macros or inline C functions. The programmer must define the cached data type, and include the SPE cache header files in order to call these interfaces, as an archived library interface is not currently supported. Multiple caches on local memory may be defined by re-including the cache header files. The external interfaces take the name of the cache (as defined by CACHE_NAME) as the first argument, which is prepended to the template function to form the name of actual function to be called.

There are two sets of cache services provided - safe and unsafe. The safe interfaces provide the programmer with an easy to use set of interfaces that can be used as a first pass implementation for code being ported to the CBE. For workloads with a high cache hit rate, the synchronous nature of *cache_rd* and *cache_wr* should not pose a big problem, since in most cases they will not block. Using *cache_wr* to store to the cache instead of the using LSA pointers returned by the unsafe interfaces has the advantage of not having to worry about locking data in the cache. The disadvantage of using *cache_wr* is that it does an address translation (EA -> LSA), which in the case of the unsafe model, is extraneous since the LSA pointer was already translated on the read.

The unsafe services provide a more efficient means of accessing the LS when compared to the safe services, at the expense of a little more programming complexity. Once an LSA pointer is acquired by a "read with intent to write" service (i.e cache_rw()or cache_touch()), any subsequent cache accesses must be preceded by a lock operation on that pointer. This ensures those cache accesses will not cause the referenced data to be cast out while the reference is held. For an N-way cache, up to N-1 locks may be simultaneously held. If more than N data references are needed, the locked data should be written and unlocked before proceeding.

The asynchronous services (touch/wait) provide the potential benefit of reducing the perceived wait time for memory references. They provide more benefit in the case where the cache hit rate is low, since they allow computation to occur in parallel with memory accesses, and if used optimally, can achieve little to no wait time for those accesses.

Since the size of a cache line is a power of two, the size of the cached data type should also be a power of two (or aligned up to the next power of two) and no bigger than the cache line size. The current implementation does not support data objects that span cache lines, so a cache line must be able to wholly contain N objects.

Depending on the memory access patterns of the application, it may be desirable to define multiple caches to suit each of the classes of data in use. For example, if some of the data is read-only, and some of it is read-write, it might be advantageous to cache those portions of the data in separate caches appropriately.

One big advantage of the software cache over a hardware cache is that one can easily change the topology and algorithms to optimize its effectiveness for any given workload. Extensive research has shown the potential benefits of and methodology for choosing a cache that's optimal for a given workload. Testing and analysis can be done iteratively to find what works best for each application.

4.1.1 Example

Define a cache:

```
#define CACHE_NAME          my_items
#define CACHED_TYPE         item_t
```

```
#define CACHE_TYPE              1/* r/w */
#define CACHELINE_LOG2SIZE      7/* 128 bytes */
#define CACHE_LOG2NWAY          2/* 4-way */
#define CACHE_LOG2NSETS         4/* 16 sets */
#include <cache-api.h>
```

Using the cache to swap two values (safe interfaces):

```
a = cache_rd(my_items, eaddr_a);
b = cache_rd(my_items, eaddr_b);
cache_wr(my_items, eaddr_b, a);
cache_wr(my_items, eaddr_a, b);
```

Using the cache to swap two values (unsafe interfaces):

```
a_ptr = cache_rw(my_items, eaddr_a)
cache_lock(a_ptr);
b_ptr = cache_rw(my_items, eaddr_b)
cache_unlock(a_ptr);

/* both items in cache, can now safely be modified through ptr */

tmp = *a_ptr;
*a_ptr = *b_ptr;
*b_ptr = tmp;
```

4.1.2 Tag IDs

By default, the software managed cache use a tag ID corresponding to the 5 least significant bits of the cache set being accessed. This use of tag ID's is both greedy and non-cooperative, and doesn't not take into account other application uses of tag IDs.

To be more cooperative, only a subset of the tag IDs should be reserved by the tag manager and used. It is the responsibility of the application code to reverse the tag IDs and override the cache to use the reserved tags.

For example, to reserve and use 8 tags for use by the cache code, application code must:

Override the CACHE_SET_TAGID macro so that the tag ID is computed from a base tag ID and the 3 lsb's of the cache set:

```
#define CACHE_SET_TAGID(set)        (tag_base + (set & 7))
```

Include the cache api header file:

```
#include <cache-api.h>
```

Prior to accessing the cache, reserve the 8 tags and set the tag base variable:

```
#include <spu_mfcio.h>
. . .
unsigned int tag_base;
if ((tag_base = mfc_multi_tag_reserve(8)) == MFC_TAG_INVALID) exit(1);
```

4.2 External (Safe) Interfaces

The external safe interfaces fully encapsulate the local store (LS) and require no knowledge of LS addresses or cache state. They can be called safely at any time.

4.2.1 cache_rd

C Specification

```
#include <cache-api.h>
CACHED_TYPE cache_rd(name, unsigned eaddr)
```

Descriptions

The *cache_rd* service reads from the cache, specified by *name,* the data from the specified 32-bit effective address. If the data is not present, the service blocks until it has been read into the cache.

See Also

cache_wr on page 17

4.2.2 cache_wr

C Specification

```
#include <cache-api.h>
void cache_wr(name, unsigned eaddr, CACHED_TYPE val)
```

Descriptions

The *cache_wr* service writes the specified value, *val*, to the cache specified by *name*. If the data is not present the service blocks until the data has been read into the cache and modified.

See Also

cache_rd on page 16
cache_flush on page 18

4.2.3 cache_flush

C Specification

```
#include <cache-api.h>
void cache_flush(name)
```

Descriptions

The *cache_flush* service writes all modified (dirty) cache lines back to main memory, for the cache specified by *name*. This should be call at the end of any program that uses a read/write cache style.

See Also

cache_wr on page 17
cache_flush on page 18

4.2.4 cache_pr_stats

C Specification

```
#include <cache-api.h>
void cache_pr_stats(name)
```

Descriptions

When CACHE_STATS is defined, the software cache maintains internal statistics on the cache activity. The *cache_pr_stats* service displays the statistics.

See Also

4.3 Internal (Unsafe) Interfaces

The following internal interfaces return pointers to local store addresses (LSAs) and require that the caller follow the prescribed conventions for correctness. Not following the conventions can result in inconsistencies in the cache.

4.3.1 cache_rw

C Specification

```
#include <cache-api.h>
CACHED_TYPE *cache_rw(name, unsigned eaddr)
```

Descriptions

The *cache_rw* service returns a LSA , within the cache specified by *name*, which holds the data that was cached from the specified effective address, *eaddr*. If the data is not currently in the cache (a miss), the call blocks until it has been read into the cache. The returned pointer can be used to directly read and write the data in the cache. The cache line containing the data is marked dirty at the time it is read (if it's a read/write cache), since it is expected that the data will be modified directly through the pointer. Storing to a cache that is defined as read-only is an error.

See Also

cache_rd on page 16
cache_wr on page 17
cache_flush on page 18
cache_rw on page 20
cache_touch on page 21

4.3.2 cache_touch

C Specification

```
#include <cache-api.h>
CACHED_TYPE *cache_touch(name, unsigned eaddr)
```

Descriptions

The *cache_touch* service is an asynchronous version of the *cache_rw* service. It returns a LS pointer, but does not block (whether or not the data is present in the cache). The returned pointer cannot be used until a subsequent *cache_wait* is performed, to ensure the data is present. If the cache is read/write, the containing cache line is also marked dirty at the time of the touch.

See Also

cache_rw on page 20
cache_wait on page 22

4.3.3 cache_wait

C Specification

```
#include <cache-api.h>
void cache_wait(name, unsigned lsa)
```

Descriptions

The *cache_wait* service blocks on a previously initiated (but unfinished) DMA request (a cache touch). It waits for the tag group id associated with the specified local store address, *lsa*. If there is no outstanding DMA for the associated tag group id, the service returns immediately.

See Also

cache_touch on page 21

4.3.4 cache_lock

C Specification

```
#include <cache-api.h>
void cache_lock(name, unsigned lsa)
```

Descriptions

The *cache_lock* service locks the cache line associated with the given local store address for the specified cache. This service should be used whenever a cache pointer needs to be used across multiple loads or stores to the cache (to guarantee the referenced data is not cast out). Up to N-1 locks can be held at a time, where N is the associativity of the cache.

See Also

cache_unlock on page 24

4.3.5 cache_unlock

C Specification

```
#include <cache-api.h>
void cache_unlock(name, unsigned lsa)
```

Descriptions

The *cache_unlock* service unlocks the cache line associated with the given local store address, *lsa,* for the cache speficied by *name*.

See Also

cache_lock on page 23

4.4 Specialized Interfaces

4.4.1 cache_rd_x4

C Specification

```
#include <cache-api.h>
vec_uint4 cache_rd_x4(name, vec_uint4 eaddr4)
```

Descriptions

The *cache_rd_x4* service reads four unsigned integers into the cache specified by *name* and returns their values in a SIMD vector. Upon return, all four data items are guaranteed to be in the cache (assuming 4-way or greater cache associativity)

See Also

cache_rd on page 16

5. FFT Library

The FFT (Fast Fourier Transform) library supports both 1-D FFTs as well as a base kernel functions that can be used to efficiently implement 2-D FFTs.

This library is supported on both the PPE and SPE. However, the 1D FFT functions are provided on the SPE only.

Name(s)

libfft_example.a

Header File(s)

<libfft_example.h>

5.1 fft_1d_r2

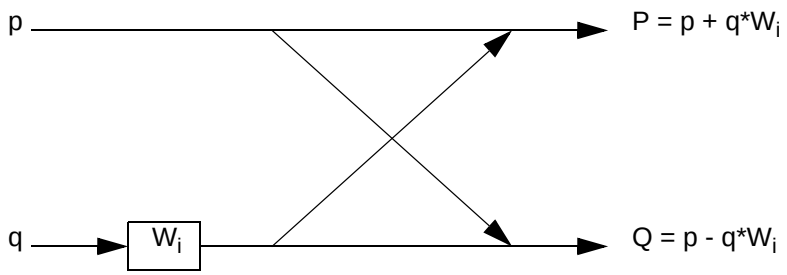
C Specification

```
#include <fft_1d_r2.h>
inline void _fft_1d_r2(vector float *out, vector float *in, vector float *W, int log2_size)

#include <libfft_example.h>
void fft_1d_r2(vector float *out, vector float *in, vector float *W, int log2_size)
```

Descriptions

The *fft_1d_r2* subroutine performs a single precision, complex, Fast Fourier Transform using the DFT (Discrete Fourier Transform) with radix-2 decimation in time. The input data, *in*, is an array of complex numbers of length 2^{log2_size} entries. The result is returned in the array of complex number specified by the *out* parameter. This routine supports an in-place transformation by specifying *in* and *out* to be the same array.

The implementation uses the Cooley-Tukey algorithm consisting of *log2_size* butterfly stages. The basic butterfly stage is:



where p, q, W_i, P, and Q are complex numbers.

This routine requires the caller to provide pre-computed twiddle factors, *W*. *W* is an array of single-precision complex numbers of length $2^{(log2\_size-2)}$ entries and is computes are follows for forward (time domain to frequency domain):

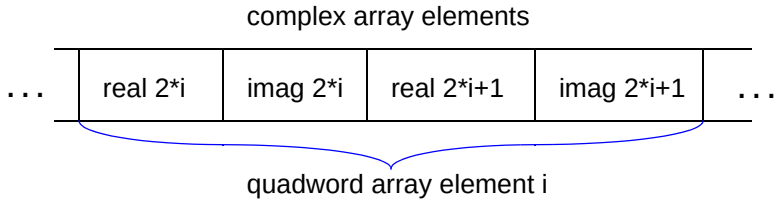
```
n = 1 << log2_size;
for (i=0; i<n/4; i++) {
      W[i].real = cos(i * 2*M_PI/n);
      W[i].imag = -sin(i * 2*M_PI/n);
}
```

Due to symmetry of the twiddle factors, the values can be more efficiently (reduced trig functions) computed as:

```
n = 1 << log2_size;
for (i=0; i<n/4; i++) {
      W[i].real = cos(i * 2*M_PI/n);
      W[n/4 - i].imag = -W[i].real;
```

```
}
```

The arrays of complex numbers are stored as quadwords with real and imaginary components interleaved.



This routine can also be used to perform a inverse (frequency domain to time domain) DFT by scaling the result by 1/log2_size and performing an in-place swap as follows:

```
vector unsigned int mask = (vector unsigned int){-1, -1, 0, 0};
vector float *start, *end, s0, s1, e0, e1;

n = 1 << log2_size;
fft_1d_r2(out, in, W, log2_size);
scale = spu_splats(1.0f/n);
s0 = e1 = *start;
for (i=0; i<n/4; i++) {
      s1 = *(start+1);
      e0 = *(--end);
      *start++ = spu_mul(spu_sel(e0, e1, mask), scale);
      *end = spu_mul(spu_sel(s0, s1, mask), scale);
      s0 = s1;
      e1 = e0;
}
```

Dependencies

See Also

fft_2d on page 30

5.2 fft_2d

C Specification

```
#include <fft_2d.h>
inline void _fft_2d(vector float *inreal, vector float *inimag, vector float *outreal,
                                                    vector float *outimag, int forward)

#include <libfft_example.h>
void fft_2d(vector float *inreal, vector float *inimag, vector float *outreal,
                                                    vector float *outimag, int forward)
```

Descriptions

The *fft_2d* subroutine transforms 4 rows of complex 2-D data from the time domain to the frequency domain (or vice versa). The direction of the transformation is specified by the *forware* parameter. If *forware* is non-zero, then *fft* converts the data from the time domain to the frequency domain. If *forward* is zero, then *fft* converts the data from the frequency domain to the time domain.

The complex input data is specified by the array pointers *inreal* and *inimag* corresponding the 4 rows of real and imaginary input data. The 4 rows are transformed and written to the output arrays as specified by the *outreal* and *outimag* parameters. The size of the rows was specified by the *init_fft_2d log2_samplesize* parameter.

The input data is row ordered and the output data is row interleaved. So, if R*n*E*m* means the *m*th element of the *n*th row, then the input looks like R1E1 R1E2 R1E3 ... R1E*n* R2E0 R2E1 ... R2E*n* R3E0 R3E1 ... R3E*n* R4E0 R4E1 ... R4E*n* (for a row length of *n*). The organization of the output looks like R1E1 R2E1 R3E1 R4E1 R1E2 R2E2 R3E2 R4E2 R1E3 R2E3 R3E3 R4E3 ... R1E*n* R2E*n* R3E*n* R4E*n*. This allows for more optimal processing of 2-D data since a 2-D FFT entails a 1-D FFT of the rows followed by a 1-D FFT of the columns.

The input and output arrays must be unique. That is, a FFT can not be performed in place.

Example Usage

Let's say that you have a 1024 by 1024 image that needs to be converted from the time domain to the frequency domain, and then you have to do some processing in the frequency domain, followed by a conversion back to the time domain, and let's further stipulate that you want the processing to be done inline rather than through subroutine calls, for improved performance.

The *fft* subroutine is called 256 times to process all the rows of the matrix (each time loading the results in the correct location of the output array, which now is time-domain and half frequency-domain). We then process this output array, doing FFTs on the columns (which now conveniently look like rows) and then loading the results back into the original array, which is now completely in the frequency domain.

After processing the data in the frequency domain, we simply reverse the process by executing the same code, but changing the value in the *forward* flag.

Example pseudocode follows:

```
#include <fft_2d.h>
vector float Ar[256*1024], Ai[256*1024], Br[256*1024], Bi[256*1024];
```

```
vector float Wr[1024], Wi[1024];
// Initialize the fft system to process the 1024x1024 image
// log2(1024) = 10
_init_fft_2d(10);

// Here you load Ar and Ai with your time domain data (real and imaginary)

// Convert the data from the time domain to the frequency domain.
for (i=0; i<256; i++) {
      fft_2d(&Ar[1024*i], &Ai[1024*i], Wr, Wi, 1);
      for (j=0; j<1024; j++) {
            Br[i+256*j] = Wr[j];
            Bi[i+256*j] = Wi[j];
      }
}
for (i=0; i<256; i++) {
      fft_2d(&Br[1024*i], &Bi[1024*i], Wr, Wi, 1);
      for (j=0; j<1024; j++) {
            Ar[i+256*j] = Wr[j];
            Ai[i+256*j] = Wi[j];
      }
}

// Now, Ar and Ai contain your data in the frequency domain.
// Do some processing in this domain.

// Convert the data back from the frequency domain to the time domain.
for (i=0; i<256; i++) {
      fft_2d(&Ar[1024*i], &Ai[1024*i], Wr, Wi, 0);
      for (j=0; j<1024; j++) {
            Br[i+256*j] = Wr[j];
            Bi[i+256*j] = Wi[j];
      }
}
for (i=0; i<256; i++) {
      fft_2d(&Br[1024*i], &Bi[1024*i], Wr, Wi, 0);
      for (j=0; j<1024; j++) {
            Ar[i+256*j] = Wr[j];
            Ai[i+256*j] = Wi[j];
      }
}
```

Dependencies

transpose_matrix4x4 on page 95

See Also

init_fft_2d on page 32
fft_1d_r2 on page 28

5.3 init_fft_2d

C Specification

```
#include <fft_2d.h>
inline void _init_fft_2d(int log2_samplesize)

#include <libfft_example.h>
void init_fft_2d(int log2_samplesize)
```

Descriptions

The *init_fft_2d* subroutine initializes the FFT library by precomputing several data arrays that are used by the *fft_2d* subroutine. The FFT data arrays are initialized according to the number of samples along each access of the 2-D data to be transformed. The number of samples is specified by the *log2_samplesize* parameter and must be in the range 5 to 11, corresponding to supported 2-D data arrays sizes of 32x32 up to 2048x2048.

The results are undefined for *log2_samplesize*'s less than 5 or greater than 11.

Dependencies

cosf4 in SIMD Math library
sinf4 in SIMD Math library

See Also

fft_2d on page 30

6. Game Math Library

The game math library consists of a set of routines applicable to game needs where precision and mathematical accuracy can be sacraficed for performace. Fully accurate math functions can be found in the *Math Library*.

This library is supported on both the PPE and SPE.

Name(s)

libgmath.a

Header File(s)

<libgmath.h>

6.1 cos8, cos14, cos18

C Specification

```
#include <cos8.h>
inline float _cos8(float angle)

#include <cos8_v.h>
inline vector float _cos8_v(vector float angle)

#include <cos14.h>
inline float _cos14(float angle)

#include <cos14_v.h>
inline vector float _cos14_v(vector float angle)

#include <cos18.h>
inline float _cos18(float angle)

#include <cos18_v.h>
inline vector float _cos18_v(vector float angle)

#include <libgmath.h>
float cos8(float angle)

#include <libgmath.h>
vector float cos8(vector float angle)

#include <libgmath.h>
float cos14(float angle)

#include <libgmath.h>
vector float cos14(vector float angle)

#include <libgmath.h>
float cos18(float angle)

#include <libgmath.h>
vector float cos18(vector float angle)
```

Descriptions

The *cos8*, *cos14*, and *cos18* subroutines compute the cosine of the input angle(s) specified by the parameter *angle*. The input angle is expressed in radians.

cos8, *cos14*, and *cos18* are accurate to (approximately) at least 8, 14, and 18 bits respectively for all angles in the -2 PI to 2 PI. Accuracy degrades the further the input angle is outside this range.

cos8 computes the cosine using an 8 segment piece wise quadratic approximation over the interval [0, 2*PI). *cos14* also uses an 8 segment piece wise quadratic approximation, but over the interval [0, 0.5*PI).

Symmetry is exploited to generate results for the entire [0, 2*PI) interval. *cos18* uses a 8 segment piece wise cubic approximation over the interval [0, 0.5*PI).

Dependencies

See Also

sin8, sin14, sin18 on page 39

6.2 pack_color8

C Specification

```
#include <pack_color8.h>
inline unsigned int _pack_color8(vector float rgba)

#include <libgmath.h>
unsigned int pack_color8(vector float rgba)
```

Descriptions

The *pack_color8* subroutine clamps a vectored floating point color to the normalized range 0.0 to 1.0, converts each component to a 8-bit fixed point number, and packs the 4 components into a 32-bit unsigned integer. The vectored floating-point color consists of 4 red, green, blue, and alpha color components.

Dependencies

See Also

pack_rgba8 on page 38
unpack_color8 on page 43

6.3 pack_normal16

C Specification

```
#include <pack_normal16.h>
inline signed short _pack_normal16(float normal)

#include <pack_normal16_v.h>
inline double _pack_normal16_v(vector float normal)

#include <libgmath.h>
signed short pack_normal16(float normal)

#include <libgmath.h>
double pack_normal16_v(vector float normal)
```

Descriptions

The *pack_normal16* subroutine take a floating-point normal component and packs it into a fixed-point 16-bit value. The vectored form of this function takes 4 floating point normal components and packs them into 64 bits (i.e., 4 16-bit packed fixed-point values).

This subroutine i1s designed to work on values (like normals) that are in the nominal range -1.0 to 1.0. Values outside this are wrapped producing undefined behavior. However, code supports extending the range to efficiently handle extended or reduced ranges. See <normal16.h>.

unpack_normal16 can be used to unpack a 16-bit normal back into full 32-bit floating point format.

Dependencies

See Also

unpack_normal16 on page 44

6.4 pack_rgba8

C Specification

```
#include <pack_rgba8.h>
inline unsigned int _pack_rgba8(float red, float green, float blue, float alpha)

#include <pack_rgba8_v.h>
inline vector unsigned int _pack_rgba8_v(vector float red, vector float green,
                                         vector float blue, vector float alpha)

#include <libgmath.h>
unsigned int pack_rgba(float red, float green, float blue, float alpha)

#include <libgmath.h>
vector unsigned int packr_gba8_v(vector float red, vector float green, vector float blue,
                                 vector float alpha)
```

Descriptions

The *pack_rgba8* subroutine clamps a 4 component normalized color (red, green, blue, and alpha) to the range 0.0 to 1.0, converts and packs it into a 32-bit, packed RGBA, 8-bits per component, fixed-point color. The vectored form clamps, converts, and packs 4 RGBA colors simultaneously.

Packed colors can be unpacked (one component at a time) using the *unpack_rgba8* subroutine.

Dependencies

See Also

unpack_rgba8 on page 45
pack_color8 on page 36

6.5 sin8, sin14, sin18

C Specification

```
#include <sin8.h>
inline float _sin8(float angle)

#include <cos8_v.h>
inline vector float _sin8_v(vector float angle)

#include <sin14.h>
inline float _sin14(float angle)

#include <sin14_v.h>
inline vector float _sin14_v(vector float angle)

#include <sin18.h>
inline float _sin18(float angle)

#include <sin18_v.h>
inline vector float _sin18_v(vector float angle)

#include <libgmath.h>
float sin8(float angle)

#include <libgmath.h>
vector float sin8(vector float angle)

#include <libgmath.h>
float sin14(float angle)

#include <libgmath.h>
vector float sin14(vector float angle)

#include <libgmath.h>
float sin18(float angle)

#include <libgmath.h>
vector float sin18(vector float angle)
```

Descriptions

The *sin8*, *sin14*, and *sin18* subroutines compute the sine of the input angle(s) specified by the parameter *angle*. The input angle is expressed in radians.

sin8, *sin14*, and *sin18* are accurate to (approximately) at least 8, 14, and 18 bits respectively for all angles in the 0.5*PI to 2.5*PI. Accuracy degrades the further the input angle is outside this range.

sin8, *sin14*, and *sin18* use the same underlying technique used by the *cos8*, *cos14*, and *cos18* subroutines by biasing the input angle by -0.5*PI and effectively calling the cosine function.

Dependencies

See Also

cos8, cos14, cos18 on page 34

6.6 set_spec_exponent9

C Specification

```
#include <set_spec_exponent9.h>
inline void _set_spec_exponent9(spec9Exponent *exp, signed int exponent)

#include <libgmath.h>
void set_spec_exponent9(spec9Exponent *exp, signed int exponent)
```

Descriptions

The set_spec_exponent9 subroutine computes exponent coefficient needed by the spec9 subroutine to compute the the power function of the form x^y. The exponent, specified by the exponent parameter, is an integer within the range 0 to 255. The coefficients are returned in the structure pointed to by exp.

Dependencies

recipf4 in SIMD Math library

See Also

spec9 on page 42

6.7 spec9

C Specification

```
#include <spec9.h>
inline float _spec9(float base, spec9Exponent *exp)

#include <spec9_v.h>
inline vector float _spec9_v(vector float base, spec9Exponent *exp)

#include <libgmath.h>
float spec9(float base, spec9Exponent *exp)

#include <libgmath.h>
vector float spec9_v(vector float base, spec9Exponent *exp)
```

Descriptions

The spec9 subroutine computes the power function of the form x^y for the limited set of values traditionally used in specular lighting. spec9 exploits the shuffle byte instruction to compute the power function using a 8 segment, piece wise quadratic approximation. The exponent (whose coefficients are computed by the set_spec_exponent9 subroutine and specified by the exponent parameter) is an integer within the range 0 to 255. The base (specified by the base parameter) is a floating point value in the range 0.0 to 1.0.

The quadratic coefficients are regenerated whenever there is a change (from call to call) of the exponent.

Results are accurate to at least (approximately) 9 bits of accuracy and are guaranteed to be continuous.

Base values less than 0.0 produces 0.0. Base value greater than 1.0 produce a 1.0. Undefined results will occur for exponents outside the 0-255 range.

Programmer Notes

The *spec9* subroutine has been structured so that repeated calculations using the same exponent can be made with minimal overhead. For each unique exponent, the exponent coefficients can be generated using the *set_spec_exponent9* subroutine. These coefficients can then be used multipe times to *spec9* subroutines calls.

Dependencies

See Also

set_spec_exponent9 on page 41

6.8 unpack_color8

C Specification

```
#include <unpack_color8.h>
inline vector float _unpack_color8(unsigned int rgba)

#include <libgmath.h>
vector float unpack_color8(unsigned int rgba)
```

Descriptions

The *unpack_color8* subroutine takes a 32-bit unsigned integer consisting of 4 8-bit packed color components and produces a vectored floating-point normalized color in which each channel of the vectored color is a separate channel - e.g., red, green, blue, and alpha.

Dependencies

See Also

pack_color8 on page 36
unpack_rgba8 on page 45

6.9 unpack_normal16

C Specification

```
#include <unpack_normal16.h>
inline float _unpack_normal16(float normal)

#include <unpack_normal16_v.h>
inline vector float _unpack_normal16_v(vector float normal)

#include <libgmath.h>
float unpack_normal16(float normal)

#include <libgmath.h>
vector float unpack_normal_v(vector float normal)
```

Descriptions

The *unpack_normal16* subroutine converts a signed 16-bits packed normal produced by the packNormal16 subroutine back into the floating-point normalized range -1.0 to 1.0. The vectored form of this function converts 4 packed normal components simultaneously.

Dependencies

See Also

pack_normal16 on page 37

6.10 unpack_rgba8

C Specification

```
#include <unpack_rgab8.h>
inline float _unpack_rgba8(unsigned int rgba, int component)

#include <unpack_rgba8_v.h>
inline vector float _unpack_rgab8_v(vector unsigned int rgba,
                                    int component)

#include <libgmath.h>
float unpack_rgba8(unsigned int rgba, int component)

#include <libgmath.h>
vector float unpackRGBA8_v(vector unsigned in rgba, int component)
```

Descriptions

The *unpack_rgba8* subroutine extracts one 8-bit fixed point color component from a packed color and returns the color component as a floating-point normalized (0.0 to 1.0) color component.

To maximize efficiency, a fixed point color component of 0xFF does not produce exactly 1.0. Instead, $1.0-2^{23}$ is produced.

Dependencies

See Also

pack_rgba8 on page 38
unpack_color8 on page 43

7. Image Library

The image library consists of a set of routines for processing images - arrays of data. The image library currently supports the following:

- Convolutions of varying size kernels with various image types.
- Histograms of byte data.

This library is supported on both the PPE and SPE.

Name(s)

libimage.a

Header File(s)

<libimage.h>

7.1 Convolutions

Image convolutions are supported for a number of small kernel sizes, including 3x3, 5x5, 7x7, and 9x9. Supported image formats are single component floating point (‘1f’), single component unsigned short (‘1us’), and four component unsigned byte (‘4ub’).

7.1.1 conv3x3_1f, conv5x5_1f, conv7x7_1f, conv9x9_1f

C Specification

```
#include <conv3x3_1f.h>
inline void _conv3x3_1f(const float *in[3], float *out, const vec_float4 m[9], int w)

#include <conv5x5_1f.h>
inline void _conv5x5_1f(const float *in[5], float *out, const vec_float4 m[25], int w)

#include <conv7x7_1f.h>
inline void _conv7x7_1f(const float *in[7], float *out, const vec_float4 m[49], int w)

#include <conv9x9_1f.h>
inline void _conv9x9_1f(const float *in[9], float *out, const vec_float4 m[81], int w)

#include <libimage.h>
void conv3x3_1f(const float *in[3], float *out, const vec_float4 m[9], int w)

void conv5x5_1f(const float *in[5], float *out, const vec_float4 m[25], int w)

void conv7x7_1f(const float *in[7], float *out, const vec_float4 m[49], int w)

void conv9x9_1f(const float *in[9], float *out, const vec_float4 m[81], int w)
```

Descriptions

Compute output pixels as the weighted sum of the input images's 3x3, 5x5, 7x7, or 9x9 neighborhood and the filter mask 'm'.

The image format is single component floating point. The filter mask 'm' represents an arbitrary 3x3, 5x5, 7x7, or 9x9 kernel, where each entry has been replicated from 'float' to 'vec_float4' form.

Border pixels require a policy for defining values outside the image. Three compile time options are supported. The default behaviour is to use _BORDER_COLOR_F (pre-defined to 0) for all values beyond the left or right edges of the input image. For values above or below the image, the caller is responsible for supplying scanlines cleared to the appropriate value.

When _WRAP_CONV is defined, the input values are periodically repeated -- in other words, the input wraps from left to right (and visa-versa). The caller is responsible for managing the input scanlines to support wrapping from top to bottom.

When _CLAMP_CONV is defined, the input values are clamped to the border -- in other words, the right most value is repeated for values beyond the right edge of the image; the left most value is repeated for values beyond the left edge of the image. The caller is responsible for managing the input scanlines to support clamping from top to bottom.

Dependencies

The input and output scanlines must be quad-word aligned. The scanline width 'w' must be a multiple of 16 pixels. Neither the input nor the output values are clamped or scaled to a fixed range.

See Also

conv3x3_1us, conv5x5_1us, conv7x7_1us, conv9x9_1us on page 50
conv3x3_4ub, conv5x5_4ub, conv7x7_4ub, conv9x9_4ub on page 52

7.1.2 conv3x3_1us, conv5x5_1us, conv7x7_1us, conv9x9_1us

C Specification

```
#include <conv3x3_1us.h>
inline void _conv3x3_1us (const unsigned short *in[3], unsigned short *out,
                          const vec_float4 m[9], int w)

#include <conv5x5_1us.h>
inline void _conv5x5_1us (const unsigned short *in[5], unsigned short *out,
                          const vec_float4 m[25], int w)

#include <conv7x7_1us.h>
inline void _conv7x7_1us (const unsigned short *in[7], unsigned short *out,
                          const vec_float4 m[49], int w)

#include <conv9x9_1us.h>
inline void _conv9x9_1us (const unsigned short *in[9], unsigned short *out,
                          const vec_float4 m[81], int w)

#include <libimage.h>
void conv3x3_1us (const unsigned short *in[3], unsigned short *out, const vec_float4 m[9],
                  int w)

void conv5x5_1us (const unsigned short *in[5], unsigned short *out, const vec_float4 m[25],
                  int w)

void conv7x7_1us (const unsigned short *in[7], unsigned short *out, const vec_float4 m[49],
                  int w)

void conv9x9_1us (const unsigned short *in[9], unsigned short *out, const vec_float4 m[81],
                  int w)
```

Descriptions

Compute output pixels as the weighted sum of the input images's 3x3, 5x5, 7x7, or 9x9 neighborhood and the filter mask 'm'.

The image format is single component unsigned short. The filter mask 'm' represents an arbitrary 3x3, 5x5, 7x7, or 9x9 kernel, where each entry has been converted to 'float' and replicated to 'vec_float4' form.

Border pixels require a policy for defining values outside the image. Three compile time options are supported. The default behaviour is to use _BORDER_COLOR_US (pre-defined to 0) for all values beyond the left or right edges of the input image. For values above or below the image, the caller is responsible for supplying scanlines cleared to the appropriate value.

When _WRAP_CONV is defined, the input values are periodically repeated --in other words, the input wraps from left to right (and visa-versa). The caller is responsible for managing the input scanlines to support wrapping from top to bottom.

When _CLAMP_CONV is defined, the input values are clamped to the border - in other words, the right most value is repeated for values beyond the right edge of the image; the left most value is repeated for values beyond the left edge of the image. The caller is responsible for managing the input scanlines to support clamping from top to bottom.

Dependencies

The input and output scanlines must be quad-word aligned. The scanline width 'w' must be a multiple of 16 pixels. Neither the input nor the output values are clamped or scaled to a fixed range.

See Also

conv3x3_1f, conv5x5_1f, conv7x7_1f, conv9x9_1f on page 48
conv3x3_4ub, conv5x5_4ub, conv7x7_4ub, conv9x9_4ub on page 52

7.1.3 conv3x3_4ub, conv5x5_4ub, conv7x7_4ub, conv9x9_4ub

C Specification

```
#include <conv3x3_4ub.h>
inline void _conv3x3_4ub(const unsigned int *in[3], unsigned int *out, const vec_int4 m[9],
                                                    int w, unsigned short scale, unsigned int shift)

#include <conv5x5_4ub.h>
inline void _conv5x5_4ub(const unsigned int *in[5], unsigned int *out, const vec_int4 m[25],
                                                    int w, unsigned short scale, unsigned int shift)

#include <conv7x7_4ub.h>
inline void _conv7x7_4ub(const unsigned int *in[7], unsigned int *out, const vec_int4 m[49],
                                                    int w, unsigned short scale, unsigned int shift)

#include <conv9x9_4ub.h>
inline void _conv9x9_4ub(const unsigned int *in[9], unsigned int *out, const vec_int4 m[81],
                                                    int w, unsigned short scale, unsigned int shift)

#include <libimage.h>
void conv3x3_4ub(const unsigned int *in[3], unsigned int *out, const vec_int4 m[9], int w,
                                                    unsigned short scale, unsigned int shift)

void conv5x5_4ub(const unsigned int *in[5], unsigned int *out, const vec_int4 m[25], int w,
                                                    unsigned short scale, unsigned int shift)

void conv7x7_4ub(const unsigned int *in[7], unsigned int *out, const vec_int4 m[49], int w,
                                                    unsigned short scale, unsigned int shift)

void conv9x9_4ub(const unsigned int *in[9], unsigned int *out, const vec_int4 m[81], int w,
                                                    unsigned short scale, unsigned int shift)
```

Descriptions

Compute output pixels as the weighted sum of the input images's 3x3, 5x5, 7x7, or 9x9 neighborhood and the filter mask 'm'.

The image format is our component unsigned byte, also known as packed integer. The filter mask 'm' represents an arbitrary 3x3, 5x5, 7x7, or 9x9 kernel, where each entry has been replicated to 'vec_int4' form.

Scaled integer arithmetic is used to compute the weighted sum. For masks whose components sum to zero or one (common for many sharpenning or edge-detect filters), values of 1 and 0 are appropriate for ‘scale’ and ‘shift’. For masks whose components sum to a value that is an an power of two (e.g. 8, 16, etc.), the ‘scale’ value is again 1, and the shift value should be the log2(sum). For masks whose components sum to a value that is not an power of two (common for many blurring or averaging filters), the ‘scale’ and ‘shift’ values may be computed as follows:

```
scale = 2**floor(log2(sum)) * 65535 / sum
shift = 16 + floor(log2(sum))
```

Border pixels require a policy for defining values outside the image. Three compile time options are supported. The default behaviour is to use _BORDER_COLOR_UB (pre-defined to 0) for all values beyond the left or right edges of the input image. For values above or below the image, the caller is responsible for supplying scanlines cleared to the appropriate value.

When _WRAP_CONV is defined, the input values are periodically repeated --in other words, the input wraps from left to right(and visa-versa). The caller is responsible for managing the input scanlines to support wrapping from top to bottom.

When _CLAMP_CONV is defined, the input values are clamped to the border --in other words, the right most value is repeated for values beyond the right edge of the image; the left most value is repeated for values beyond the left edge of the image. The caller is responsible for managing the input scanlines to support clamping from top to bottom.

Dependencies

The input and output scanlines must be quad-word aligned. The scanline width 'w' must be a multiple of 16 pixels. Neither the input nor the output values are clamped or scaled to a fixed range.

See Also

conv3x3_1f, conv5x5_1f, conv7x7_1f, conv9x9_1f on page 48
conv3x3_1us, conv5x5_1us, conv7x7_1us, conv9x9_1us on page 50

7.2 Histograms

7.2.1 histogram_ub

C Specification

```
#include <histogram_ub.h>
inline void _histogram_ub(unsigned int *counts, unsigned char *data, int size)

#include <libimage.h>
void histogram_ub(unsigned int *counts, unsigned char *data, int size)
```

Descriptions

The *histogram_ub* subroutine generates a histogram of characters (unsigned bytes) in the data array, *data*. The number of characters in the data array is specified by the *size* parameter. The *counts* array consists of 256 32-bit counters. It serves as both the input and output in that the count is adjusted according to the number of occurances of each byte in the data array.

The count array, *counts*, must be quadword aligned when computing a histogram on the SPE.

Dependencies

See Also

8. Large Matrix Library

The large matrix library consists of various utility functions that operate on large vectors as well as large matrices of single precision floating-point numbers.

The size of input vectors and matrices are limited by SPE local storage size.

This library is currently only supported on the SPE.

Name(s)

liblarge_matrix.a

Header File(s)

<liblarge_matrix.h>

8.1 index_max_abs_col

C Specification

```
#include <liblarge_matrix.h>
int index_max_abs_col(int n, float *A, int col, int stride);
```

Description

The *index_max_abs_col* subroutine finds the index of the maximum absolute value in the specified column of matrix *A*.

Parameters

n	the number of elements in the specified column
A	the matrix
col	the column
stride	row stride of matrix *A*

Dependencies

See Also

index_max_abs_vec on page 57

8.2 index_max_abs_vec

C Specification

```
#include <liblarge_matrix.h>
int index_max_abs_vec(int n, float *dx);
```

Description

The *index_max_abs_vec* subroutine finds the index of the maximum absolute value in the array of floating point numbers pointed to by *dx*.

Parameters

n	the number of elements in the array *dx*
dx	array of floating point numbers

Dependencies

See Also

index_max_abs_col on page 56

8.3 lu2_decomp

C Specification

```
#include <liblarge_matrix.h>
int lu2_decomp(int m, int n, float *A, int lda, int *ipiv)

#include <liblarge_matrix.h>
int lu3_decomp(int m, int n, float *A, int lda, int *ipiv)
```

Description

The *lu2_decomp* and *lu3_decomp* subroutines compute the LU factorization of a dense general *m* by *n* matrix *a* using partial pivoting with row interchanges. The factorization is done in place.

The factorization has the form:

$$[A] = [P][L][U]$$

where **P** is a permutation matrix, **L** is lower triangular with unit diagonal elements (lower trapezoidal if m > n) and **U** is upper triangular (upper trapezoidal if m < n).

Matrix *a* and vector *ipiv* must be quadword aligned

These are the right-looking Level 2 BLAS version of the algorithm. The *lu2_decomp* subroutine is suitable for computing the LU Decomposition of a narrow matrix where the number of rows is much greater than the number of columns. The *lu_decomp_3* subroutine should be used for general large square matrix since it is more efficient.

Parameters

m	number of rows of matrix *A*. *m* >= 0
n	number of columns of matrix *A*. *n* >= 0
A	on entry, this is the *m* by *n* matrix to be factored. On exit, the factors L and U from the factorization A = P*L*U; the unit diagonal elements of L are not stored.
lda	stride of matrix *A*
ipiv	on entry, this is just an empty array of integers. On output, this is an array of integers representing the pivot indices.

Returns

0	if successful
> 0	matrix is singular. U(j, j) = 0. The factorization has been completed but the factor U is exactly singular and division by zero will occur if it is used to solve a system of equations
< 0:	illegal input parameters

Dependencies

index_max_abs_col on page 56
scale_vector on page 68
swap_vectors on page 74
nmsub_number_vector on page 65

See Also

lu_decomp_block on page 60

8.4 lu_decomp_block

C Specification

```
#include <liblarge_matrix.h>
int lu_decomp_block (int m, int n, float *A, int lda, int *ipiv)
```

Description

The *lu_decomp_block* subroutine computes the LU factorization of a dense general *m* by *n* matrix A using partial pivoting with row interchanges. The factorization is done in place.

The factorization has the form

$$[A] = [P][L][U]$$

where **P** is a permutation matrix, **L** is lower triangular with unit diagonal elements (lower trapezoidal if *m* > *n*) and **U** is upper triangular (upper trapezoidal if *m* < *n*).

Matrix *a* and integer array *ipiv* must be quadword aligned.

This is the right-looking Level 3 BLAS version of the algorithm. This version of LU decomposition should be more efficient than the subroutine *lu_decomp* described above.

Parameters

m	number of rows of matrix *A*. *m* >= 0
n	number of columns of matrix *A*. *n* >= 0
A	on entry, this is the *m* by *n* matrix to be factored. On exit, the factors L and U from the factorization A = P*L*U; the unit diagonal elements of L are not stored.
lda	stride of matrix *A*
ipiv	on entry, this is just an empty array of integers. On output, this is an array of integers representing the pivot indices.

Returns

0	if successful
> 0	matrix is singular. U(j, j) = 0. The factorization has been completed but the factor U is exactly singular and division by zero will occur if it is used to solve a system of equations
< 0	illegal input parameters

Dependencies

lu2_decomp on page 58
swap_matrix_rows on page 73

solve_unit_lower on page 70
nmsub_matrix_matrix on page 63

See Also

lu2_decomp on page 58

Notes

LU Decomposition is done according to the blocked algorithm referenced in Jack Dongarra's paper. (***Fill in the name of the paper)***. The size of the block is set at compile time as BLOCKSIZE. Default size of BLOCKSIZE is 32 with 4, 8, 16, 32, 64 as valid BLOCKSIZE. The size of the matrix (*m* and *n*) do not have to be multiples of BLOCKSIZE, however, the algorithm works much more efficiently when *m* and *n* are multiples of BLOCKSIZE.

Only limited testing has been done for non-square matrix (*m* is different from *n*)

8.5 madd_matrix_matrix

C Specification

```
#include <liblarge_matrix.h>
void madd_matrix_matrix(int m, int p, int n, float *A, int lda, float *B, int ldb, float *C,
                        int ldc)
```

Description

The *madd_matrix_matrix* subroutine performs the matrix-matrix operation C = A*B + C, where A, B, and B are matrices.

Matrices *A*, *B*, and *C* are arranged in row-major order and must be quadword aligned. Parameters *m*, *n*, and *p* must be multiples of 4.

Parameters

m	number of rows of matrix *c* and of matrix *A*
p	number of columns of matrix *c* and number of columns of matrix *B*
n	number of columns of matrix *a* and number of rows of matrix *B*
A	an *m* by *n* matrix arranged in row-major order with a stride of *lda*
lda	stride of matrix *A*
B	an *n* by *p* matrix arranged in row-major order with a stride of *ldb*
ldb	stride of matrix *B*
C	an *m* by *p* matrix arranged in row-major order. On exit, the matrix *C* is overwritten by the resulting matrix

Dependencies

See Also

nmsub_matrix_matrix on page 63

8.6 nmsub_matrix_matrix

C Specification

```
#include <liblarge_matrix.h>
void nmsub_matrix_matrix(int m, int p, int n, float* A, int lda, float *B, int ldb, float *C,
                                                                                    int ldc)
```

Description

The *nmsub_matrix_matrix* subroutine performs the matrix-matrix operation C = C - A*B, where A, B, and C are matrices.

Matrices *A*, *B*, and *C* are arranged in row-major order, and must be quadword aligned. Parameters *m*, *n*, and *p* must be multiples of 4.

Parameters

m	number of rows of matrix *c* and of matrix *A*
p	number of columns of matrix *c* and number of columns of matrix *B*
n	number of columns of matrix *a* and number of rows of matrix *B*
A	an *m* by *n* matrix arranged in row-major order with a stride of *lda*
lda	stride of matrix *A*
B	an *n* by *p* matrix arranged in row-major order with a stride of *ldb*
ldb	stride of matrix B
C	an *m* by *p* matrix arranged in row-major order. On exit, the matrix *c* is overwritten by the resulting matrix
ldc	stride of matrix C

Dependencies

See Also

madd_matrix_matrix on page 62

8.7 madd_number_vector

C Specification

```
#include <liblarge_matrix.h>
void madd_number_vector(int n, float da, float x[], float y[])
```

Description

The *madd_number_vector* subroutine performs the product of the number *da* and the vector *x* . The resulting vector is added to the vector *y* .

$$y = da*x + y$$

Arrays *x* and *y* do **not** have to be quadword aligned, however, the last 2 hex digits of their addresses must be the same.

Parameters

n	size of arrays *x* and *y*
da	scaling factor
x	*n*-element array
y	*n*-element array

Dependencies

See Also

nmsub_number_vector on page 65
madd_vector_vector on page 66
scale_vector on page 68

8.8 nmsub_number_vector

C Specification

```
#include <liblarge_matrix.h>
void nmsub_number_vector (int n, float da, float x[], float y[])
```

Description

The *nmsub_number_vector* subroutine performs the product of the number *da* and the vector *x* . The resulting vector is subtracted from the vector *y* .

$$y = y - da*x$$

Arrays *x* and *y* do **not** have to be quadword aligned, however, the last 2 hex digits of their addresses must be the same.

Parameters

n	size of arrays *x* and *y*
da	scaling factor
x	*n*-element array
y	*n*-element array

Dependencies

See Also

madd_number_vector on page 64
nmsub_vector_vector on page 67
scale_vector on page 68

8.9 madd_vector_vector

C Specification

```
#include <liblarge_matrix.h>
void madd_vector_vector (int m, int n, float *col, int c_stride, float *row, float *A,
                                                                    int a_stride)
```

Description

The *madd_vector_vector* subroutine performs the vector-vector operation:

$$A = A + col*row$$

where *A* is an *m* by *n* matrix, *row* is a *n* elements row-vector and *col* is an *m* elements column-vector with an element stride of *a_stride*. *col*, *row*, and matrix *A* do not have to be quadword aligned. However, the least significant 2 bits of the addresses of vector *row* and matrix *A* must match.

Dependencies

See Also

nmsub_vector_vector on page 67
madd_number_vector on page 64

8.10 nmsub_vector_vector

C Specification

```
#include <liblarge_matrix.h>
void nmsub_vector_vector (int m, int n, float *col, int c_stride, float *row, float *A,
                          int a_stride)
```

Description

The *nmsub_vector_vector* subroutine performs the vector-vector operation:

$$A = A - col * row$$

where *A* is an *m* by *n* matrix, *row* is a *n* elements row-vector and *col* is an *m* elements column-vector with an element stride of *c_stride*. *col*, *row*, and matrix *A* do not have to be quadword aligned however, the least significant 2 bits of the addresses of vector *row* and matrix *A* must match.

Dependencies

See Also

madd_vector_vector on page 66
nmsub_number_vector on page 65

8.11 scale_vector

C Specification

```
#include <liblarge_matrix.h>
void scale_vector(int n, float scale_factor, float *x)
```

Description

The *scale_vector* subroutine scales each element of the *n element vector x* by the specified *scale_factor value.*

```
x = scale_factor*x
```

where *x* is an *n* element vector (array) and scale factor is a single precision floating point number. *n* must be at least 4.

Dependencies

See Also

scale_matrix_col on page 69
madd_number_vector on page 64
nmsub_number_vector on page 65

8.12 scale_matrix_col

C Specification

```
#include <liblarge_matrix.h>
void scale_matrix_col(int n, float scale_factor, float *A, int col, int stride)
```

Description

The *scale_matrix_col subroutine* performs the operation:

A[col] = scale_factor*A[col]

where A is matrix with *n* rows and at least *col* columns, *scale_factor* is a single precision floating-point number, and *stride* is stride for matrix *A*. *n* must be a multiple of 4 and *A* must be quadword aligned.

Dependencies

See Also

scale_vector on page 68
madd_number_vector on page 64
nmsub_number_vector on page 65

8.13 solve_unit_lower

C Specification

```
#include <liblarge_matrix.h>
void solve_unit_lower(int m, int n, const float *A, int lda, float *B, int ldb)
```

Description

The *solve_unit_lower* subroutine solves the matrix equation

$$A*X = B$$

where A is a unit lower triangular square matrix of size *m*, X is an *m by n* matrix, and B is an *m* by *n* matrix.

The solution *X* is returned in the matrix *B. A* and *B* must be quadword aligned and *m* and *n* must be multiples of 4.

Inputs

m	number of rows and columns of matrix *A*, number of rows of matrix *B*
n	number of columns of matrix *B*
A	unit lower triangular square matrix of size *m*
lda	stride of matrix *A*
B	general matrix of size *m* by *n*
ldb	stride of matrix *B*

Output

B	solution to matrix equations A*X = B

Dependencies

See Also

solve_unit_lower_1 on page 71
solve_upper_1 on page 72
solve_linear_system_1 on page 75

8.14 solve_unit_lower_1

C Specification

```
#include <liblarge_matrix.h>
void solve_unit_lower_1(int m, const float *A, int lda, float *b)
```

Description

The *solve_unit_lower* subroutine solves the matrix equation

$$A*x = b$$

where A is a unit lower triangular square matrix of size *m*, x and b are *m* element vectors.

The solution *x* is returned in vector *b. A* and *b* must be quadword aligned, *m* must be multiple of 4

Inputs

m	number of rows and columns of matrix *A*, number of elements of vector b
A	unit lower triangular square matrix of size *m*
lda	stride of matrix *A*
b	vector of length *m*

Outputs

b	solution x to equation A*x = b

Dependencies

See Also

solve_unit_lower on page 70
solve_upper_1 on page 72
solve_linear_system_1 on page 75

8.15 solve_upper_1

C Specification

```
#include <liblarge_matrix.h>
void solve_upper_1(int m, const float *A, int lda, float *b)
```

Description

The *solve_unit_lower* subroutine solves the matrix equation

$$A*x = b$$

where A is a unit upper triangular square matrix of size *m*, x and b are *m* element vectors.

The solution *x* is returned in vector *b. A* and *b* must be quadword aligned, *m* must be multiple of 4

Inputs

m	number of rows and columns of matrix *A*, number of elements of vector b
A	unit upper triangular square matrix of size *m*
lda	stride of matrix *A*
b	vector of length *m*

Outputs

b	solution x to equation A*x = b

Dependencies

See Also

solve_unit_lower on page 70
solve_unit_lower_1 on page 71
solve_linear_system_1 on page 75

8.16 swap_matrix_rows

C Specification

```
#include <liblarge_matrix.h>
void swap_matrix_rows(int n, float *A, int lda, int k1, int k2, int *ipiv)
```

Description

This *swap_matrix_rows* subroutine performs a series of row interchanges on the matrix *A*. The rows are interchanged, one row at a time starting with row *k1* and continues up to (but not including) row *k2.* The row is interchanged with the row specified in the corresponding array element of *ipiv.*

```
for (i=k1; i<k2; i++) {
    swap rows i and ipiv[i] of matrix A
}
```

The matrix *A* contains *n* columns with a row stride of *lda*.

Parameters

n	number of columns in matrix *A*
A	a *n* column matrix in column major order with a stride of *lda*
lda	stride of matrix *A*
k1	the first row to be swapped
k2	the row following the last row to be swapped
ipiv	an array of row indices to be swapped with

Dependencies

See Also

swap_vectors on page 74

8.17 swap_vectors

C Specification

```
#include <liblarge_matrix.h>
void swap_vectors(int n, float *sx, float *sy)
```

Description

The *swap_vectors* subroutine interchanges two vectors, *sx* and *sy,* of length *n*.

Both sx and sy must be quad_word aligned

Dependencies

See Also

swap_matrix_rows on page 73

8.18 solve_linear_system_1

C Specification

```
#include <liblarge_matrix.h>
int solve_linear_system_1(int n, float *A, int lda, int *ipiv, float *b)
```

Description

The *solve_linear_system* subroutine computes the solution to a real system of linear equations

A*x = b

where A is a square *n* by *n* matrix, and x and b are *n* element vectors. The resulting solution is returned in vector *b.*

The LU decomposition with partial pivoting and row interchanges is used to factor matrix *A* as

A = P*L*U

where P is a permutation matrix, L is a unit lower triangular, and U is a upper triangular. The factored form of A is then used to solve the system of equations A*x = b

Parameters

n	size of matrix *A*, must be a multiple of 4
A	On entry, *n* by *n* coefficient matrix A. On exit, the factors L and U from the LU factorization
lda	stride of matrix *A*
ipiv	*n* element vector of integers. On exit, it has the pivot indices that define the permutation matrix P; row I of matrix was interchanged with row ipiv[i]
b	On entry, the n element vector representing the right hand side. On exit, if the return code is 0, this contains the solution x of the linear equation A*x = b

Returns:

0	if successful
> 0	U(i,i) is exactly zero. The factorization has been completed but the factor U is exactly singular so the solution could not be computed
< 0	illegal inputs

Dependencies

lu_decomp_block on page 60
swap_matrix_rows on page 73
solve_unit_lower_1 on page 71
solve_upper_1 on page 72

See Also

lu_decomp_block on page 60
swap_matrix_rows on page 73
solve_unit_lower_1 on page 71
solve_upper_1 on page 72

8.19 transpose_matrix

C Specification

```
#include <liblarge_matrix.h>
void transpose_matrix(int m, int n, float *A, int lda, float *B, int ldb)
```

Description

The *transpose_matrix* subroutine performs the transpose operation on matrix *A* and returns the resulting transpose matrix in *B*. Matrices *A* and *B* are *m* by *n* with rows strides of *lda* and *lba respectively.*

The number of row (*m*), the number of columns (*n*), and the row strides of the input matrix (*A*) and output matrix (*B*), must be a multiple of 4 to keep all rows quadword aligned.

Parameters

m	number of rows in matrix *A* and cols in *A*
n	number of columns in matrix *A* and rows in *B*
A	pointer to matrix to be transposed. Matrix must be quadword aligned
lda	stride of matrix *A*
B	pointer to matrix *B*, matrix must be quadword aligned
ldb	leading dimension of matrix *B*

Dependencies

See Also

9. Matrix Library

The matrix library consists of various utility libraries that operate on matrices as well as quaternions. The library is supported on both the PPE and SPE.

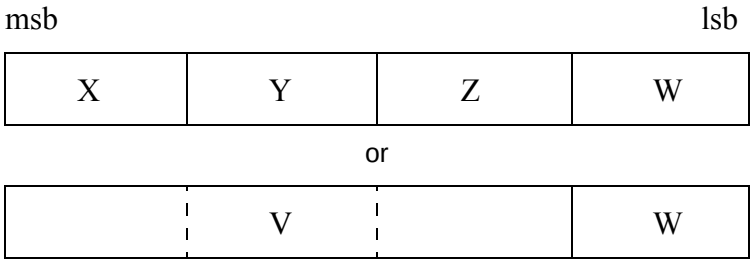
Unless specifically noted, all 4x4 matrices are maintained as an array of 4 128-bit SIMD vectors containing matrix entries as follows:

	msb			lsb
0	m[0]	m[1]	m[2]	m[3]
1	m[4]	m[5]	m[6]	m[7]
2	m[8]	m[9]	m[10]	m[11]
3	m[12]	m[13]	m[14]	m[15]

Double precision 4x4 matrices are defined as an array of 8 128-bit SIMD vectors containing matrix entries as follows:

	msb	lsb
0	m[0]	m[1]
1	m[2]	m[3]
2	m[4]	m[5]
3	m[6]	m[7]
4	m[8]	m[9]
5	m[10]	m[11]
6	m[12]	m[13]
7	m[14]	m[15]

Quaternions are stored as 4 component SIMD vector..



where V is a 3 component vector.

9.1 cast_matrix4x4_to_

C Specification

```
#include <cast_matrix4x4_to_dbl.h>
inline void _cast_matrix4x4_to_dbl(vector double *out, vector float *in)

#include <cast_matrix4x4_to_flt.h>
inline void _cast_matrix4x4_to_flt(vector float *out, vector double *in)

#include <libmatrix.h>
void cast_matrix4x4_to_dbl(vector double *out, vector float *in)

#include <libmatrix.h>
void cast_matrix4x4_to_flt(vector float *out, vector double *in)
```

Descriptions

The *cast_matrix4x4_to_dbl* subroutine converts a 4x4 single-precision floating-point matrix into a double precision matrix.

The *cast_matrix4x4_to_flt* subroutine converts a 4x4 double-precision floating-point matrix into a single precision matrix.

The input and output matrices are pointed to by *in* and *out* respectively and are both 128-bit aligned.

Dependencies

See Also

9.2 frustum_matrix4x4

C Specification

```
#include <frustum_matrix4x4.h>
inline void _frustum_matrix4x4(vector float *out, float left,
                                        float right, float bottom, float top, float near, float far)

#include <libmatrix.h>
void frustum_matrix4x4(vector float *out, float left, float right,
                                        float bottom, float top, float near, float far)
```

Descriptions

The *frustum_matrix4x4* subroutine constructs a 4x4 perspective projection transformation matrix and stores the result to *out*. The frustum matrix matches that of OpenGL's glFrustum function as it is computed as follows:

$$\text{out} = \begin{bmatrix} 2\times n/(r-l) & 0 & (r+l)/(r-l) & 0 \\ 0 & 2\times n/(t-b) & (t+b)/(t-b) & 0 \\ 0 & 0 & (-(f+n))/(f-n) & -2\times f\times n/(f-n) \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

where l, r, b, t, n and f correspond to the input parameters *left*, *right*, *bottom*, *top*, *near*, and *far*, respectively.

Dependencies

recipf4 in SIMD Math library

See Also

ortho_matrix4x4 on page 87
perspective_matrix4x4 on page 88

9.3 identity_matrix4x4

C Specification

```
#include <identity_matrix4x4.h>
inline void _identity_matrix4x4(vector float *out)

#include <libmatrix.h>
void identity_matrix4x4(vector float *out)
```

Descriptions

The *identity_matrix4x4* subroutine constructs a 4x4 identity matrix and stores the matrix into *out.* The 4x4 identity matrix is:

$$\text{out} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Dependencies

See Also

9.4 inverse_matrix4x4

C Specification

```
#include <inverse_matrix4x4.h>
inline int _inverse_matrix4x4(vector float *out, const vector float *in)

#include <libmatrix.h>
int inverse_matrix4x4(vector float *out, const vector float *in)
```

Descriptions

The *inverse_matrix4x4* subroutine computes the inverse of the 4x4 matrix pointed to by *in* and store the result into the 4x4 matrix pointed to by *out.* The inverse is computed using Kramer's rule and exploits SIMD to achieve significant performance improvements over simple scalar code.

If the input matrix, *in*, is found to be singular, the inverse is not computed and a non-zero value is returned. Otherwise, zero is returned.

Dependencies

See Also

9.5 mult_matrix4x4

C Specification

```
#include <mult_matrix4x4.h>
inline void _mult_matrix4x4(vector float *out, const vector float *m1,
                                                const vector float *m2)

#include <mult_matrix4x4.h>
inline void _mult_matrix4x4(vector float *out, const vector float *m1,
                                                const vector float *m2)

#include <libmatrix.h>
void mult_matrix4x4(vector float *out, const vector float *m1,
                                                const vector float *m2)

#include <libmatrix.h>
void mult_matrix4x4(vector float *out, const vector float *m1,
                                                const vector float *m2)
```

Descriptions

The *mult_matrix4x4* subroutine multiples the two input 4x4 floating-point matrices, *m1* and *m2*, and places the result in *out.*

$$[out] = [m1] \times [m2]$$

Both single precision and double precision matrix multiplies are supported.

Dependencies

See Also

9.6 mult_quat

C Specification

```
#include <mult_quat.h>
inline vector float_mult_quat(vector float q1, vector float q2)

#include <libmatrix.h>
void mult_quat(vector float q1, vector float q2)
```

Descriptions

The *mult_quat* subroutine multiplies unit length input quaternions *q1* and *q2* and returns the resulting quaternion. The product of two unit quaternions is the composite of the *q1* rotation followed by the *q2* rotation.

$$q1 \times q2 = [(v1 \times v2) + (w1 \times v2) + (w2 \times v1), w1 \times w2 - (v1 \bullet v2)]$$

where: *q1*=[*v1*,*w1*] and *q2*=[*v2*,*w2*]

Dependencies

See Also

quat_to_rot_matrix4x4 on page 89
rot_matrix4x4_to_quat on page 91

9.7 ortho_matrix4x4

C Specification

```
#include <ortho_matrix4x4.h>
inline void _ortho_matrix4x4(vector float *out, float left, float right,
                                                float bottom, float top, float near, float far)

#include <libmatrix.h>
void ortho_matrix4x4(vector float *out, float left, float right,
                                                float bottom, float top, float near, float far)
```

Descriptions

The *ortho_matrix4x4* subroutine constructs a 4x4 orthographic projection transformation matrix and stores the result to *out*. The ortho matrix matches that of OpenGL's glOrtho function as it is computed as follows:

$$out = \begin{bmatrix} 2/(r-l) & 0 & 0 & (r+l)/(r-l) \\ 0 & 2/(t-b) & 0 & (t+b)/(t-b) \\ 0 & 0 & (-2)/(f-n) & (-(f+n))/(f-n) \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where l, r, b, t, n and f correspond to the input parameters *left*, *right*, *bottom*, *top*, *near*, and *far*, respectively.

Dependencies

recipf4 in SIMD Math library

See Also

frustum_matrix4x4 on page 82
perspective_matrix4x4 on page 88

9.8 perspective_matrix4x4

C Specification

```
#include <perspective_matrix4x4.h>
inline void _perspective_matrix4x4(vector float *out, float fovy,
                                   float aspect, float near, float far)

#include <libmatrix.h>
void perspective_matrix4x4(vector float *out, float fovy, float aspect,
                                   float near, float far)
```

Descriptions

The *perspective_matrix4x4* subroutine constructs a 4x4 perspective projection transformation matrix and stores the result to *out*. The perspective matrix matches that of OpenGL's glPerspective function as it is computed as follows:

$$out = \begin{bmatrix} (\cot((fovy)/2))/(aspect) & 0 & 0 & 0 \\ 0 & \cot((fovy)/2) & 0 & 0 \\ 0 & 0 & (f+n)/(n-f) & 2 \times f \times n/(n-f) \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

where n and f correspond to the input parameters *near*, and *far*, respectively.

Dependencies

recipf4 in SIMD Math library
tanf4 in SIMD Math library

See Also

ortho_matrix4x4 on page 87
perspective_matrix4x4 on page 88

9.9 quat_to_rot_matrix4x4

C Specification

```
#include <quat_to_rot_matrix4x4.h>
inline void _quat_to_rot_matrix4x4(vector float *out, vector float quat)

#include <libmatrix.h>
void quat_to_rot_matrix4x4(vector float *out, vector float quat)
```

Descriptions

The *quat_to_rot_matrix4x4* subroutine converts the unit quaternion *quat* into a 4x4 floating-point rotation matrix. The rotation matrix is computed from the unit quaternion [x, y, x, w] as follows:

$$out = \begin{bmatrix} 1-2\times y\times y-2\times z\times z & 2\times x\times y-2\times z\times w & 2\times x\times z+2\times y\times w & 0 \\ 2\times x\times y+2\times z\times w & 1-2\times x\times x-2\times z\times z & 2\times y\times z+2\times x\times w & 0 \\ 2\times x\times z-2\times y\times w & 2\times y\times z+2\times x\times w & 1-2\times x\times x-2\times y\times y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Dependencies

See Also

rot_matrix4x4_to_quat on page 91

9.10 rotate_matrix4x4

C Specification

```
#include <rotate_matrix4x4.h>
inline void _rotate_matrix4x4(vector float *out, vector float vec,
                                                 float angle)

#include <libmatrix.h>
void rotate_matrix4x4(vector float *out, vector float vec, float angle)
```

Descriptions

The *rotate_matrix4x4* subroutine constructs a 4x4 floating-point matrix the performs a rotation of *angle* radians about the normalized (unit length) vector *vec*. The resulting rotation matrix is store to *out.*

The rotation matrix is computed as follows:

$$\begin{bmatrix} X \times X \times (1-C) + C & X \times Y \times (1-C) - Z \times S & X \times Z \times (1-C) + Y \times S & 0 \\ Y \times X \times (1-C) + Z \times S & Y \times Y \times (1-C) + C & Y \times Z \times (1-C) - X \times S & 0 \\ Z \times X \times (1-C) - Y \times S & Z \times Y \times (1-C) + X \times S & Z \times Z \times (1-C) + C & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where: ***X***, ***Y***, ***Z*** are the components of *vec*; ***C*** and ***S*** is the cosine and sine of *angle*.

Dependencies

See Also

9.11 rot_matrix4x4_to_quat

C Specification

```
#include <rot_matrix4x4_to_quat.h>
inline vector float _rot_matrix4x4_to_quat(vector float *matrix)

#include <libmatrix.h>
vector float rot_matrix4x4_to_quat(vector float *matrix)
```

Descriptions

The *rot_matrix4x4_to_quat* subroutine converts floating-point rotation matrix into a unit quaternion and returns the results. The rotation matrix is the upper-left 3x3 of the 4x4 matrix specified by the *matrix* parameter and is assumed to have a positive trace (i.e., the sum of the diagonal entries, *matrix*[0][0], *matrix*[1][1] and *matrix*[2][2], is greater than 0.

Dependencies

See Also

quat_to_rot_matrix4x4 on page 89

9.12 scale_matrix4x4

C Specification

```
#include <scale_matrix4x4.h>
inline void _scale_matrix4x4(vector float *out, vector float *in,
                                                vector float scales)

#include <libmatrix.h>
void scale_matrix4x4(vector float *out, vector float *in,
                                                vector float scales)
```

Descriptions

The *scale_matrix4x4* subroutine multiplies the 4x4 floating-point matrix *in* by a scale matrix defined by the *scales* parameter and returns the resulting matrix in *out.*

$$\begin{bmatrix}\text{out}\end{bmatrix} = \begin{bmatrix}\text{in}\end{bmatrix} \times \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & Sw \end{bmatrix}$$

where: *scales* = [*Sx*, *Sy*, *Sz*, *Sw*].

Dependencies

See Also

9.13 slerp_quat

C Specification

```
#include <slerp_quat.h>
inline vector float _slerp_quat(vector float q1, vector float q2,
                                float t)

#include <libmatrix.h>
vector float slerp_quat(vector float q1, vector float q2, float t)
```

Descriptions

The *slerp_quat* subroutine performs **s**pherical **l**inear int**erp**olation between two unit quaternions, *q1* and *q2*. Spherical linear interpolation is the interpolation of the shortest distance between orientations *q1* and *q2* along a great arc on the 4-D sphere. The interpolation factor, *t*, varies from 0.0 to 1.0 corresponding to orientations *q1* and *q2* respectively. Undefined results occur if *t* is outside the range [0.0, 1.0].

The slerp is computed as follows:

$$\text{slerp\_quat}(q1, q2, t) = \frac{q1 \times \sin((1-t) \times \phi) + q2 \times \sin(t \times \phi)}{\sin(\phi)}$$

$$\text{where:} \qquad \cos(\phi) = q1 \cdot q2$$

If the spherical distance between *q1* and *q2* is small, then linear interpolation is performed to maintain numeric stability.

Dependencies

sinf4 in SIMD Math library
divf4 in SIMD Math library
acosf4 in SIMD Math library

See Also

rot_matrix4x4_to_quat on page 91
quat_to_rot_matrix4x4 on page 89

9.14 splat_matrix4x4

C Specification

```
#include <spat_matrix4x4.h>
inline void _splat_matrix4x4(vector float *out, const vector float *in)

#include <libmatrix.h>
void splat_matrix4x4(vector float *out, const vector float *in)
```

Descriptions

The *splat_matrix4x4* subroutine converts a 4x4 floating-point matrix into a vector replicated matrix suitable for simultaneously transforming 4 independent vectors using SIMD vector operations. The input matrix, *in,* is a 4x4 matrix encoded as 4 128-bit vectors. This is equivalent to a quad word aligned 16 entry floating-point array. *splat_matrix4x4* takes each of the 16 32-bit entries and replicates it across a 128-bit floating-point vector and stores the result into the *out* output array.

Dependencies

See Also

9.15 transpose_matrix4x4

C Specification

```
#include <transpose_matrix4x4.h>
inline void _transpose_matrix4x4(vector float *out, vector float *in)

#include <libmatrix.h>
void transpose_matrix4x4(vector float *out, vector float *in)
```

Descriptions

The *transpose_matrix4x4* subroutine performs a matrix transpose of the 4x4 matrix *in* and stores the resulting matrix to *out.* This subroutine is capable of performing a transpose on itself (i.e., *in* can equal *out*).

This routine can also be used to convert a 4 element array of 4-component coordinates and return 4 4-element parallel arrays. Eg:

Address Offset	In	Out
0	x1	x1
4	y1	x2
8	z1	x3
12	w1	x4
16	x2	y1
20	2	y2
24	z2	y3
28	w2	y4
32	x3	z1
36	y3	z2
40	z3	z3
44	w3	z4
48	x4	w2
52	y4	w2
56	z4	w3
60	w4	w3

Dependencies

See Also

10. Misc Library

The misc library consists of a set of general purpose routines that don't logically fit within any of the specific libraries. The library is supported on both the PPE and SPE.

Name(s)

libmisc.a

Header File(s)

<libmisc.h>

10.1 calloc_align

C Specification

```
#include <libmisc.h>
void *calloc_align(size_t nmemb, size_t size, unsigned int log2_align)

#include <calloc_align.h>
inline void *_calloc_align(size_t nmemb, size_t size, unsigned int log2_align)
```

Description

The *calloc_align* subroutine attempts to allocate at least *size* bytes from local store memory heap with a power of 2 byte alignment of 2^{log2_align}. For example, a call of:

calloc_align(4096, 7).

allocates a memory heap buffer of 4096 bytes aligned to a 128 byte boundary.

If the requested *size* cannot be allocated due to resource limitations, or if *size* is less than or equal to zero, *calloc* returns NULL. On success, *calloc_align* returns a non-NULL, properly aligned local store pointer and the memory is set to zero.

To free or re-allocate a memory buffer allocated by *calloc_align, free_align* or *realloc_align* must be used.

Dependencies

calloc in newlib

See Also

free_align on page 104
malloc_align on page 106
realloc_align on page 118

10.2 clamp_0_to_1

C Specification

```
#include <clamp_0_to_1.h>
inline float _clamp_0_to_1(float x)

#include <clamp_0_to_1_v.h>
inline vector float _clamp_0_to_1_v(vector float x)

#include <libmisc.h>
float clamp_0_to_1(float x)

#include <libmisc.h>
vector float clamp_0_to_1_v(vector float x)
```

Descriptions

The *clamp_0_to_1* subroutine clamps floating-point the input value *x* to the range 0.0 to 1.0 and returns the result. Clamping is performed using the HW clamping performed during float to unsigned integer conversion, so the actual clamp range is 0.0 to 1.0-*epsilon*.

The *clamp_0_to_1_v* subroutine performs 0.0 to 1.0 clamping on a vector of 4 independent floating-point values.

Dependencies

See Also

clamp on page 100
clamp_minus1_to_1 on page 101

10.3 clamp

C Specification

```
#include <clamp.h>
inline float _clamp(float x, float min, float max)

#include <clamp_v.h>
inline vector float _clamp_v(vector float x, vector float min, vector float max)

#include <libmisc.h>
float clamp(float x, float min, float max)

#include <libmisc.h>
vector float clamp_v(vector float x, vector float min, vector float max)
```

Descriptions

The *clamp* subroutine clamps floating-point the input value *x* to the range specified by the *min* and *max* input parameters. It is assumed that *min* is less or equal to *max***.**

The *clamp_v* subroutine performs clamping on a vector of 4 independent floating-point values. The vectored clamp assumes the each component of the *min* vector is less than or equal to the corresponding component of the *max* vector.

Dependencies

See Also

clamp_0_to_1 on page 99
clamp_minus1_to_1 on page 101

10.4 clamp_minus1_to_1

C Specification

```
#include <clamp_minus1_to_1.h>
inline float _clamp_minus1_to_1(float x)

#include <clamp_minus1_to_1_v.h>
inline vector float _clamp_minus1_to_1_v(vector float x)

#include <libmisc.h>
float clamp_minus1_to_1(float x)

#include <libmisc.h>
vector float clamp_minus1_to_1_v(vector float x)
```

Descriptions

The *clamp_minus1_to_1* subroutine clamps floating-point the input value *x* to the range -1.0 to 1.0 and returns the result. Clamping is performed using the HW clamping performed during float to signed integer conversion, so the actual clamp range is -1.0+*epsilon* to 1.0-*epsilon*.

The *clamp_minus1_to_1_v* subroutine performs -1.0 to 1.0 clamping on a vector of 4 independent floating-point values.

Dependencies

See Also

clamp on page 100
clamp_0_to_1 on page 99

10.5 copy_from_ls

C Specification (SPE only)

```
#include <libmisc.h>
size_t copy_from_ls(uint64_t to, uint32_t from, size_t n)
```

Descriptions

The *copy_from_ls* subroutine copies *n* bytes from the local store address specified by *from* to the 64-bit effective address specified by *to*. This copy routine is synchronous (the copy is complete upon return) and supports any size (*n*) and alignment (of *to* and *from*). As such, this routine should not be used by applications wishing to maximize performance.

This routine returns the number of bytes copied - *n*.

This routine is only supported on the SPE.

Dependencies

memcpy in newlib

See Also

copy_to_ls on page 103

10.6 copy_to_ls

C Specification (SPE only)

```
#include <libmisc.h>
size_t copy_to_ls(uint32_t to, uint64_t from, size_t n)
```

Descriptions

The *copy_to_ls* subroutine copies *n* bytes from the 64-bit effective address specified by *from* to the local store address specified by *to.* This copy routine is synchronous (the copy is complete upon return) and supports any size (*n*) and alignment (of *to* and *from*). As such, this routine should not be used by applications wishing to maximize performance.

This routine returns the number of bytes copied - *n*.

This routine is only supported on the SPE.

Dependencies

memcpy in newlib

See Also

copy_from_ls on page 102

10.7 free_align

C Specification

```
#include <libmisc.h>
void free_align(void *ptr)

#include <free_align.h>
inline void _free_align(void *ptr)
```

Description

The *free_align* subroutine deallocates a block of local store memory previously allocated by *calloc_align*, *malloc_align*, or *realloc_align*. The memory to be freed is pointed to by *ptr.* If *ptr* is NULL, then no operation is performed.

Dependencies

free in newlib

See Also

calloc_align on page 98
malloc_align on page 106
realloc_align on page 118

10.8 load_vec_unaligned

C Specification

```
#include <load_vec_unaligned.h>
inline vector unsigned char _load_vec_unaligned(unsigned char *ptr)

#include <libmisc.h>
vector unsigned char load_vec_unaligned(unsigned char *ptr)
```

Descriptions

The *load_vec_unaligned* subroutine fetches the quadword beginning at the address specified by *ptr* and returns it as a unsigned character vector. This routine assumes that *ptr* is likely not aligned to a quadword boundary and therefore fetches the quadword containing the byte pointed to by *ptr* and the following quadword.

Dependencies

See Also

store_vec_unaligned on page 119

10.9 malloc_align

C Specification

```
#include <libmisc.h>
void *malloc_align(size_t size, unsigned int log2_align)

#include <malloc_align.h>
inline void *_malloc_align(size_t size, unsigned int log2_align)
```

Description

The *malloc_align* subroutine attempts to allocate at least *size* bytes from local store memory heap with a power of 2 byte alignment of 2^{log2_align}. For example, a call of:

malloc_align(4096, 7).

allocates a memory heap buffer of 4096 bytes aligned to a 128 byte boundary.

If the requested *size* cannot be allocated due to resource limitations, or if *size* is less than or equal to zero, *malloc_align* returns NULL. On success, *malloc_align* returns a non-NULL, properly aligned local store pointer.

To free or re-allocate a memory buffer allocated by *malloc_align, free_align* must be used.

Dependencies

malloc in c lib

See Also

calloc_align on page 98
free_align on page 104
posix_memalign or *memalign*
realloc_align on page 118

10.10 max_float_v

C Specification

```
#include <max_float_v.h>
inline vector float _max_float_v(vector float v1, vector float v2)

#include <libmisc.h>
vector float max_float_v(vector float v1, vector float v2)
```

Descriptions

The *max_float_v* subroutine returns the component-by-component maximum of two floating-point vectors, *v1* and *v2*.

Dependencies

See Also

max_vec_float on page 109
max_int_v on page 108
min_float_v on page 111

10.11 max_int_v

C Specification

```
#include <max_int_v.h>
inline vector signed int _max_int_v(vector signed int v1, vector signed int v2)

#include <libmisc.h>
vector signed int max_int_v(vector signed int v1, vector signed int v2)
```

Descriptions

The *max_int_v* subroutine returns the component-by-component maximum of two signed integer vectors, *v1* and *v2*.

Dependencies

See Also

max_vec_int on page 110
max_float_v on page 107
min_int_v on page 112

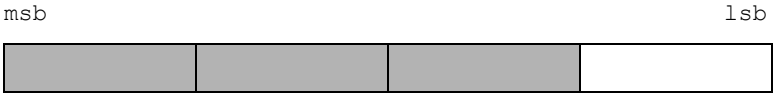
10.12 max_vec_float

C Specification

```
#include <max_vec_float3.h>
inline float _max_vec_float3(vector float v_in)

#include <max_vec_float4.h>
inline float _max_vec_float4(vector float v_in)

#include <libmisc.h>
float max_vec_float3(vector float v_in)

#include <libmisc.h>
float max_vec_float4(vector float v_in)
```

Descriptions

The *max_vec_float4* subroutine returns the maximum component of the 4-component, floating-point vector *v_in*. The *max_vec_float3* subroutine returns the maximum component of the 3 most signficant components of the floating-point vector *v_in.*



Dependencies

See Also

max_vec_int on page 110
max_vec_float on page 109

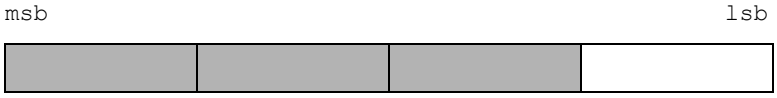
10.13 max_vec_int

C Specification

```
#include <max_vec_int3.h>
inline signed int _max_vec_int3(vector signed int v_in)

#include <max_vec_int4.h>
inline signed int _max_vec_float4(vector signed int v_in)

#include <libmisc.h>
signed int max_vec_int3(vector signed int v_in)

#include <libmisc.h>
float max_vec_int4(vector signed int v_in)
```

Descriptions

The *max_vec_int4* subroutine returns the maximum component of the 4-component, signed, integer vector *v_in*. The *max_vec_int3* subroutine returns the maximum component of the 3 most signficant components of the signed, integer vector *v_in.*



Dependencies

See Also

max_vec_float on page 109
min_vec_int on page 114

10.14 min_float_v

C Specification

```
#include <min_float_v.h>
inline vector float _min_float_v(vector float v1, vector float v2)

#include <libmisc.h>
vector float min_float_v(vector float v1, vector float v2)
```

Descriptions

The *min_float_v* subroutine returns the component-by-component minimum of two floating-point vectors, *v1* and *v2*.

Dependencies

See Also

min_vec_float on page 113
min_int_v on page 112
max_float_v on page 107

10.15 min_int_v

C Specification

```
#include <min_int_v.h>
inline vector signed int _min_int_v(vector signed int v1, vector signed int v2)

#include <libmisc.h>
vector signed int min_int_v(vector signed int v1, vector signed int v2)
```

Descriptions

The *min_int_v* subroutine returns the component-by-component minimum of two signed integer vectors, *v1* and *v2*.

Dependencies

See Also

min_vec_int on page 114
min_float_v on page 111
max_int_v on page 108

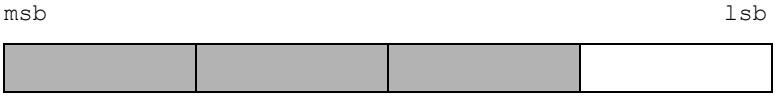
10.16 min_vec_float

C Specification

```
#include <min_vec_float3.h>
inline float _min_vec_float3(vector float v_in)

#include <min_vec_float4.h>
inline float _min_vec_float4(vector float v_in)

#include <libmisc.h>
float min_vec_float3(vector float v_in)

#include <libmisc.h>
float min_vec_float4(vector float v_in)
```

Descriptions

The *min_vec_float4* subroutine returns the minimum component of the 4-component, floating-point vector *v_in*. The *min_vec_float3* subroutine returns the minimum component of the 3 most signficant components of the floating-point vector *v_in.*



Dependencies

See Also

min_vec_int on page 114
max_vec_float on page 109

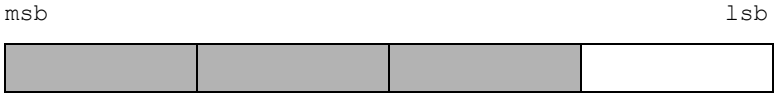
10.17 min_vec_int

C Specification

```
#include <min_vec_int3.h>
inline signed int _min_vec_int3(vector signed int v_in)

#include <min_vec_int4.h>
inline signed int _min_vec_float4(vector signed int v_in)

#include <libmisc.h>
signed int min_vec_int3(vector signed int v_in)

#include <libmisc.h>
float min_vec_int4(vector signed int v_in)
```

Descriptions

The *min_vec_int4* subroutine returns the minimum component of the 4-component, signed, integer vector *v_in*. The *min_vec_int3* subroutine returns the minimum component of the 3 most signficant components of the signed, integer vector *v_in.*



Dependencies

See Also

min_vec_float on page 113
max_vec_int on page 110

10.18 rand

C Specification (PPE only)

```
#include <rand_v.h>
inline vector signed int _rand_v(void)

#include <libmisc.h>
vector signed int rand_v(void)
```

Descriptions

The *rand_v* subroutine generates a vector of 31-bit uniformly cyclic, pseudo random numbers. This functions is also provided for the SPE in the C library.

Note: This random number implementation will never produce a random equal to 0 or 0x7FFFFFFF.

Dependencies

See Also

srand on page 120
rand_0_to_1 on page 117
rand_minus1_to_1 on page 116

10.19 rand_minus1_to_1

C Specification

```
#include <rand_minus1_to_1.h>
inline float _rand_minus1_to_1(void)

#include <rand_minus1_to_1_v.h>
inline vector float _rand_minus1_to_1_v(void)

#include <libmisc.h>
float rand_minus1_to_1(void)

#include <libmisc.h>
vector float rand_minus1_to_1_v(void)
```

Descriptions

The *rand_minus1_to_1* subroutine generates a uniformly cyclic, pseudo random number in the half closed interval [-1.0, 1.0).

The *rand_minus1_to_1_v* subroutine generates a vector of uniformly cyclic, pseudo random numbers in the half closed interval [-1.0, 1.0).

Dependencies

rand on page 115

See Also

srand on page 120
rand_0_to_1 on page 117

10.20 rand_0_to_1

C Specification

```
#include <rand_0_to_1.h>
inline float _rand_0_to_1(void)

#include <rand_0_to_1_v.h>
inline vector float _rand_0_to_1_v(void)

#include <libmisc.h>
float rand_0_to_1(void)

#include <libmisc.h>
vector float rand_0_to_1_v(void)
```

Descriptions

The *rand_0_to_1* subroutine generates a uniformly cyclic, pseudo random number in the half closed interval [0.0, 1.0).

The *rand_0_to_1_v* subroutine generates a vector of uniformly cyclic, pseudo random numbers in the half closed interval [0.0, 1.0).

Dependencies

rand on page 115

See Also

rand on page 115
rand_minus1_to_1 on page 116

10.21 realloc_align

C Specification

```
#include <libmisc.h>
void *realloc_align(void *ptr, size_t size, unsigned int log2_align)

#include <realloc_align.h>
inline void *_realloc_align(void *ptr, size_t size, unsigned int log2_align)
```

Description

The re*malloc_align* subroutine changes the size of the memory block pointed to by *ptr* to *size* bytes, aligned on a power of 2 byte alignment of 2^{log2_align}. The contents will be unchanged to the minumum of the old and new sizes; newly allocated memory will be uninitialized. If *ptr* is NULL, then the call is equivalent to *malloc_align*(*size, log2_align*). If *size* is equal to 0, then the call is equivalent to *free_align*(*ptr*). Unless *ptr* is NULL, its must have been returned by an earlier call to *malloc_align*, *calloc_align*, or *realloc_align*.

Dependencies

realloc in newlib

See Also

calloc_align on page 98
free_align on page 104
malloc_align on page 106

10.22 store_vec_unaligned

C Specification

```
#include <store_vec_unaligned.h>
inline void _store_vec_unaligned(unsigned char *ptr, vector unsigned char data)

#include <libmisc.h>
void store_vec_unaligned(unsigned char *ptr, vector unsigned char data)
```

Descriptions

The *store_vec_unaligned* subroutine stores a quadword/vector *data* to memory at the unaligned address specified by *ptr.* Data surrounding the quadword is unaffected by the store.

Dependencies

See Also

load_vec_unaligned on page 105

10.23 srand

C Specification (PPE only)

```
#include <srand_v.h>
inline void _srand_v(vector unsigned int seed)

#include <libmisc.h>
void srand_v(vector unsigned int seed)
```

Descriptions

The *srand_v* subroutine sets the random number seed used by the PPE vectorized random number generation subroutine - *rand*_v, *rand_0_to_1_v*, and *rand_minus1_to_1_v*. No restrictions are placed on the value of the seed yet only the 31 lsb (least significant bits) are saved.

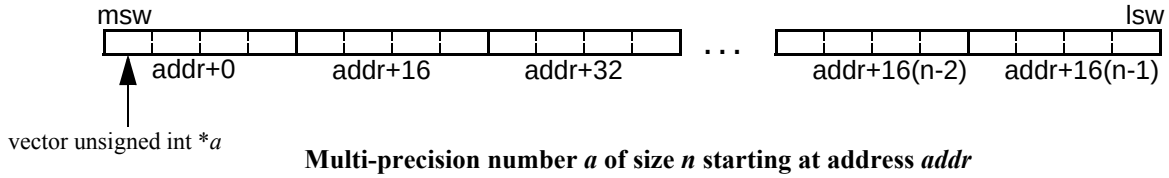
Dependencies

See Also

rand on page 115
rand_0_to_1 on page 117
rand_minus1_to_1 on page 116

11. Multi-Precision Math Library

The multi-precision math library consists of a set routines that perform mathematical functions on unsigned integers of a large number of bits. All multi-precision numbers are expressed as an array of unsigned integer vectors (vector unsigned int) of user specified length (in quadwords). The numbers are assumed to big endian ordered.



Multi-precision number *a* of size *n* starting at address *addr*

The compile time define, MPM_MAX_SIZE, specificies the maximum size (in quadwords) of an input multi-precision number. The default size is 32 cooresponding to 4096 bit numbers.

This library is currently only supported on the SPE.

Name(s)

libmpm.a

Header File(s)

<libmpm.h>

11.1 mpm_abs

C Specification

```
#include <mpm_abs.h>
inline void _mpm_abs(vector unsigned int *a, int size)

#include <libmpm.h>
void mpm_abs(vector unsigned int *a, int size)
```

Descriptions

The *mpm_abs* subroutine takes the absolute value of the multi-precision number pointed to by the parameter *a.* The number *a* is of *size* quadwords.

$$a = \text{abs}(a)$$

Dependencies

mpm_neg on page 141

See Also

11.2 mpm_add

C Specification

```
#include <mpm_add.h>
inline vector unsigned int _mpm_add(vector unsigned int *s, vector unsigned int *a,
                                    vector unsigned int *b, int size)
#include <mpm_add2.h>
inline int _mpm_add2(vector unsigned int *s, vector unsigned int *a, int a_size,
                                    vector unsigned int *b, int b_size)
#include <mpm_add3.h>
inline void _mpm_add3(vector unsigned int *s, int s_size, vector unsigned int *a, int a_size,
                                    vector unsigned int *b, int b_size)

#include <libmpm.h>
vector unsigned int mpm_add(vector unsigned int *s, vector unsigned int *a,
                                    vector unsigned int *b, int size)
#include <libmpm.h>
int _mpm_add2(vector unsigned int *s, vector unsigned int *a, int a_size,
                                    vector unsigned int *b, int b_size)
#include <libmpm.h>
void _mpm_add3(vector unsigned int *s, int s_size, vector unsigned int *a, int a_size,
                                    vector unsigned int *b, int b_size)
```

Descriptions

The *mpm_add* subroutine adds two multi-precision numbers of *size* quadwords pointed to by *a* and *b*. The result is stored in the array pointed to by *s*. The carry out of the sum is returned. A value of (0,0,0,1) is returned when a carry out occurred. Otherwise (0,0,0,0) is returned.

$$s = a + b$$

The *mpm_add2* subroutine adds two unsigned multi-precision numbers *a* and *b* of *a_size* and *b_size* quadwords respectively. The result is stored in the array pointed to by *s* and the size of the result is returned. This size is either max(*a_size, b_size*) or max(*a_size*, *b_size*)+1 if the result overflowed.

The *mpm_add3* subroutine adds two unsigned multi-precision numbers *a* and *b* of *a_size* and *b_size* quadwords respectively. The result is stored in the array pointed to by *s* of *s_size* quadwords.

Dependencies

See Also

mpm_add_partial on page 124
mpm_sub on page 144

11.3 mpm_add_partial

C Specification

```
#include <mpm_add_partial.h>
inline void _mpm_add_partial(vector unsigned int *s, vector unsigned int *a, vector unsigned int *b,
                             vector unsigned int *c, int size)

#include <libmpm.h>
void mpm_add_partial(vector unsigned int *s, vector unsigned int *a, vector unsigned int *b,
                     vector unsigned int *c, int size)
```

Descriptions

The *mpm_add_partial* subroutine adds two multi-precision numbers of *size* quadwords pointed to by *a* and *b* using a technique in which word carry outs are accumulated in a seperate multi-precision number *c.* The sum is stored in the array pointed to by *s*. The carry array *c* is both an input and an output. All numbers are of *size* quadwords.

This function can be used to significantly improve the performance of accumulating multiple multi-precision numbers. For example, to accumulate 4 mult-precision numbers n1, n2, n3, and n4.

```
vector unsigned int s[size], c[size], n1[size], n2[size], n3[size], n4[size];
for (i=0, i<size; i++) c[size] = (vector unsigned int)(0);
mpm_add_partial(s, n1, n2, c, size);
mpm_add_partial(s, s, n3, c, size);
mpm_add_partial(s, s, n4, c, size);
rotate_left_1word(c, size);
(void)mpm_add(s, s, c);
```

Dependencies

See Also

mpm_add on page 123

11.4 mpm_cmpeq

C Specification

```
#include <mpm_cmpeq.h>
inline unsigned int _mpm_cmpeq(vector unsigned int *a, vector unsigned int *b, int size)

#include <mpm_cmpeq2.h>
inline unsigned int _mpm_cmpeq2(vector unsigned int *a, int a_size, vector unsigned int *b,
                                int b_size)

#include <libmpm.h>
unsigned int mpm_cmpeq(vector unsigned int *a, vector unsigned int *b, int size)

#include <libmpm.h>
unsigned int _mpm_cmpeq2(vector unsigned int *a, int a_size, vector unsigned int *b,
                         int b_size)
```

Descriptions

The *mpm_cmpeq* subroutine compares two multi-precision numbers *a* and *b* of *size* quadwords. If the two numbers are equal then 0xFFFFFFFF is returned; otherwise 0x0 is returned.

The *mpm_cmpeq2* subroutine compares two multi-precision numbers *a* and *b* of *a_size* and *b_size* quadwords respectively. If the two numbers are equal then 0xFFFFFFFF is returned; otherwise 0x0 is returned.

Dependencies

See Also

mpm_cmpge on page 126
mpm_cmpgt on page 127

11.5 mpm_cmpge

C Specification

```
#include <mpm_cmpge.h>
inline unsigned int _mpm_cmpge(vector unsigned int *a, vector unsigned int *b, int size)

#include <mpm_cmpge2.h>
inline unsigned int _mpm_cmpge2(vector unsigned int *a, int a_size vector unsigned int *b,
                                                    int b_size)

#include <libmpm.h>
unsigned int mpm_cmpge(vector unsigned int *a, vector unsigned int *b, int size)

#include <libmpm.h>
unsigned int mpm_cmpge2(vector unsigned int *a, int a_size vector unsigned int *b,
                                                    int b_size)
```

Descriptions

The *mpm_cmpge* subroutine compares two unsigned multi-precision numbers *a* and *b* of *size* quad-words. If the number pointed to by *a* is greater than or equal to the number pointed to by *b* then 0xFFFFFFF is returned; otherwise 0x0 is returned.

The *mpm_cmpge2* subroutine compares two unsigned multi-precision numbers *a* and *b* of *a_size* and *b_size* quadwords respectively. If the number pointed to by *a* is greater than or equal to the number pointed to by *b* then 0xFFFFFFF is returned; otherwise 0x0 is returned.

Dependencies

See Also

mpm_cmpeq on page 125
mpm_cmpgt on page 127

11.6 mpm_cmpgt

C Specification

```
#include <mpm_cmpgt.h>
inline unsigned int _mpm_cmpgt(vector unsigned int *a, vector unsigned int *b, int size)

#include <mpm_cmpgt2.h>
inline unsigned int _mpm_cmpgt2(vector unsigned int *a, int a_size, vector unsigned int *b,
                                                        int b_size)

#include <libmpm.h>
unsigned int mpm_cmpgt(vector unsigned int *a, vector unsigned int *b, int size)

#include <libmpm.h>
unsigned int mpm_cmpgt2(vector unsigned int *a, int a_size, vector unsigned int *b,
                                                        int b_size)
```

Descriptions

The *mpm_cmpgt* subroutine compares two multi-precision numbers *a* and *b* of *size* quadwords. If the number pointed to by *a* is greater than the number pointed to by *b* then 0xFFFFFFF is returned; otherwise 0x0 is returned.

The *mpm_cmpgt2* subroutine compares two multi-precision numbers *a* and *b* of *a_size* and *b_size* quadwords respectively. If the number pointed to by *a* is greater than the number pointed to by *b* then 0xFFFFFFF is returned; otherwise 0x0 is returned.

Dependencies

See Also

mpm_cmpeq on page 125
mpm_cmpge on page 126

11.7 mpm_div

C Specification

```
#include <mpm_div.h>
inline void _mpm_div(vector unsigned int *q, vector unsigned int *r,
                     vector unsigned int *a, int a_size,
                     vector unsigned int *b, int b_size)

#include <mpm_div2.h>
inline void _mpm_div2(vector unsigned int *q, vector unsigned int *a, int a_size,
                     vector unsigned int *b, int b_size)

#include <libmpm.h>
void mpm_div(vector unsigned int *q, vector unsigned int *r,
                     vector unsigned int *a, int a_size,
                     vector unsigned int *b, int b_size)

#include <libmpm.h>
void mpm_div2(vector unsigned int *q, vector unsigned int *a, int a_size,
                     vector unsigned int *b, int b_size)
```

Descriptions

The *mpm_div* subroutine divides the unsigned multi-precision number of *a_size* quadwords pointed to by *a* by the unsigned multi-precision number of *b_size* quadwords pointed to by *b*. The resulting quotient of *a_size* quadwords is returned in *q*, and the remainder of *b_size* quadwords is returned in *r*.

$q = a / b$
$r = a - q * b$

The divisor *b* must be non-zero. An infinite loop may result if *b* is zero. Furthermore, this implementation assumes that all input arrays must be unique and do not overlap except for the dividend *a* and quotient *q* arrrays can be the same.

The *mpm_div2* subroutine is equivalent to *mpm_div* except the remainder is not computed.

Dependencies

See Also

mpm_mod on page 132
mpm_mul on page 138

11.8 mpm_fixed_mod_reduction

C Specification

```
#include <mpm_fixed_mod_reduction.h>
inline void _mpm_fixed_mod_reduction(vector unsigned int *r, const vector unsigned int *a,
                                     const vector unsigned int *m,
                                     const vector unsigned int *u, int n)

#include <libmpm.h>
void mpm_fixed_mod_reduction(vector unsigned int *r, const vector unsigned int *a,
                             const vector unsigned int *m,
                             const vector unsigned int *u, int n)
```

Description

The *mpm_fixed_mod_reduction* subroutine performs a modulus reduction of *a* for the fixed modulus *m* and returns the result in the array *r*.

$$r = a \bmod m$$

The modulus *m* is multi-precision unsigned integer of *n* quadwords and must be non-zero. The input *a* is a multi-precision unsigned integer of 2**n* quadwords. The result, *r*, is *n* quadwords.

This subroutine utilizes an optimization known as Barrett's algorithm to reduce the complexity of computing the modulo operation. The optimization requires the precomputation of the contant *u*. The value *u* is the quotient of $2^{128*2*n}$ divided by m and is n+2 quadwords in length.

The compile-time define MPM_MAX_SIZE controls the maximum supported value *n*. The default value of 32 corresponds to a maximum size of 4096 bits.

Dependencies

mpm_cmpgt on page 127
mpm_sub on page 144

See Also

mpm_mod_exp on page 133
mpm_mod on page 132

11.9 mpm_gcd

C Specification

```
#include <mpm_gcd.h>
inline void _mpm_gcd(vector unsigned int *g, vector unsigned int *a, int a_size,
                                              vector unsigned int *b, int b_size)

#include <libmpm.h>
void mpm_gcd(vector unsigned int *g, vector unsigned int *a, int a_size,
                                              vector unsigned int *b, int b_size)
```

Descriptions

The *mpm_gcd* subroutine computes the greatest common divisor of the two unsigned multi-precision numbers pointed to by *a* and *b* of size *a_size* and *b_size* respectively. A result of *b_size* quadwords is returned into the multi-precision number pointed to by *g.*

The computation of the GCD is commonly computed by the following recusive definition:

$$GCD(a, b) = GCD(b, a \% b)$$

where $a \% b$ is the reaminder of a divided by b (i.e., modulo).

Note: The multi-precision numbers *a* and *b* must be non-zero.

Dependencies

mpm_cmpgt on page 127
mpm_mod on page 132

See Also

mpm_div on page 128

11.10 mpm_madd

C Specification

```
#include <mpm_madd.h>
inline void _mpm_madd(vector unsigned int *d, vector unsigned int *a, int a_size,
                      vector unsigned int *b, int b_size,
                      vector unsigned int *c, int c_size)

#include <libmpm.h>
void mpm_madd(vector unsigned int *d, vector unsigned int *a, int a_size,
              vector unsigned int *b, int b_size,
              vector unsigned int *c, int c_size)
```

Descriptions

The *mpm_madd* subroutine multiples two multi-precision numbers *a* and *b* of size *a_size* and *b_size* quadwords respectively, and adds the multi-precision number *c* of *c_size* quadwords to the resulting product. The final result of *a_size*+*b_size* quadwords is returned to the multi-precision number pointed to by *d*.

$$d = a * b + c$$

Intermediate partial products are accumulated using the technique described in the *mpm_add_partial* subroutine.

Dependencies

See Also

mpm_mul on page 138
mpm_add on page 123
mpm_add_partial on page 124

11.11 mpm_mod

C Specification

```
#include <mpm_mod.h>
inline void _mpm_mod(vector unsigned int *m, vector unsigned int *a, int a_size,
                                              vector unsigned int *b, int b_size)

#include <libmpm.h>
void mpm_mod(vector unsigned int *m, vector unsigned int *a, int a_size,
                                              vector unsigned int *b, int b_size)
```

Descriptions

The *mpm_mod* subroutine computes the modulo of the unsigned multi-precision numbers *a* and *b* of size *a_size* and *b_size* quadword respectively. The result of *b_size* quadwords is returned to the multi-precision number pointed to by *m*.

$$m = a \% b$$

The modulo function is defined to be the remainder of *a* divided by *b*.

For this implementation, the modulo of any number and zero is zero.

Dependencies

mpm_cmpgt on page 127
mpm_sub on page 144

See Also

mpm_div on page 128

11.12 mpm_mod_exp

C Specification

```
#include <mpm_mod_exp.h>
inline void _mpm_mod_exp(vector unsigned int *c, const vector unsigned int *b,
                         const vector unsigned int *e, int e_size,
                         const vector unsigned int *m, int m_size, int k)

#include <mpm_mod_exp2.h>
inline void _mpm_mod_exp2(vector unsigned int *c, const vector unsigned int *b,
                         const vector unsigned int *e, int e_size,
                         const vector unsigned int *m, int m_size, int k,
                         const vector unsigned int *u)

#include <mpm_mod_exp3.h>
inline void _mpm_mod_exp3(vector unsigned int *c, const vector unsigned int *b, int b_size,
                         const vector unsigned int *e, int e_size,
                         const vector unsigned int *m, int m_size,
                         const vector unsigned int *u)

#include <libmpm.h>
void mpm_mod_exp(vector unsigned int *c, const vector unsigned int *b,
                         const vector unsigned int *e, int e_size,
                         const vector unsigned int *m, int m_size, int k)

#include <libmpm.h>
void mpm_mod_exp2(vector unsigned int *c, const vector unsigned int *b,
                         const vector unsigned int *e, int e_size,
                         const vector unsigned int *m, int m_size, int k,
                         const vector unsigned int *u)

#include <libmpm.h>
void mpm_mod_exp3(vector unsigned int *c, const vector unsigned int *b, int b_size,
                         const vector unsigned int *e, int e_size,
                         const vector unsigned int *m, int m_size,
                         const vector unsigned int *u)
```

Description

The *mpm_mod_exp* subroutine is a generic routine that compute the modulus exponentiation function

$$c = b^e \% m$$

where *b*, *e*, and *m* are large multi-precision unsigned integers of *m_size*, *e_size*, and *m_size* quadwords respectively. The result, *c*, is of *m_size* quadwords. The exponent *e* must be non-zero,

The implementation uses a variable size sliding window optimization. The maximum size of the sliding window is specified during compilation by the define MPM_MOD_EXP_MAX_K (defaults to 6). This constants controls the size of the local stack arrays. The parameter *k* specifies the size of the sliding window to be applied and must in the range 1 to MPM_MOD_EXP_MAX_K. The optimal value of *k* is chosen as a

function of the number of bits in the exponent *e.* For large exponents (1024-2048 bits) the optimal value for *k* is 6. For small exponents (4-12 bits), the optimal value for *k* is 2.

The *mpm_mod_exp2* subroutine is equivalent to *mpm_mod_exp* except that the input parameter *u* is provoded by the caller instead of being computed within the modular exponentiation function. The value *u* is the quotient of $2^{128*2*msize}$ divided by m and is *msize*+2 quadwords in length.

The *mpm_mod_exp3* subroutine is equivalent to mpm_mod_exp2 execpt that the base, *b*, is of *bsize* quadwords and the sliding window is fixed size of 6 bits. Note, even though the base can be a different length than the modulus, *m*, *b* must still be less than *m*.

Dependencies

mpm_mul on page 138
mpm_div on page 128
mpm_square on page 143
mpm_fixed_mod_reduction on page 129

See Also

mpm_mont_mod_exp on page 135

11.13 mpm_mont_mod_exp

C Specification

```
#include <mpm_mont_mod_mul.h>
inline void _mpm_mont_mod_exp(vector unsigned int *c, const vector unsigned int *b,
                              const vector unsigned int *e, int esize
                              const vector unsigned int *m, int msize,
                              int k)
#include <mpm_mont_mod_mul.h>
inline void _mpm_mont_mod_exp2(vector unsigned int *c, const vector unsigned int *b,
                              const vector unsigned int *e, int esize
                              const vector unsigned int *m, int msize,
                              int k, vector unsigned int p,
                              const vector unsigned int *a,
                              const vector unsigned int *u)
#include <mpm_mont_mod_mul.h>
inline void _mpm_mont_mod_exp3(vector unsigned int *c, const vector unsigned int *b, int bsize,
                              const vector unsigned int *e, int esize
                              const vector unsigned int *m, int msize)

#include <libmpm.h>
void mpm_mont_mod_exp(vector unsigned int *c, const vector unsigned int *b,
                              const vector unsigned int *e, int esize
                              const vector unsigned int *m, int msize,
                              int k)

#include <libmpm.h>
void mpm_mont_mod_exp2(vector unsigned int *c, const vector unsigned int *b,
                              const vector unsigned int *e, int esize
                              const vector unsigned int *m, int msize,
                              int k, vector unsigned int p,
                              const vector unsigned int *a,
                              const vector unsigned int *u)
#include <libmpm.h>
void mpm_mont_mod_exp3(vector unsigned int *c, const vector unsigned int *b, int bsize,
                              const vector unsigned int *e, int esize
                              const vector unsigned int *m, int msize)
```

Descriptions

The *mpm_mont_mod_exp* subroutine is a generic routine that uses Montgomerymodulo multiplication to compute the modulus exponentiation function:

$$c = b^e \% m$$

where *b*, *e*, and *m* are large multi-precision unsigned integers of *m_size*, *e_size*, and *m_size* quadwords respectively. The result, *c*, is of *m_size* quadwords.

The implementation uses a variable size sliding window optimization. The maximum size of the sliding window is specified during compilation by the define MPM_MOD_EXP_MAX_K (defaults to 6). This con-

stants controls the size of the local stack arrays. The parameter *k* specifies the size of the sliding window to be applied and must in the range 1 to MPM_MOD_EXP_MAX_K. The optimal value of *k* is chosen as a function of the number of bits in the exponent *e.* For large exponents (1024-2048 bits) the optimal value for *k* is 6. For small exponents (4-12 bits), the optimal value for *k* is 2.

The *mpm_mont_mod_exp2* subroutine is equivalent to *mpm_mont_mod_exp* except that several parameters must be pre-computed and passed by the caller. These parameters include:

p : quadword invsere factor. Is in the range 1 to 2^{128} - 1 and equals 2^{128} - g where (g * (*m* % 2^{128})) % 2^{128} = 1.

a: pre-computed multi-precision number of *msize* quadwords. Must equal $2^{128*msize}$ % *m.*

u: pre-computed multi-precision number of *msize* quadwords. Must equal $2^{2*128*msize}$ % *m.*

The *mpm_mont_mod_exp3* subroutine is equivalent to mpm_mont_mod_exp execpt the size of *b* is specified by the *bsize* parameter and the sliding window size is constant and equals MPM_MOD_EXP_MAX_K

Dependencies

mpm_mod on page 132
mpm_mul_inv on page 139
mpm_mont_mod_mul on page 137

See Also

mpm_mod_exp on page 133

11.14 mpm_mont_mod_mul

C Specification

```
#include <mpm_mont_mod_mul.h>
inline void _mpm_mont_mod_mul(vector unsigned int *c, const vector unsigned int *a,
                              const vector unsigned int *b,
                              const vector unsigned int *m, int size,
                              vector unsigned int p)

#include <libmpm.h>
void mpm_mont_mod_mul(vector unsigned int *c, const vector unsigned int *a,
                      const vector unsigned int *b,
                      const vector unsigned int *m, int size,
                      vector unsigned int p)
```

Descriptions

The *mpm_mont_mod_mul* subroutine performs Montgomery modular multiplication of multi-precision numbers *a* and *b* for the modulus *m*. The result of *size* quadwords is returned in the array *c* and is equal to:

$c = (a * b * y) \% m$

where y is the product inverse factor such that $0 < y < m$. That is, $(y * (2^{128*size} \% m)) \% m = 1$

The multi-precision inputs *a* and *b* are multi-precision numbers of *size* quadwords in the range 0 to *m*-1. The multi-precision modulus, *m*, is of *size* quadwords and must be odd and non-zero. The quadword inverse factor, *p*, is in the range 1 to 2^{128} - 1 and equals 2^{128} - g where (g * (*m* % 2^{128})) % 2^{128} = 1.

Note: The multi-precision numbers *m* and *c* must be unique memory arrays.

Dependencies

mpm_sub on page 144

See Also

mpm_mod on page 132
mpm_mont_mod_exp on page 135

11.15 mpm_mul

C Specification

```
#include <mpm_mul.h>
inline void _mpm_mul(vector unsigned int *p, vector unsigned int *a, int a_size,
                                              vector unsigned int *b, int b_size)

#include <libmpm.h>
void mpm_mul(vector unsigned int *p, vector unsigned int *a, int a_size,
                                              vector unsigned int *b, int b_size)
```

Descriptions

The *mpm_mul* subroutine multiples two multi-precision numbers *a* and *b* of size *a_size* and *b_size* quadwords respectively. The resulting product of *a_size+b_size* quadwords is returned to the multi-precision number pointed *p*.

$$p = a * b$$

Intermediate partial products are accumulated using the technique described in the *mpm_add_partial* subroutine.

Dependencies

See Also

mpm_madd on page 131
mpm_add_partial on page 124

11.16 mpm_mul_inv

C Specification

```
#include <mpm_mul_inv.h>
inline int _mpm_mul_inv(vector unsigned int *mi, vector unsigned int *a,
                                                 vector unsigned int *b, int size)
#include <mpm_mul_inv2.h>
inline int _mpm_mul_inv2(vector unsigned int *mi, vector unsigned int *a, int a_size
                                                 vector unsigned int *b, int b_size)
#include <mpm_mul_inv3.h>
inline int _mpm_mul_inv3(vector unsigned int *mi, vector unsigned int *a, int a_size
                                                 vector unsigned int *b, int b_size)

#include <libmpm.h>
int mpm_mul_inv(vector unsigned int *mi, vector unsigned int *a, vector unsigned int *b,
                                                 int size)
#include <libmpm.h>
int mpm_mul_inv2(vector unsigned int *mi, vector unsigned int *a, int a_size,
                                                 vector unsigned int *b, int b_size)
#include <libmpm.h>
int mpm_mul_inv3(vector unsigned int *mi, vector unsigned int *a, int a_size,
                                                 vector unsigned int *b, int b_size)
```

Descriptions

The *mpm_mul_inv*, *mpm_mul_inv2*, and *mpm_mul_inv3* subroutines compute the multiplicative inverse (*mi*) of the multi-precision number *b* with respect to *a.* That is to say, the multiplicative inverse is *mi* that satisfies the equation:

$$(mi * b) \% a = 1$$

For the *mpm_mul_inv* subroutine, the size of *a*, *b*, and *mi* is of *size* quadwords. For the *mpm_mul_inv2* and *mpm_mul_inv3* subroutines, *a* and *mi* is of *a_size* quadwords and *b* is of *b_size quadwords.*

Subroutine	Algorithm	Characteristics
mpm_mul_inv	Shift and accumulate	Efficient for conditions in which *a* and *b* are similarly sized. Small code size.
mpm_mul_inv2	Divide and multiply	Efficient for conditions in which *a* and *b* significantly differ in size. Moderate code size.
mpm_mul_inv3	Hybrid algorithm	Hybrid solution that leverages upon the implementation features of each of the other algorithms. Large code size.

A value of 0 is returned if the multilicative inverse does not exist. Otherwise, 1 is returned and the multiplicative inverse is return in the array pointed to by *mi* where 0 < *mi* < *a*.

These functions require that *b* be non-zero and less than *a*.

Dependencies

mpm_add on page 123
mpm_cmpge on page 126
mpm_cmpgt on page 127
mpm_div on page 128
mpm_mod on page 132
mpm_mul on page 138
mpm_sizeof on page 142
mpm_sub on page 144

See Also

11.17 mpm_neg

C Specification

```
#include <mpm_neg.h>
inline void _mpm_neg(vector unsigned int *n, vector unsigned int *a, int size)

#include <libmpm.h>
void mpm_neg(vector unsigned int *n, vector unsigned int *a, int size)
```

Descriptions

The *mpm_neg* subroutine negates the multi-precision number of *size* quadwords pointed to by *a* and returns the result to the multi-precision number pointed to by *n*.

$n = -a$

Dependencies

See Also

mpm_abs on page 122

11.18 mpm_sizeof

C Specification

```
#include <mpm_sizeof.h>
inline int _mpm_sizeof(vector unsigned int *a, int size)

#include <libmpm.h>
int mpm_sizeof(vector unsigned int *a, int size)
```

Descriptions

The *mpm_sizeof* subroutine computes the “true” size of the unsigned multi-precision number of *size* quadwords pointed to by *a*. The “true” size the highest numbered quadword that contain a non-zero value. A multi-precision number of zero returns a sizeof equal to 0.

Dependencies

See Also

11.19 mpm_square

C Specification

```
#include <mpm_square.h>
inline void _mpm_square(vector unsigned int *s, vector unsigned int *a, int size)

#include <libmpm.h>
void mpm_square(vector unsigned int *s, vector unsigned int *a, int size)
```

Descriptions

The *mpm_square* subroutine squares the *a* of size *size* quadwords and returns the multi-precision result of 2**size* quadwords in *s.* This subroutine is a specialized variant of *mpm_mul* which takes advantage of the fact that many of the porduct terms of a squared number are repeated.

Intermediate partial products are accumulated using the technique described in the *mpm_add_partial* subroutine.

Dependencies

See Also

mpm_mul on page 138
mpm_add_partial on page 124

11.20 mpm_sub

C Specification

```
#include <mpm_sub.h>
inline vector unsigned int _mpm_sub(vector unsigned int *s, vector unsigned int *a,
                                    vector unsigned int *b, int size)
#include <mpm_sub2.h>
inline void _mpm_sub2(vector unsigned int *s, vector unsigned int *a, int a_size,
                                    vector unsigned int *b, int b_size)

#include <libmpm.h>
vector unsigned int mpm_sub(vector unsigned int *s, vector unsigned int *a,
                                    vector unsigned int *b, int size)
#include <libmpm.h>
void mpm_sub2(vector unsigned int *s, vector unsigned int *a, int a_size,
                                    vector unsigned int *b, int b_size)
```

Descriptions

The *mpm_sub* subroutine subtracts the multi-precision number *b* from the multi-precision number *a.* The result is stored in the memory pointed to by *s*. The numbers *a*, *b*, and *s* are all *size* quadwords in length.

$$s = a - b$$

mpm_sub also returns a borrow out vector. A borrow out of (0,0,0,1) indicates that no borrow out occurred. A borrow out of (0,0,0,0) indicates a borrow resulted.

The *mpm_sub2* subroutine subtracts the multi-precision number *b* of *b_size* quadwords from the multi-precision number *a* of *a_size* quadwords*.* The result is stored in the memory pointed to by *s* of *a_size* quadwords. *a* must be larger than *b,* however, *a_size* can be smaller than *b_size.*

Dependencies

See Also

mpm_add on page 123

11.21 mpm_swap_endian

C Specification

```
#include <mpm_swap_endian.h>
inline void _mpm_swap_endian(vector unsigned int *a, int size)

#include <libmpm.h>
void mpm_swap_endian(vector unsigned int *a, int size)
```

Descriptions

The *mpm_swap_endian* subroutine swap the endian-ness (ie. byte ordering) of the multi-precision number of *size* quadwords pointed to by *a*. This subroutine converts little endian numbers to big endian numbers and vice versa.

Dependencies

See Also

12. Sync Library

The sync library provides simple seveal general purpose synchronization constructs for both the PPE and SPE. These constructs are all based upon the Cell Broadband Engine Architecture's extended *load-with-reservation* and *store-conditional* functionality. On the PPE, these functions are provided via the *lawrx*/*ldarx* and *stwcx* /*stdcx* instructions. On the SPE, these functions are provided via the *getllar* and *putllar* MFC (Memory Flow Controller) commands.

The sync library provides five sub-classes of synchronization primitives - atomic operations, mutexes, condition variables, completion variables, reader/writer locks. The function closely match those found in current traditional operating systems.

This library is supported on the PPE (32-bit and 64-bit) and SPE. In addition, PPE and SPE versions of this library are also provided that include Performance Debugging Tool (PDT) trace events for the synchronization functions. The trace enabled versions of the sync library are provided in a trace subdirectory.

Name(s)

libsync.a

Header File(s)

<libsync.h>

12.1 Atomic Operations

The synchronization library supports a large number of atomic operations on naturally aligned, 32-bit variables. These variables reside in the 64-bit effective addres space as specified by a *atomic_ea_t* data type.

12.1.1 atomic_add

C Specification

```
#include <atomic_add.h>
inline void _atomic_add(int a, atomic_ea_t ea)

#include <atomic_add_return.h>
inline int _atomic_add_return(int a, atomic_ea_t ea)

#include <libsync.h>
void atomic_add(int a, atomic_ea_t ea)

#include <libsync.h>
int atomic_add_return(int a, atomic_ea_t ea)
```

Descriptions

The *atomic_add* and *atomic_add_return* subroutines atomically adds the integer *a* to the 32-bit integer pointed to by the effective address *ea*. The *atomic_add_return* also returns the pre-added integer pointed to by *ea.*

To ensure correct operation, the word addressed by *ea* must be word (32-bit) aligned.

Dependencies

See Also

atomic_dec on page 149
atomic_inc on page 150
atomic_read on page 151
atomic_set on page 152
atomic_sub on page 153

12.1.2 atomic_dec

C Specification

```
#include <atomic_dec.h>
inline void _atomic_dec(atomic_ea_t ea)

#include <atomic_dec_return.h>
inline int _atomic_dec_return(atomic_ea_t ea)

#include <atomic_dec_and_test.h>
inline int _atomic_dec_and_test(atomic_ea_t ea)

#include <atomic_dec_if_positive.h>
inline int _atomic_dec_if_positive.h(atomic_ea_t ea);

#include <libsync.h>
void atomic_dec(atomic_ea_t ea)

#include <libsync.h>
int atomic_dec_return(atomic_ea_t ea)

#include <libsync.h>
int atomic_dec_and_test(atomic_ea_t ea)

#include <libsync.h>
int atomic_dec_if_positive.h(atomic_ea_t ea);
```

Descriptions

The *atomic_dec*, *atomic_dec_return*, and *atomic_dec_and_test* subroutines atomically decrement (subtract 1 from) the 32-bit integer pointed to the effective address *ea*. The *atomic_dec_return* subroutine also returns the pre-decremented integer pointed to by *ea.* The *atomic_dec_and_test* subroutine also returns the comparison of the pre-decremented integer pointed to by *ea* with 0, returning 0 if the pre-decremented integer is non-zero, and 1 if the pre-decremented integer is zero.

The *atomic_dec_if_positive* subroutine atomically tests the integer pointed to by *ea* and decrements it if it is positive (greater than or equal to zero). The integer at *ea* minus 1 is returned, regardless of its value.

To ensure correct operation, the word addressed by *ea* must be word (32-bit) aligned.

Dependencies

See Also

atomic_add on page 148
atomic_inc on page 150
atomic_read on page 151
atomic_set on page 152
atomic_sub on page 153

12.1.3 atomic_inc

C Specification

```
#include <atomic_inc.h>
inline void _atomic_inc(atomic_ea_t ea)

#include <atomic_inc_return.h>
inline int _atomic_inc_return(atomic_ea_t ea)

#include <libsync.h>
void atomic_inc(atomic_ea_t ea)

#include <libsync.h>
int atomic_inc_return(atomic_ea_t ea)
```

Descriptions

The *atomic_inc* and *atomic_inc_return* subroutines atomically increments the 32-bit integer pointed to by the effective address *ea*. The *atomic_inc_return* also returns the pre-incremented integer pointed to by *ea.* This routine implements the *fetch and increment* primitive described in Book I of the PowerPC User Instruction Set Architecture.

To ensure correct operation, the word addressed by *ea* must be word (32-bit) aligned.

Dependencies

See Also

atomic_add on page 148
atomic_dec on page 149
atomic_read on page 151
atomic_set on page 152
atomic_sub on page 153

12.1.4 atomic_read

C Specification

```
#include <atomic_read.h>
inline int _atomic_read(atomic_ea_t ea)

#include <libsync.h>
int atomic_read(atomic_ea_t ea)
```

Descriptions

The *atomic_read* subroutine atomically reads the 32-bit integer pointed to the effective address *ea*. On the PPE, an atomic read is simply a volatile load.

To ensure correct operation, the word addressed by *ea* must be word (32-bit) aligned.

Dependencies

See Also

atomic_add on page 148
atomic_dec on page 149
atomic_inc on page 150
atomic_set on page 152
atomic_sub on page 153

12.1.5 atomic_set

C Specification

```
#include <atomic_set.h>
inline void _atomic_set(atomic_ea_t ea, int val)

#include <libsync.h>
void atomic_set(atomic_ea_t ea, int val)
```

Descriptions

The *atomic_set* subroutine atomically writes the integer specified by *val* to the 32-bit integer pointed to by the effective address *ea*. This routine implements the *fetch and store* primitive described in Book I of the PowerPC User Instruction Set Architecture.

To ensure correct operation, the word addressed by *ea* must be word (32-bit) aligned.

Dependencies

See Also

atomic_add on page 148
atomic_dec on page 149
atomic_inc on page 150
atomic_read on page 151
atomic_sub on page 153

12.1.6 atomic_sub

C Specification

```
#include <atomic_sub.h>
inline void _atomic_sub(int a, atomic_ea_t ea)

#include <atomic_sub_return.h>
inline int _atomic_sub_return(int a, atomic_ea_t ea)

#include <atomic_sub_and_test.h>
inline int _atomic_sub_and_test(int a, atomic_ea_t ea)

#include <libsync.h>
void atomic_sub(int a, atomic_ea_t ea)

#include <libsync.h>
int atomic_sub_return(int a, atomic_ea_t ea)

#include <libsync.h>
int atomic_sub_and_test(int a, atomic_ea_t ea)
```

Descriptions

The *atomic_sub, atomic_sub_return*, and *atomic_sub_and_test* subroutines atomically subtracts the integer *a* from the 32-bit integer pointed to the effective address *ea*. The *atomic_sub_return* also returns the pre-subtracted integer pointed to by *ea.* The *atomic_sub_and_test* subroutine also returns the comparison of the pre-subtracted integer pointed to by *ea* with 0, returning 0 if the pre-decremented integer is non-zero, and 1 if the pre-decremented integer is zero.

To ensure correct operation, the word addressed by *ea* must be word (32-bit) aligned.

Dependencies

See Also

atomic_add on page 148
atomic_dec on page 149
atomic_inc on page 150
atomic_read on page 151
atomic_set on page 152

12.2 Mutexes

The following set of routines operate on mutex (**mut**ual **ex**clusion) objects and are used to ensure exclusivity. Mutex objects are specified by a 64-bit effective address of type mutex_ea_t, which points to a naturally aligned 32-bit integer.

12.2.1 mutex_init

C Specification

```
#include <mutex_init.h>
inline void _mutex_init(mutex_ea_t lock)

#include <libsync.h>
void mutex_init(mutex_ea_t lock)
```

Descriptions

The *mutex_init* subroutine initializes the *lock* mutex object by setting its value to 0 (i.e., unlocked).

To ensure correct operation, the word addressed by *lock* must be word (32-bit) aligned.

Dependencies

See Also

mutex_lock on page 155
mutex_trylock on page 156
mutex_unlock on page 157

12.2.2 mutex_lock

C Specification

```
#include <mutex_lock.h>
inline void _mutex_lock(mutex_ea_t lock)

#include <libsync.h>
void mutex_lock(mutex_ea_t lock)
```

Descriptions

The *mutex_lock* subroutine acquires a lock by waiting (spinning) for the mutex object, specified by *lock*, to become zero, then atomically writing a 1 to the lock variable.

To ensure correct operation, the word addressed by *lock* must be word (32-bit) aligned.

Dependencies

See Also

mutex_init on page 154
mutex_trylock on page 156
mutex_unlock on page 157

12.2.3 mutex_trylock

C Specification

```
#include <mutex_trylock.h>
inline int _mutex_trylock(mutex_ea_t lock)

#include <libsync.h>
int mutex_trylock(mutex_ea_t lock)
```

Descriptions

The *mutex_trylock* subroutine tries to acquire a lock by checking the mutex object, specified by *lock*. If the lock variable is set, then 0 is returned and lock is not acquired. Otherwise, the lock is acquired and 1 is returned.

This subroutine should not be called from a tight loop.

To ensure correct operation, the word addressed by *lock* must be word (32-bit) aligned.

Dependencies

See Also

mutex_init on page 154
mutex_lock on page 155
mutex_unlock on page 157

12.2.4 mutex_unlock

C Specification

```
#include <mutex_unlock.h>
inline void _mutex_unlock(mutex_ea_t lock)

#include <libsync.h>
void mutex_unlock(mutex_ea_t lock)
```

Descriptions

The *mutex_unlock* subroutine releases the mutex lock specified by the *lock* parameter.

To ensure correct operation, the word addressed by *lock* must be word (32-bit) aligned.

Dependencies

See Also

mutex_init on page 154
mutex_lock on page 155
mutex_trylock on page 156

12.3 Conditional Variables

The following routines operate on condition variables. There primary operations on condition variables are *wait* and *signal*. When a thread executes a *wait* call on a condition variable, its is suspended waiting on that condition variable. Its execution is not resumed until another thread signals (or broadcasts) the condition variable.

12.3.1 cond_broadcast

C Specification

```
#include <cond_broadcast.h>
inline void _cond_broadcast(cond_ea_t cond)

#include <libsync.h>
void cond_broadcast(cond_ea_t cond)
```

Descriptions

The *cond_broadcast* subroutine is used to unblock all threads waiting on the conditional variable specified by *cond*. To unblock a single thread, *cond_signal* should be used.

To ensure correct operation, the word addressed by *cond* must be word (32-bit) aligned.

Dependencies

See Also

cond_init on page 159
cond_signal on page 160
cond_wait on page 161

12.3.2 cond_init

C Specification

```
#include <cond_init.h>
inline void _cond_init(cond_ea_t cond)

#include <libsync.h>
void cond_init(cond_ea_t cond)
```

Descriptions

The *cond_init* subroutine initializes the condition variable specified by *cond*. The condition variable is initialized to 0.

To ensure correct operation, only one thread (PPE or SPE) should initialize the condition variable. In addition the word addressed by *cond* must be word (32-bit) aligned.

Dependencies

See Also

cond_broadcast on page 158
cond_signal on page 160
cond_wait on page 161

12.3.3 cond_signal

C Specification

```
#include <cond_signal.h>
inline void _cond_signal(cond_ea_t cond)

#include <libsync.h>
void cond_signal(cond_ea_t cond)
```

Descriptions

The *cond_signal* subroutine is used to unblock a single thread waiting on the conditional variable specified by *cond*. To unblock a all threads, *cond_broadcast* should be used.

To ensure correct operation, the word addressed by *cond* must be word (32-bit) aligned.

Dependencies

See Also

cond_broadcast on page 158
cond_init on page 159
cond_wait on page 161

12.3.4 cond_wait

C Specification

```
#include <cond_wait.h>
inline void _cond_wait(cond_ea_t cond, mutex_ea_t lock)

#include <libsync.h>
void cond_wait(cond_ea_t cond, mutex_ea_t lock)
```

Descriptions

The *cond_wait* subroutine atomically releases the mutex specified by *lock* and causes the calling thread to block on the condition variable *cond.* The thread may be unblocked by another thread calling *cond_broadcast* or *cond_signal.*

To ensure correct operation, the word addressed by *cond* must be word (32-bit) aligned.

Dependencies

See Also

cond_broadcast on page 158
cond_init on page 159
cond_signal on page 160

12.4 Completion Variables

12.4.1 complete

C Specification

```
#include <complete.h>
inline void _complete(completion_ea_t comp)

#include <libsync.h>
void complete(completion_ea_t comp)
```

Descriptions

The *complete* subroutine is used to notify all threads waiting on the completion variable that the completion is true by atomically storing 1 to *comp*.

To ensure correct operation, the word addressed by *comp* must be word (32-bit) aligned.

Dependencies

See Also

complete_all on page 163
init_completion on page 164
wait_for_completion on page 165

12.4.2 complete_all

C Specification

```
#include <complete_all.h>
inline void _complete_all(completion_ea_t comp)

#include <libsync.h>
void complete_all(completion_ea_t comp)
```

Descriptions

The *complete_all* subroutine is used to notify all threads waiting on the completion variable that the completion is true by atomically storing 1 to *comp*.

To ensure correct operation, the word addressed by *comp* must be word (32-bit) aligned.

Dependencies

See Also

complete on page 162
init_completion on page 164
wait_for_completion on page 165

12.4.3 init_completion

C Specification

```
#include <init_completion.h>
inline void _init_completion(completion_ea_t comp)

#include <libsync.h>
void init_completion(completion_ea_t comp)
```

Descriptions

The *init_completion* subroutine initializes the completion variable specified by *comp*. The completion variable is initialized to 0.

To ensure correct operation, the word addressed by *comp* must be word (32-bit) aligned.

Dependencies

See Also

complete on page 162
complete_all on page 163
wait_for_completion on page 165

12.4.4 wait_for_completion

C Specification

```
#include <wait_for_completion.h>
inline void _wait_for_completion(completion_ea_t comp)

#include <libsync.h>
void wait_for_completion(completion_ea_t comp)
```

Descriptions

The *wait_for_completion* subroutine cause the current running thread to wait until another thread or device that a completion is true (not zero).

This function should not be called if SPE asynchronous interrupts are enabled.

To ensure correct operation, the word addressed by *comp* must be word (32-bit) aligned.

Dependencies

See Also

complete on page 162
complete_all on page 163
init_completion on page 164

12.5 Reader/Writer Locks

12.5.1 read_lock

C Specification(SPU only)

```
#include <read_lock.h>
inline void _read_lock(eaddr_t lock)

#include <libsync.h>
void read_lock(eaddr_t lock)
```

Descriptions

The *read_lock* subroutine acquires the reader lock specified by the effective address of the reader/writer lock variable, *lock.* A reader lock is a non-exclusive mutex which allow multiply simultaneous readers.

To ensure correct operation, the word addressed by *lock* must be word (32-bit) aligned.

Dependencies

See Also

read_trylock on page 167
read_unlock on page 168
write_lock on page 169

12.5.2 read_trylock

C Specification(SPU only)

```
#include <read_trylock.h>
inline int _read_trylock(eaddr_t lock)

#include <libsync.h>
int read_trylock(eaddr_t lock)
```

Descriptions

The *read_trylock* subroutine attempts to acquire a reader lock specified by the effective address of the reader/writer lock variable, *lock.* A reader lock is a non-exclusive mutex which allow multiply simultaneous readers. If the reader lock is acquired, 1 is returned. Otherwise, 0 is returned and the reader lock is not acquired.

To ensure correct operation, the word addressed by *lock* must be word (32-bit) aligned.

Dependencies

See Also

read_lock on page 166
read_unlock on page 168
write_trylock on page 170

12.5.3 read_unlock

C Specification(SPU only)

```
#include <read_unlock.h>
inline void _read_unlock(eaddr_t lock)

#include <libsync.h>
void read_unlock(eaddr_t lock)
```

Descriptions

The *read_unlock* subroutine unlocks the reader lock specified by the effective address of the reader/writer lock variable, *lock.* A reader lock is a non-exclusive mutex which allow multiply simultaneous readers.

To ensure correct operation, the word addressed by *lock* must be word (32-bit) aligned.

Dependencies

See Also

read_lock on page 166
read_trylock on page 167
write_unlock on page 171

12.5.4 write_lock

C Specification(SPU only)

```
#include <write_lock.h>
inline void _write_lock(eaddr_t lock)

#include <libsync.h>
void write_lock(eaddr_t lock)
```

Descriptions

The *write_lock* subroutine acquires the writer lock specified by the effective address of the reader/writer lock variable, *lock.* A writer lock is a exclusive mutex which allows a single writer.

To ensure correct operation, the word addressed by *lock* must be word (32-bit) aligned.

Dependencies

See Also

read_lock on page 166
write_trylock on page 170
write_unlock on page 171

12.5.5 write_trylock

C Specification(SPU only)

```
#include <write_trylock.h>
inline int _write_trylock(eaddr_t lock)

#include <libsync.h>
int write_trylock(eaddr_t lock)
```

Descriptions

The *write_trylock* subroutine attempts to acquire a writer lock specified by the effective address of the reader/writer lock variable, *lock.* A writer lock is a exclusive mutex which a single writer. If the writer lock is acquired, 1 is returned. Otherwise, 0 is returned and the writer lock is not acquired.

To ensure correct operation, the word addressed by *lock* must be word (32-bit) aligned.

Dependencies

See Also

read_trylock on page 167
write_lock on page 169
write_unlock on page 171

12.5.6 write_unlock

C Specification(SPU only)

```
#include <write_unlock.h>
inline void _write_unlock(eaddr_t lock)

#include <libsync.h>
void write_unlock(eaddr_t lock)
```

Descriptions

The *write_unlock* subroutine unlocks the writer lock specified by the effective address of the reader/writer lock variable, *lock.* A writer lock is a exclusive mutex which allow a single writer.

To ensure correct operation, the word addressed by *lock* must be word (32-bit) aligned and a writer lock must be active.

Dependencies

See Also

read_unlock on page 168
write_lock on page 169
write_trylock on page 170

13. Vector Library

The vector library consists of a set of general purpose routines that operate on vectors. This library is supported on both the PPE and SPE.

Name(s)

libvector.a

Header File(s)

<libvector.h>

13.1 clipcode_ndc

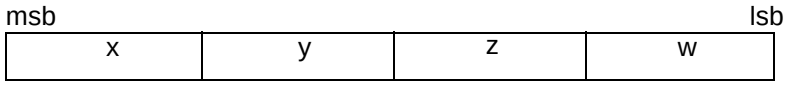
C Specification

```
#include <clipcode_ndc.h>
inline unsigned int _clipcode_ndc(vector float v)

#include <clipcode_ndc_v.h>
inline vector unsigned int _clipcode_ndc_v(vector float x, vector float y, vector float z,
                                           vector float w)

#include <libvector.h>
unsigned int clipcode_ndc(vector float v)

#include <libvector.h>
vector unsigned int _clipcode_ndc_v(vector float x, vector float y, vector float z,
                                    vector float w)
```

Descriptions

The *clipcode_ndc* subroutine generates a clipcode for the **n**ormalized homogeneous **d**evice **c**oordinate vertex specified by *v*. The ndc coordinate is packed into a 128-bit floating-point vector as follows:

Figure 13-1. NDC Packaging (128-Bit Floating-Point Vector)



A clipcode is a set of bit flags indicating if the vertex is outside the halfspaces defined by the -1.0 to 1.0 volume. Defines for each of the 6 bit-flags are defined in libvector.h.

The clipcode is computed as follows:

```
clipcode = 0;
if (v.x < -v.w) clipcode |= CLIP_CODE_LEFT;
if (v.x > v.w) clipcode |= CLIP_CODE_RIGHT;
if (v.y < -v.w) clipcode |= CLIP_CODE_BOTTOM;
if (v.y > v.w) clipcode |= CLIP_CODE_TOP;
if (v.z < -v.w) clipcode |= CLIP_CODE_NEAR;
if (v.z > v.w) clipcode |= CLIP_CODE_FAR;
```

The *clipcode_ndc_v* subroutine generates a vector of 4 clipcodes for 4 vertices specified in parallel array format by the parameters *x*, *y*, *z*, and *w*.

Dependencies

See Also

clip_ray on page 175

13.2 clip_ray

C Specification

```
#include <clip_ray.h>
inline vector float _clip_ray(vector float v1, vector float v2, vector float plane)

#include <libvector.h>
vector float clip_ray(vector float v1, vector float v2, vector float plane)
```

Descriptions

The *clip_ray* subroutine computes the linear interpolation factor for the ray passing through vertices *v1* and *v2* intersecting the plane specified by the parameter *plane*. Input vertices, *v1* and *v2*, are homogeneous 3-D coodinates packed in a 128-bit floating-point vector. The plane is also defined by a 4-component 128-bit floating-point vector satisfying the equation:

$$plane.x * x + plane.y * y + plane.z * z + plane.w * w = 0$$

The output is a floating-point scalar describing tthe position along the ray in which the ray intersects the plane. A value of 0.0 corresponds to the ray intersecting at *v1*. A value of 1.0 corresponds to the ray intersecting at *v2*. The resulting scalar is replicated across all components of a 4-component floating-point vector and is suitable for computing the intersecting vectex using a lerp_vec (linear interpolation) subroutine.

Correct results are produced only if the ray is uniquely defined (i.e., *v1* != *v2*) and that it intersects the plane.

Dependencies

divf4 in SIMD math library

See Also

lerp_vec on page 185

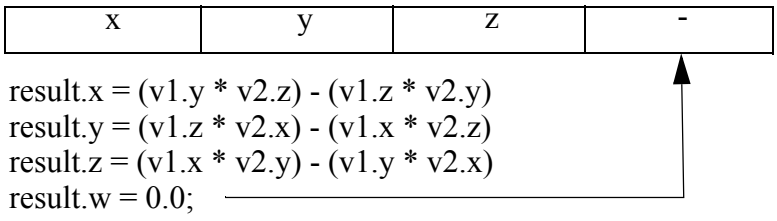
13.3 cross_product

C Specification

```
#include <cross_product3.h>
inline vector float _cross_product3(vector float v1, vector float v2)

#include <cross_product3_v.h>
inline void _cross_product3_v(vector float *xOut, vector float *yOut, vector float *zOut,
                              vector float x1, vector float y1, vector float z1,
                              vector float x2, vector float y2, vector float z2)

#include <cross_product4.h>
inline vector float _cross_product4(vector float v1, vector float v2)

#include <libvector.h>
vector float cross_product3(vector float v1, vector float v2)

#include <libvector.h>
vector float cross_products3_v(vector float *xOut, vector float *yOut, vector float *zOut,
                              vector float x1, vector float y1, vector float z1,
                              vector float x2, vector float y2, vector z2)
```

Descriptions

The *cross_product3* subroutine computes the cross products of two 3-component input vector - *v1* cross *v2*. The 3-component inputs and outputs are packed into 128-bit, 4-component floating point vectors. The 4th input components are not used, however, computation is performed in such a way that the 4th component of the result is 0.0.



The *cross_product3_v* subroutine simultaneously computes 4 cross products of two 3-component input vectors. The input (*x1*, *y1*, *z1*, *x2*, *y2*, *z2*) and outputs (*xOut*, *yOut*, *zOut*) are specified as parallel arrays (i.e., each component of the 4 input vectors resides in a single floating-point vector).

```
*xOut = (y1 * z2) - (z1 *y2)
*yOut = (z1 * x2) - (x1 * z2)
*zOut = (x1 * y2) - (y1 * x2)
```

The *cross_product4* subroutine computes the cross product of two 4-component vectors - v1 and v2. The first three components (x, y, z) are computed as a traditional 3-D cross producet (see cross_product3). The 4th component, w, is the scalar product of the two input's 4th components.

```
result.w = v1.w * v2.w
```

Dependencies

See Also

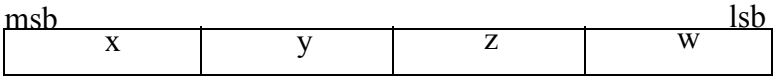
13.4 dot_product

C Specification

```
#include <dot_product3.h>
inline float _dot_product3(vector float v1, vector float v2)

#include <dot_product3_v.h>
inline vector float _dot_product3_v(vector float x1, vector float y1, vector float z1,
                                    vector float x2, vector float y2, vector float z2)

#include <dot_product4.h>
inline float _dot_product4(vector float v1, vector float v2)

#include <dot_product4_v.h>
inline vector float _dot_product4_v(vector float x1, vector float y1, vector float z1,
                                    vector float w1,vector float x2, vector float y2,
                                    vector float z2, vector float w2)


#include <libvector.h>
float dot_product3(vector float v1, vector float v2)

#include <libvector.h>
vector float dot_product3_v(vector float x1, vector float y1, vector float z1,
                                    vector float x2, vector float y2, vector float z2)

#include <libvector.h>
float dot_product4(vector float v1, vector float v2)

#include <libvector.h>
vector float dot_product4_v(vector float x1, vector float y1, vector float z1,
                                    vector float w2, vector float x2, vector float y2,
                                    vector float z2, vector float w2)
```

Descriptions

The *dot_product3* subroutine computes the dot product of two input vectors - *v1* dot *v2*. The inputs, *v1* and *v2*, are 4 component vectors in which only the first 3 (most significant) components contribute to the dot product computation.



$$\text{dot\_product3}(v1, v2) = v1.x \times v2.x + v1.y \times v2.y + v1.z \times v2.z$$

The *dot_product3_v* subroutine computes 4 simultaneous dot products of the two input SIMD vectors. The input vectors are specified in parallel array format by input parameters *x1*, *y1*, *z1*, and *x2*, *y2*, *z2*.

The *dot_product4* subroutine computes the dot product of the two input vectors, *v1* and *v2*, over all four components. This form of dot product is useful when computing the angular distance between unit quate-

rions.

$$\text{dot\_product4}(v1, v2) = v1.x \times v2.x + v1.y \times v2.y + v1.z \times v2.z + v1.w \times v1.w$$

The *dot_product4_v* subroutine computes 4 simultaneous dot products over all 4 components of the two input SIMD vectors. The input vectors are specified in parallel array format by input parameters *x1*, *y1*, *z1*, w1, and *x2*, *y2*, *z2, w2*.

Dependencies

See Also

13.5 intersect_ray_triangle

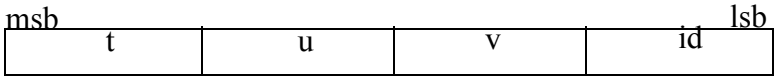
C Specification

```
#include <intersect_ray_triangle.h>
inline vector float _intersect_ray_triangle(vector float ro, vector float rd,
                                            vector float hit, const vector float p[3], float id)

#include <intersect_ray_triangle_v.h>
inline void _intersect_ray_triangle_v(vector float hit[4],
                                            vector float rox, vector float roy, vector float roz,
                                            vector float rdx, vector float rdy, vector float rdz,
                                            vector float p0x, vector float p0y, vector float p0z,
                                            vector float p1x, vector float p1y, vector float p1z,
                                            vector float p2x, vector float p2y, vector float p2z,
                                            vector float id)

#include <intersect_ray1_triangle8_v.h>
inline void _intersect_ray1_triangle8_v(vector float hit[8],
                                            vector float rox, vector float roy, vector float roz,
                                            vector float rdx, vector float rdy, vector float rdz,
                                            const vector float p0x[2], const vector float p0y[2],
                                            const vector float p0z[2], const vector float p1x[2],
                                            const vector float p1y[2], const vector float p1z[2],
                                            const vector float p2x[2], const vector float p2y[2],
                                            const vector float p2z[2], const vector float ids[2])

#include <intersect_ray8_triangle1_v.h>
inline void _intersect_ray8_triangle1_v(vector float hit_t[2], vector float hit_u[2], vector float hit_v[2],
                                            vector unsigned int hit_id[2],
                                            const vector float rox[2], const vector float roy[2],
                                            const vector float roz[2], const vector float rdx[2],
                                            const vector float rdy[2], const vector float rdz[2],
                                            const vector float edgex[2], const vector float edgey[2],
                                            const vector float edgez[2],
                                            vector float p0x, vector float p0y, vector float p0z,
                                            vector float p1x, vector float p1y, vector float p1z,
                                            vector float p2x,vector float p2y, vector float p2z,
                                            vector unsigned int ids[2])

#include <libvector.h>
vector float intersect_ray_triangle(vector float ro, vector float rd, vector float hit,
                                            const vector float p[3], float id)

#include <libvector.h>
void intersect_ray_triangle_v(vector float hit[4], vector float rox, vector float roy, vector float roz,
                                            vector float rdx, vector float rdy, vector float rdz,
                                            vector float p0x, vector float p0y, vector float p0z,
                                            vector float p1x, vector float p1y, vector float p1z,
                                            vector float p2x, vector float p2y, vector float p2z,
```

```
                                                    vector float id)

#include <libvector.h>
void intersect_ray1_triangle8_v(vector float hit[8], vector float rox, vector float roy, vector float roz,
                                                    vector float rdx, vector float rdy, vector float rdz,
                                                    const vector float p0x[2], const vector float p0y[2],
                                                    const vector float p0z[2], const vector float p1x[2],
                                                    const vector float p1y[2], const vector float p1z[2],
                                                    const vector float p2x[2], const vector float p2y[2],
                                                    const vector float p2z[2], const vector float ids[2])

#include <libvector.h>
void intersect_ray8_triangle1_v(vector float hit_t[2], vector float hit_u[2], vector float hit_v[2],
                                                    vector unsigned int hit_id[2],
                                                    const vector float rox[2], const vector float roy[2],
                                                    const vector float roz[2], const vector float rdx[2],
                                                    const vector float rdy[2], const vector float rdz[2],
                                                    const vector float edgex[2], const vector float edgey[2],
                                                    const vector float edgez[2],
                                                    vector float p0x,vector float p0y, vector float p0z,
                                                    vector float p1x, vector float p1y, vector float p1z,
                                                    vector float p2x, vector float p2y, vector float p2z,
                                                    vector unsigned int ids[2])
```

Descriptions

The *intersect_ray_triangle* subroutines determines if a ray intersects a given triangle. The ray is defined by its 3-D origin *ro* and direction *rd.* The triangle is defined by three points specified by the 3-D vertices contained in the array *p*. The routine returns an accumulated hit record consisting of *t* (distance from the ray origin to the intersection), *u* and *v* (the triangle's parameterized u,v intersection coordinates), and *id* (the intersecting triangle id). The hit records is packed into a floating-point vector as follows.



The paramter *hit* is the accumulated hit record of all previous triangle intersections with the given ray. The hit record (return value) is updated with new information if ray intersects the triangle before the intersection defined by the input hit record *h*.

The *intersect_ray_triangle_v* subroutine determines the 4 simultaneous ray-triangle intersection. The rays (specified by the ray origins, *rox*, *roy*, *roz*, and ray directions, *rdx*, *rdy*, *rdz*), and triangles (specified by *p0x*, *p0y*, *p0z*, *p1x*, *p1y*, *p1z*, *p2x*, *p2y*, and *p2z*), are expressed in parallel array format.

The *intersect_ray1_triangle8_v* subroutine determines if a single ray intersects 8 triangles. The input array (specified by *rox*, *roy*, *roz*, *rdx*, *rdy*, *rdz*) is expressed as a structure of arrays and can be considered a 1 ray (such that its component must be replicated across the vector) or 4 rays. Eight hit records, *hit[8]*, are updated. These hit records are store in a SOA (structure of array) form, such that:

hit[0] = t components for the first 4 triangles
hit[1] = u components for the first 4 triangles
hit[2] = v components for the first 4 triangles
hit[3] = triangle id for the first 4 triangles
hit[4] = t components for the second 4 triangles

hit[5] = u components for the second 4 triangles
hit[6] = v components for the second 4 triangles
hit[7] = triangle id for the second 4 triangles

The eight hit records must ultimately be merged to determine the intersection for the ray. This merge can be performed after all the ray-triangle intersections are performed.

The *intersect_ray8_triangle1_v* subroutine determines if a set of 8 rays intersects the given triangle. The eight hit records (specified by *hit_t*, *hit_u*, *hit_v*, and *hit_id*) and rays (specified by *rox*, *roy*, *roz*, *rdx*, *rdy*, and *rdz*) are expressed in parallel array form. The triangle being is also expressed in parallel array form such that the individual component of the vertices (*p0x*, *p0y*, *p0z*, *p1x*, *p1y*, *p1z*, *p2x*, *p2y*, and *p2z*) must be pre-replicated (splated) across the entire array. The caller must also pre-compute the triangle edges. These are specified by parameters *edgex*, *edgey*, and *edgez* and are computed as follows:

```
edgex[0] = p1x - p0x;        edgex[1] = p2x - p0x;
edgey[0] = p1y - p0y;        edgey[1] = p2y - p0y;
edgez[0] = p1z - p0z;        edgez[1] = p2z - p0z;
```

Dependencies

cross_product on page 176
dot_product on page 178
load_vec_float on page 187

See Also

Ray Tracing on Programmable Graphics Hardware; Purcell, Buck, Mask, Hanrahan; ACM Transactions on Graphics, Proceedings of ACM Siggraph 2002; July 2002, Volume 21, Number 3, pages 703-712.

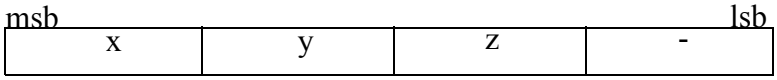
13.6 inv_length_vec

C Specification

```
#include <inv_length_vec3.h>
inline float _inv_length_vec3(vector float v)

#include <inv_length_vec3_v.h>
inline vector float _inv_length_vec3_v(vector float x, vector float y, vector float z)

#include <libvector.h>
float inv_length_vec3(vector float v)

#include <libvector.h>
vector float inv_length_vec3(vector float x, vector float y, vector float z)
```

Descriptions

The *inv_length_vec3* subroutine computes the reciprocal of the magnitude of the 3-D vector specified by the input parameter *v*. The 3 components of the input vector are contained in the most significant components of the 128-bit floating-point vector *v*.



result = 1.0 / sqrt(v.x*v.x + v.y*v.y + v.z*v.z)

The *inv_length_vec3_v* subroutine simultaneously computes the reciprocal of the magnitude of four 3-component vectors specified as parallel arrays by the input parameters *x*, *y*, *z*. The resulting 4 values are returned as a 128-bit, floating-point vector.

Dependencies

sum_across_float on page 192
rsqrtf4 in SIMD Math library

See Also

length_vec on page 184

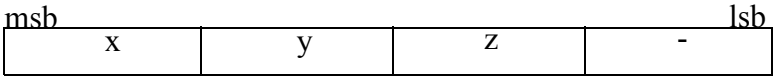
13.7 length_vec

C Specification

```
#include <length_vec3.h>
inline float _length_vec3(vector float v)

#include <length_vec3_v.h>
inline vector float _length_vec3_v(vector float x, vector float y, vector float z)

#include <libvector.h>
float length_vec3(vector float v)

#include <libvector.h>
vector float length_vec3(vector float x, vector float y, vector float z)
```

Descriptions

The *length_vec3* subroutine computes the magnitude of the 3-D vector specified by the input parameter *v*. The 3 components of the input vector are contained in the most significant components of the 128-bit floating-point vector *v*.



result = sqrt(v.x*v.x + v.y*v.y + v.z*v.z)

The *length_vec3_v* subroutine simultaneously computes the magnitude of four 3-component vectors specified as parallel arrays by the input parameters *x*, *y*, *z*. The resulting 4 values are returned as a 128-bit, floating-point vector.

Dependencies

sum_across_float on page 192
sqrtf4 in SIMD Math library

See Also

inv_length_vec on page 183

13.8 lerp_vec

C Specification

```
#include <lerp_vec4.h>
inline vector float _lerp_vec4(vector float v1, vector float v2, vector float t)

#include <lerp_vec2_v.h>
inline void _lerp_vec2_v(vector float *xout, vector float *yout, vector float x1,
                         vector float y1, vector float x2, vector float y2,
                         vector float t)

#include <lerp_vec3_v.h>
inline void _lerp_vec3_v(vector float *xout, vector float *yout, vector float *zout,
                         vector float x1, vector float y1, vector float z1,
                         vector float x2, vector float y2, vector float z2,
                         vector float t)

#include <lerp_vec4_v.h>
inline void _lerp_vec4_v(vector float *xout, vector float *yout, vector float *zout,
                         vector float *wout, vector float x1, vector float y1,
                         vector float z1, vector float w1, vector float x2,
                         vector float y2, vector float z2, vector float w2,
                         vector float t)

#include <libvector.h>
vector float lerp_vec4(vector float v1, vector float v2, vector float t)

#include <libvector.h>
void lerp_vec2_v(vector float *xout, vector float *yout, vector float x1, vector float y1,
                 vector float x2, vector float y2, vector float t)

#include <libvector.h>
void lerp_vec3_v(vector float *xout, vector float *yout, vector float *zout, vector float x1,
                 vector float y1, vector float z1, vector float x2,
                 vector float y2, vector float z2, vector float t)

#include <libvector.h>
void lerp_vec4_v(vector float *xout, vector float *yout, vector float *zout,
                 vector float *wout, vector float x1, vector float y1,
                 vector float z1, vector float w1, vector float x2,
                 vector float y2, vector float z2, vector float w2,
                 vector float t)
```

Descriptions

The *lerp_vec4* subroutine computes the vertex of the linear interpolation between vertices *v1* and *v2* corresponding to the linear interpolation factor *t*.

$$vout = (1-t) * v1 + t * v2$$

The linear interpolation factor is typically a scalar that has been replicated (splatted) across all component of a vector. However, this subroutine does allow a per-component interpolation factor.

The *lerp_vec2_v* subroutine computes 4 2-D vectices of the linear interpolation between 4 2-D vertex pairs for 4 interpolation vectors *t*. The input and output vertices are expressed in parallel array format.

The *lerp_vec3_v* subroutine computes 4 3-D vectices of the linear interpolation between 4 3-D vertex pairs for 4 interpolation vectors *t*. The input and output vertices are expressed in parallel array format.

The *lerp_vec4_v* subroutine computes 4 4-D vectices of the linear interpolation between 4 4-D vertex pairs for 4 interpolation vectors *t*. The input and output vertices are expressed in parallel array format.

Dependencies

See Also

clip_ray on page 175

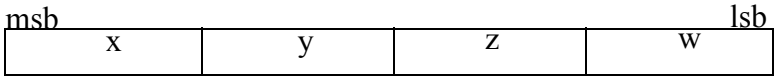
13.9 load_vec_float

C Specification

```
#include <load_vec_float4.h>
inline vector float _load_vec_float4(float x, float y, float z, float w)

#include <libvector.h>
vector float load_vec_float4(float x, float y, float z, float w)
```

Descriptions

The *load_vec_float4* subroutine loads 4 independent, floating-point values (*x*, *y*, *z*, and *w*) into a 128-bit, floating-point vector and returns the vector. The vector is loaded as follows:



Dependencies

See Also

load_vec_int on page 188

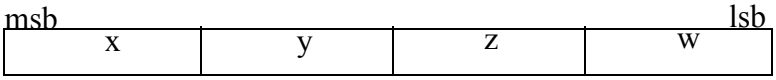
13.10 load_vec_int

C Specification

```
#include <load_vec_int4.h>
inline vector signed int _load_vec_int4(signed int x, signed int y, signed int z,
                                        signed int w)

#include <libvector.h>
vector signed int load_vec_int4(signed int x, signed int y, signed int z, signed int w)
```

Descriptions

The *load_vec_int4* subroutine loads 4 independent, 32-bit, signed integer values (*x*, *y*, *z*, and *w*) into a 128-bit, signed integer vector and returns the vector. The vector is loaded as follows:



Dependencies

See Also

load_vec_float on page 187

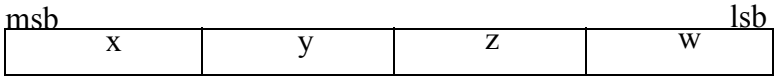
13.11 normalize

C Specification

```
#include <normalize3.h>
inline vector float _normalize3(vector float in)

#include <normalize3_v.h>
inline void _normalize3_v(vector float *xOut, vector float *yOut, vector float *zOut,
                          vector float xIn, vector float yIn, vector float zIn)

#include <normalize4.h>
inline vector float _normalize4(vector float in)

#include <libvector.h>
vector float normalize3(vector float in)

#include <libvector.h>
void normalize3_v(vector float *xOut, vector float *yOut, vector float *zOut,
                  vector float xIn, vector float yIn, vector float zIn)

#include <libvector.h>
vector float normalize4(vector float in)
```

Descriptions

The *normalize3* subroutine normalizes (to unit length) the input vector specified by the parameter *in* and returns the resulting vector. The input and output vectors are 3 component vectors stored in a 128-bit, floating-point, SIMD vector. The resulting 4th (least significant) component is undefined.




```
len = sqrt(in.x*in.x + in.y*in.y + in.z*in.z)
result.x = in.x / len
result.y = in.y / len
result.z = in.z / len
```

The *normalize3_v* subroutine simultaneously normalizes four 3-component vectors specified as parallel arrays by the input parameters *xIn*, *yIn*, *zIn*. The resulting 4 normalized vectors are returned in parallel array format into the memory pointed to by input parameters *xOut*, *yOut*, and *zOut*.

The *normalize4* subroutine normalizes the 4-component vector specified by the input parameter *in* and returns the resulting vector. The subroutine is suitable for normalizing quaternions.

```
len = sqrt(in.x*in.x + in.y*in.y + in.z*in.z + in.w*in.w)
result.x = in.x / len
result.y = in.y / len
result.z = in.z / len
result.w = in.w / len
```

Dependencies

rsqrtf4 in SIMD Math library

See Also

inv_length_vec on page 183

13.12 reflect_vec

C Specification

```
#include <reflect_vec3.h>
inline vector float _reflect_vec3(vector float vec, vector float normal)

#include <reflect_vec3_v.h>
inline void _reflect_vec3_v(vector float *rx, vector float *ry, vector float *rz,
                            vector float vx, vector float vy, vector float vz,
                            vector float nx, vector float ny, vector float nz)

#include <libvector.h>
vector float reflect_vec3(vector float vec, vector float normal)

#include <libvector.h>
void reflect_vec3_v(vector float *rx, vector float *ry, vector float *rz, vector float vx,
                    vector float vy, vector float vz, vector float nx,
                    vector float ny, vector float nz)
```

Descriptions

The *reflect_vec3* subroutine computes the reflection vector for light direction specified by input parameter *vec* off a surface whose normal is specified by the input parameter *normal* and returns the resulting reflection vector. The inputs, *vec* and *normal*, are normalized, 3-component vectors. The reflection vector is computed as follows:

$$\text{reflect\_vec3}(\text{vec}, \text{normal}) = \text{vec} - 2 \times (\text{vec} \cdot \text{normal}) \times \text{normal}$$

The *reflect_vec3_v* subroutine simultaneously computes 4 reflection vectors. Inputs and outputs are specified as parallel arrays. The normalized light directions are specified by input parameters *vx*, *vy*, and *vz*. The normalized surface normals are specified by input parameters *nx*, *ny*, and *nz*. The resulting reflection vectors are returned in *rx*, *ry*, and *rz*.

Dependencies

See Also

13.13 sum_across_float

C Specification

```
#include <sum_across_float3.h>
inline float _sum_across_float3(vector float v)

#include <sum_across_float4.h>
inline float _sum_across_float4(vector float v)

#include <libvector.h>
float sum_across_float3(vector float v)

#include <libvector.h>
float sum_across_float4(vector float v)
```

Descriptions

The *sum_across_float4* subroutine sums the 4 components of the 128-bit, floating-point vector *v* and returns the result.

The *sum_across_float3* subroutine sums only the 3 most significant components of the 128-bit, floating-point vector *v* and returns the result.

Dependencies

See Also

13.14 xform_norm3

C Specification

```
#include <xform_norm3.h>
inline vector float _xform_norm3(vector float in, const vector float *m)

#include <xform_norm3_v.h>
inline void _xform_norm3_v(vector float *xout, vector float *yout, vector float *zout,
                           vector float xin, vector float yin, vector float zin,
                           const vector float *m)

#include <libvector.h>
vector float xform_norm3(vector float in)

#include <libvector.h>
void xform_norm3_v(vector float *xout, vector float *yout, vector float *zout,
                   vector float xin, vector float yin, vector float zin,
                   const vector float *m)
```

Descriptions

The *xform_norm3* subroutine transforms a 3-D, row vector, *in*, by the upper left 3x3 of the 4x4 matrix *m* to produce a 3-D row vector (*out*). The three components of the 3-D vector are stored in the 3 most signifi-

$$[\text{out}] = [\text{in}] \times [\text{m}]$$

cant fields of a 128-bit, floating-poit vector. The transformation ignores the w component (4th) of the input vector. The 4x4 matrix is stored row-ordereds in 4 floating-point vectors. Consult the *Matrix Library* documentation for more details.

The *xform_vec3_v* subroutine transforms four 3-D vectors specified by *xin*, *yin*, and *zin*, by the upper left 3x3 matrix of a replicated 4x4 matrix *m* to produce four 3-D vectors, *xout*, *yout*, and *zout*. The input and output vectors are specified in parallel array format. That is, each vector component, x , y, and z, are maintained in seperate arrays. The arrays are 128-bit, floating-point vectors and thus contain 4 entries. The input matrix is a 4x4, row ordered, array of 128-bit floating point vectors. Typically, this is a replicated matrix created using the *splat_matrix4x4* subroutine. However, the matrix need not be replicated. Each component of the matrix entries is used to transform the corresponding component of the input vectors.

Programmer Notes

The vectored forms of the xform_norm3 routine, *xform_norm3_v, xform_vec4_v*, is constructed from a set of macros. These macros can be used directly to eliminate inefficiencies produced when transforming an array of normals. For example, the following function:

```
#include <xform_norm3_v.h>
void xform_array(vector float *xout, vector float *yout, vector float *zout,
                 vector float *xin, vector float *yin,
                 vector float *zin, vector float *win,
```

```
                                                vector float *matrix, int count)
{
      int i;
      for (i=0; i<count; i++) {
            _xform_norm3(xout++, yout++, zout++, *xin++, *yin++, *zin++, matrix);
      }
}
```

can be written so that the matrix is not repeatedly reloaded by using the underlying macros as follows:

```
#include <xform_norm3_v.h>
void xform_array(vector float *xout, vector float *yout, vector float *zout,
                                                vector float *xin, vector float *yin,
                                                vector float *zin, vector float *matrix, int count)
{
      int i;
      _DECLARE_MATRIX_3X3_V(matrix);
      _LOAD_MATRIX_3X3_V(matrix);
      for (i=0; i<count; i++) {
            _XFORM_NORM3_V(xout++, yout++, zout++, *xin++, *yin++, *zin++, matrix);
      }
}
```

Dependencies

See Also

splat_matrix4x4 on page 94
xform_vec on page 195

13.15 xform_vec

C Specification

```
#include <xform_vec3.h>
inline vector float _xform_vec3(vector float in, const vector float *m)

#include <xform_vec4.h>
inline vector float _xform_vec4(vector float in, const vector float *m)

#include <xform_vec3_v.h>
inline void _xform_vec3_v(vector float *xout, vector float *yout, vector float *zout,
                          vector float *wout, vector float xin,
                          vector float yin, vector float zin,
                          const vector float *m)

#include <xform_vec4_v.h>
inline void _xform_vec4_v(vector float *xout, vector float *yout, vector float *zout,
                          vector float *wout, vector float xin,
                          vector float yin, vector float zin, vector float win,
                          const vector float *m)

#include <libvector.h>
vector float xform_vec3(vector float in, const vector float *m)

#include <libvector.h>
vector float xform_vec4(vector float in, const vector float *m)

#include <libvector.h>
void xform_vec3_v(vector float *xout, vector float *yout, vector float *zout,
                  vector float *wout, vector float xin,
                  vector float yin, vector float zin,
                  const vector float *m)

#include <libvector.h>
void xform_vec4_v(vector float *xout, vector float *yout, vector float *zout,
                  vector float *wout, vector float xin,
                  vector float yin, vector float zin, vector float win,
                  const vector float *m)
```

Descriptions

The *xform_vec3* subroutine transforms a 3-D, row vector, *in*, by the 4x4 matrix *m* to produce a 4-D row vector (*out*). The three components of the 3-D vector are stored in the 3 most significant fields of a 128-

$$[\text{out}] = [\text{in}] \times [\text{m}]$$

bit, floating-poit vector. The transformation assumes a w component (4th) of the input vector to be 1.0. The 4x4 matrix is stored row-ordereds in 4 floating-point vectors. Consult the *Matrix Library* documentation for more details.

The *xform_vec4* subroutine transforms a 4-D, row vector, *in*, by the 4x4 matrix *m* to produce a 4-D, row vector (*out*).

The *xform_vec3_v* subroutine transforms four 3-D vectors specified by *xin*, *yin*, and *zin*, by the replicated 4x4 matrix *m* to produce four 4-D vectors, *xout*, *yout*, *zout* and *wout*. The transformation assumes the 4th (w) components of the input vector are 1.0. The input and output vectors are specified in parallel array format. That is, each vector component, x , y, z, and w, are maintained in seperate arrays. The arrays are 128-bit, floating-point vectors and thus contain 4 entries. The input matrix is a 4x4, row ordered, array of 128-bit floating point vectors. Typically, this is a replicated matrix created using the *splat_matrix4x4* sub-routine. However, the matrix need not be replicated. Each component of the matrix entries is used to transform the corresponding component of the input vectors.

The *xform_vec4_v* subroutine is identical to *xform_vec3_v* except the input vectors are 4 dimensional.

Programmer Notes

The vectored forms of the xform_vec routines, *xform_vec3_v* and *xform_vec4_v*, are constructed from a set of macros. These macros can be used directly to eliminate inefficiencies produced when transforming an array of vectors. For example, the following function:

```
#include <xform_vec4_v.h>
void xform_array(vector float *xout, vector float *yout, vector float *zout,
                                        vector float *wout, vector float *xin,
                                        vector float *yin, vector float *zin,
                                        vector float *win, vector float *matrix, int count)
{
    int i;
    for (i=0; i<count; i++) {
        _xform_vec4(xout++, yout++, zout++, wout++, *xin++, *yin++, *zin++, *win++,
                                        matrix);
    }
}
```

can be written so that the matrix is not repeatedly reloaded by using the underlying macros as follows:

```
#include <xform_vec4_v.h>
void xform_array(vector float *xout, vector float *yout, vector float *zout,
                                        vector float *wout, vector float *xin,
                                        vector float *yin, vector float *zin,
                                        vector float *win, vector float *matrix, int count)
{
    int i;
    _DECLARE_MATRIX_4X4_V(matrix);
    _LOAD_MATRIX_4X4_V(matrix);
    for (i=0; i<count; i++) {
        _XFORM_VEC4_V(xout++, yout++, zout++, wout++, *xin++, *yin++, *zin++, *win++,
                                        matrix);
    }
}
```

Dependencies

See Also

splat_matrix4x4 on page 94
xform_norm3 on page 193

14. Revision Log

The section documents the significant areas of change made to the example libraries for each release of the SDK.

Table 1: Revision Log

Revision Date	Contents of Modification
SDK 1.0 10/31/2005 01/20/2006	Initial release of a public SDK library documentation.
SDK 1.1 6/30/2006	Correct description and change parameter for fft_2d subroutine. Improve documentation for fft_1d_r2.
SDK 2.0 12/14/2006	Changes include: • removal of the *math* library. Most of the functions provided in the math library have been are now provided in the C (scalar functions) and SIMDmath (SIMD functions) libraries. • add return value to *inverse_matrix4x4* to indicate singularity. • add SW managed cache library documentation • minor typographical errors and clarifications
SDK 2.1 3/26/2007	Minor documentation corrections in the large matrix and vector libraries.
SDK 3.0 9/14/2007	Changes include: • removal of the *audio*, *curve*, *noise*, *oscillator*, and *surface* libraries. • removal of the *sample C* library. These functions are now fully supported by the newlib C library. • move to *sim* library to the simulator. Documentation is now provided by the simulator documentation. • renamed *fft* library to *fft_example* to avoid name conflicts with other FFT libraries.