Cell Broadband Engine SDK Libraries

Overview and Users Guide

Version 1.1 (public SDK)

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

DRAFT - Not Approved for Customer Distribution
June 30, 2006
Version 1.1

Preface

The document provides descriptions of the Cell Broadband Engine processor C-language libraries provided in the IBM public SDK. The following topics are discussed:

The **overview** of the strategy used to select, name, and package supported/implemented functions.

The **users guide** detailing usage information on supported functions.

See the *Revision Log* on page 421 for a list of changes to the 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 SDK libraries function. This document has been organized into the following sections.

Strategy

The following observations, rules, and guidelines are the basis for determining and developing the libraries and their contents:

- Provide specialized and optimized functions specifically targeted at producing reusable and efficient functions for the processor / architecture. As such, must of the library focus is targeted for the SPE processor.
- Provide the foundation for the development of a set of application samples and/or workloads.
- Provide libraries/functions that abstract HW features and functions.
- Provide libraries that address the primary target processor applications.
- Provide both vectored and scalar subroutines. Vectored PPE-targeted routines exploit the Vector/SIMD multimedia extension.
- Provide features as static (non-shared) library subroutines as well as and inlineable subroutines.
- Provide readable and well documented source code that can be easily customized and tailored to the end users needs and/or data formats.
- Provide test functions used to verify the correctness and accuracy claims for each function.

- Provide no special handling of erroneous inputs or conditions (e.g. divide by zero; out of supported range inputs).

Naming and Packaging Conventions

An important objective of the library development is usability. Library subroutines must be provided as callable routines and inlines routines without occurring additional linkage, data, and/or test overhead. It is also considered important that software be retargetable to a different processor without undue editing and still be readily obvious what processor is being targeted. As such, the following naming and packaging of the library subroutines is proposed.

- Libraries are a collection of subroutines of similar function for a given target processor.
- Libraries (linked, non-shared archives) are provided for applications wishing external subroutines. Each library subroutine resides in its own object file so that inadvertent inclusion of unneeded functions are minimized.
- Libraries are “shipped” into the *lib* for libraries targeted to run on the PPE processor and *lib/spu* for libraries targeted to run on the SPE processor.
- Each library provides a header file that includes externs for the exported library subroutines and defines for appropriate constants and enumerants. The header files are located in the directory *include/**library_name***.**h** for libraries targeted to run on the PPE processor and *include/**spu/library_name***.**h** for libraries targeted to run on the SPE (for example, include/spu/libmath.h).
- Libraries may support both SIMD (vectored) and scalar functions.
- Library subroutines which simultaneously operate on 4 independent pieces of data (i.e., SIMD or vectored) are suffixed with *_v*.
- Library subroutines using or supporting double precision data are suffixed with *_d* (or *_dv* for double precision vectored routines) unless a standard name exists for the function. For example, the C99 specification defines a double precision cosine function as *cos*. In this case, *cos* will be used instead of *cos_d.*
- Most subroutines are made available as inline subroutines. The inline subroutines are made available in header files. Each header file contains (typically) only a single subroutine so that users can include only the routines needed.
- Inlined subroutines are uniquely name by prefixing an underscore (“_”) before the equivalent library routine name.
- The inlineable subroutine header files are located in *include/**routine_name*** for library functions targeted at the PPE processor and *include/spu/**routine_name*** for function targeted at the SPE processor. The ***routine_name*** is the name exported by the library function and not the inlined subroutine name - it does not contain the leading underscore.

4. C Library

The C library is an SPE only library containing a collection of functions typically found in a standard C99 (ISO/IEC 9899:1999) library. In addition, several POSIX.1 functions are also provided. This library has been implemented in accordance with the JSRE C/C++ Language Extension specification. Only the C locale is implemented with no wide character support.

The documentation for this library is broken down into three sections. The first section documents functions that are executed entirely on the SPE. The second section documents functions that are serviced by the PPE (i.e., the control plane processor). The third section documents the memory heap management and allocation function.

Name(s)

libc.a

Header File(s)

<assert.h>
<ctype.h>
<errno.h>
<fcntl.h>
<fenv.h>
<float.h>
<inttypes.h>
<iso646.h>
<limits.h>
<math.h>
<setjmp.h>
<stdbool.h>
<stddef.h>
<stdint.h>
<stdio.h>
<stdlib.h>
<string.h>
<unistd.h>
<sys/mman.h>
<sys/stat.h>
<sys/time.h>
<sys/timex.h>
<sys/types.h>

4.1 SPE Serviced C Library Functions

This section documents the C library functions that are executed entirely on the SPE. There also is a several memory heap function also entirely executed on the SPE in *SPE Local Storage Memory Allocation* on page 70.

4.1.1 abort

C Specification

```
#include <stdlib.h>
void abort(void)
```

Description

The *abort* subroutine causes abnormal termination of an SPE task by performing a halt instruction. Under the Linux operating systems, the halt instruction shall produce a SIGABRT signal.

Abort is implemented as a macro which invokes the inline subroutine *__abort.*

Dependencies

See Also

exit on page 20

4.1.2 atof

C Specification

```
#include <stdlib.h>
double atof(const char *nptr)
```

```
#include <atof.h>
inline double _atof(const char *nptr)
```

Description

The *atof* subroutine converts the initial portion of the string pointed to by the *nptr* parameter to a double precision float. This function is equivalent to:

```
strtod(nptr, (char **)(NULL))
```

Dependencies

strtod on page 57

See Also

atoi on page 16
atoll on page 17

4.1.3 atoi

C Specification

```
#include <stdlib.h>
int atoi(const char *nptr)

#include <atoi.h>
inline int _atoi(const char *nptr)
```

Aliases

```
#include <stdlib.h>
long atol(const char *nptr)
```

Description

The *atoi* subroutine converts the initial portion of the string pointed to by the *nptr* parameter to an integer. This function is equivalent to:

```
(int)atol(nptr)
```

and

```
(int)strtol(nptr, (char **)(NULL), 10)
```

Dependencies

strtol on page 62

See Also

atof on page 15
atoll on page 17

4.1.4 atoll

C Specification

```
#include <stdlib.h>
long long atoll(const char *nptr)

#include <atoll.h>
inline long long _atoll(const char *nptr)
```

Description

The *atoll* subroutine converts the initial portion of the string pointed to by the *nptr* parameter to a long long integer. This function is equivalent to:

```
strtoll(nptr, (char **)(NULL), 10)
```

Dependencies

strtoll on page 63

See Also

atof on page 15
atoi on page 16

4.1.5 assert

C Specification

```
#include <assert.h>
void assert(scalar expresion);
```

Description

If the macro NDEBUG is defined when <assert.h> was last included, the macros assert() generates no code, and hence does nothing at all. Otherwise, the macro assert() prints an error message and terminates the program by invoking the halt intrinsic if *expression* is false (i.e., compares equal to zero).

assert() is implemented as a macro; if the expression tested has side affects, program behaviour will be different depending on whether NDEBUG is defined. This may create Heisenbugs which go away when debugging is turned on.

4.1.6 bzero

C Specification

```
#include <string.h>
void bzero(void *s, unsigned int n)
```

Description

The *bzero* subroutine sets the first *n* bytes of the string *s* to zero.

This utility is implemented as a call to *memset* (s, 0, n). See the memset routine for additional details.

Dependencies

memset on page 41

See Also

4.1.7 exit

C Specification

#include <stdlib.h>
inline void exit(int status)

Description

The *exit* subroutine causes normal program termination and the value of 8 least significant bits of *status* is returned with a stop and signal instruction (0x20xx).

Dependencies

See Also

abort on page 14

4.1.8 imaxabs

C Specification

```
#include <inttypes.h>
intmax_t imaxabs(intmax_t j)
```

```
#include <inttypes.h>
vector long long imaxabs_v(vector long long j)
```

```
#include <isalnum.h>
inline intmax_t _imaxabs(intmax_t j)
```

```
#include <isalnum.h>
inline vector long long _imaxabs_v(vector long long j)
```

Description

The *imaxabs* subroutine computes the absolute value of the maximum input value *j*. For the SPE, this is a 64-bit long long integer.

The *imaxabs_v* computes the absolute value of a SIMD pair of long long integers.

Dependencies

See Also

fabs on page 203

4.1.9 imaxdiv

C Specification

```
#include <inttypes.h>
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom)
```

```
#include <imaxdiv.h>
inline imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom)
```

Description

The *imaxdiv* subroutine computes the quotient and remainder of the maximun signed integer values *numer* and *denom*. For the SPE, this is a 64-bit long long integer. The result is returned in a imaxdiv_t structure.

quot = *numer* / *denom*
rem = *numer* % *denom*

The results are undefined if *denom* is zero.

Dependencies

See Also

divide (integer) on page 194

4.1.10 isalnum

C Specification

```
#include <ctype.h>
int isalnum(int c)

#include <isalnum.h>
inline int _isalnum(int c)
```

Description

The *isalnum* subroutine checks whether input character *c* is an alphanumeric character. Non-zero is returned if *c* is alphanumeric, otherwise 0 is returned.

This routine is equivalent to *isalpha(c)* || *isdigit(c)*.

4.1.11 isalpha

C Specification

```
#include <ctype.h>
int isalpha(int c)

#include <isalpha.h>
inline int _isalpha(int c)
```

Description

The *isalpha* subroutine checks whether input character *c* is an alphabetic character. Non-zero is returned if *c* is alphabetic, otherwise 0 is returned.

This routine is equivalent to *isupper(c)* || *islower(c)* for the supported locale.

4.1.12 isascii

C Specification

```
#include <ctype.h>
int isascii(int c)

#include <isascii.h>
inline int _isascii(int c)
```

Description

The *isascii* subroutine checks whether input character *c* is a 7-bit unsigned character value that fits into the ASCII character set. Non-zero is returned if *c* is ascii, otherwise 0 is returned.

4.1.13 isblank

C Specification

#include <ctype.h>
int isblank(int c)

#include <isblank.h>
inline int _isblank(int c)

Description

The *isblank* subroutine checks whether input character *c* is a blank character; that is a space or a tab. Non-zero is returned if *c* is a blank character, otherwise 0 is returned.

4.1.14 iscntrl

C Specification

#include <ctype.h>
int iscntrl(int c)

#include <iscntrl.h>
inline int _iscntrl(int c)

Description

The *iscntrl* subroutine checks whether input character *c* is a control character (numeric value 0-31 or 127). Non-zero is returned if *c* is a control character, otherwise 0 is returned.

4.1.15 isdigit

C Specification

```
#include <ctype.h>
int isdigit(int c)

#include <isdigit.h>
inline int _isdigit(int c)
```

Description

The *isdigit* subroutine checks whether input character *c* is a digit (‘0’ through ‘9’). Non-zero is returned if *c* is a digit, otherwise 0 is returned.

4.1.16 isgraph

C Specification

```
#include <ctype.h>
int isgraph(int c)
```

```
#include <isgraph.h>
inline int _isgraph(int c)
```

Description

The *isgraph* subroutine checks whether input character *c* is any printable character except space. Non-zero is returned if *c* is a graphic character, otherwise 0 is returned.

4.1.17 islower

C Specification

```
#include <ctype.h>
int islower(int c)

#include <islower.h>
inline int _islower(int c)
```

Description

The *islower* subroutine checks whether input character *c* is a lower-case character. Non-zero is returned if *c* is lower-case, otherwise 0 is returned.

4.1.18 isprint

C Specification

```
#include <ctype.h>
int isprint(int c)
```

```
#include <isprint.h>
inline int _isprint(int c)
```

Description

The *isprint* subroutine checks whether input character *c* is any printable character including space. Non-zero is returned if *c* is a prinable character, otherwise 0 is returned.

4.1.19 ispunct

C Specification

```
#include <ctype.h>
int ispunct(int c)
```

```
#include <ispunct.h>
inline int _ispunct(int c)
```

Description

The *ispunct* subroutine checks whether input character *c* is any printable character which is not a space or an alphanumberic character. Non-zero is returned if *c* is a “punct” character, otherwise 0 is returned.

4.1.20 isspace

C Specification

```
#include <ctype.h>
int isspace(int c)

#include <isspace.h>
inline int _isspace(int c)
```

Description

The *isspace* subroutine checks whether input character *c* is a white-space character. For the supported locale, white-space characters include form-feed, newline, carriage return, horizontal tab and vertical tab. Non-zero is returned if *c* is a white-space character, otherwise 0 is returned.

4.1.21 isupper

C Specification

```
#include <ctype.h>
int isupper(int c)

#include <isupper.h>
inline int _isupper(int c)
```

Description

The *isupper* subroutine checks whether input character *c* is an upper-case character. Non-zero is returned if *c* is upper-case, otherwise 0 is returned.

4.1.22 isxdigit

C Specification

```
#include <ctype.h>
int isxdigit(int c)

#include <isxdigit.h>
inline int _isxdigit(int c)
```

Description

The *isxdigit* subroutine checks whether input character *c* is hexidecimal digit. Hexidecimal digits include 0-9, a-f, and A-F. Non-zero is returned if *c* is a hex digit, otherwise 0 is returned.

4.1.23 longjmp

C Specification

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val)
```

Descriptions

The *longjmp* subroutine restores the environment saved by the last call of *setjmp* with the corresponding *env* argument. After *longjmp* is completed, program execution continues as if the corresponding call of *setjmp* had just returned the value *val*. *longjmp* cannot cause 0 to be returned.

If *longjmp* is invoked with *val* equalling 0, 1 will be returned instead.

Dependencies

See Also

setjmp on page 43

4.1.24 memchr

C Specification

```
#include <string.h>
void * memchr(const void *s, int c size_t n)

#include <memchr.h>
inline void * _memchr(const void *s, int c, size_t n)
```

Description

The *memchr* subroutine scans the first *n* bytes of the memory area pointed to by *s* for the character *c* and returns a pointer to the first occurance of *c*. If *c* is not found, then NULL is returned.

Dependencies

See Also

strchr on page 46
strrchr on page 55

4.1.25 memcmp

C Specification

```
#include <string.h>
int memcmp(const void *s1, cont void *s2, size_t n)
```

```
#include <memcmp.h>
inline int _memcmp(const void *s1, cont void *s2, size_t n)
```

Description

The *memcmp* subroutine compares the first *n* bytes of the memory areas pointed to by *s1* and *s2*. An integer less than, equal to, or greater than zero is returned if *s1* if found, respectively, to be less than, to match, or to be greater than *s2*.

Dependencies

See Also

strcmp on page 47
strncmp on page 52

4.1.26 memcpy

C Specification

```
#include <string.h>
void * memcpy(void * restrict dest, const void * restrict src, size_t n)

#include <memcpy.h>
inline void * _memcpy(void * restrict dest, const void * restrict src, size_t n)
```

Description

The *memcpy* subroutine copies *n* bytes from memory area *src* to memory area *dest*. The memory area may not overlap. The *memcpy* subroutine returns the pointer *dest*.

Dependencies

See Also

strcpy on page 48
strncpy on page 53

4.1.27 memmove

C Specification

```
#include <string.h>
void * memmove(void * restrict dest, const void * restrict src, size_t n)

#include <memcpy.h>
inline void * _memmove(void * restrict dest, const void * restrict src, size_t n)
```

Description

The *memmove* subroutine copies *n* bytes from memory area *src* to memory area *dest*. The source and destination areas may overlap, in that, copying is performed as if the *n* bytes pointed to by *src* are first copied to a temporary array that does not overlap the source and destination arrays; then the *n* bytes are copied into the destination arrays.

The *memmove* subroutine returns the pointer *dest*.

Dependencies

See Also

memcpy on page 39
strcpy on page 48

4.1.28 memset

C Specification

```
#include <string.h>
void * memset(void *s, int c, size_t n)
```

```
#include <memset.h>
inline void * _memset(void *s, int c, size_t n)
```

Description

The *memset* subroutine fills the first *n* bytes of the memory area pointed to by *s* with the constant byte *c*. The *memset* subroutine returns a pointer to the memory area *s*.

Dependencies

See Also

bzero on page 19

4.1.29 rand

C Specification

```
#include <rand.h>
inline signed int _rand(void)

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

#include <stdlib.h>
signed int rand(void)

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

Descriptions

The *rand* subroutine generates a 31-bit uniformly cyclic, pseudo random number. The vector version (*rand_v*) generates a vector of 31-bit random numbers.

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

Dependencies

See Also

srand on page 44
rand_0_to_1 on page 303
rand_minus1_to_1 on page 302

4.1.30 setjmp

C Specification

```
#include <setjmp.h>
int setjmp(jmp_buf env)
```

Descriptions

The *setjmp* subroutine saves the satack context/environment in structure specified by *env* for later used by *longjmp*. The stack context will be invalidated if the function which called *setjmp* returns.

setjmp returns 0 if returning directly, and non-zero when returning from *longjmp* using the save context.

setjmp and *longjmp* are useful for dealing with errors and interrupts encounterd in a low-level subroutine of a program.

Dependencies

See Also

longjmp on page 36

4.1.31 srand

C Specification

```
#include <srand.h>
inline void _srand(unsigned int seed)

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

#include <stdlib.h>
void srand(unsigned in seed)

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

Descriptions

The *srand* subroutine sets the random number seed used by the random number generation subroutines - *rand*, *rand_0_to_1*, and *rand_minus1_to_1*. No restrictions are placed on the value of the seed yet only the 31 lsb (least significant bits) are saved.

The *srand_v* subroutine sets the vectored random number seed used by the vectored random number generation subroutines - *rand_v*, *rand_0_to_1_v*, and *rand_minus1_to_1_v*.

Dependencies

See Also

rand on page 42
rand_0_to_1 on page 303
rand_minus1_to_1 on page 302

4.1.32 strcat

C Specification

#include <string.h>
char * strcat(char * restrict dest, const * restrict src)

#include <strcat.h>
inline char * _strcat(char * restrict dest, const * restrict src)

Description

The *strcat* subroutine apends the string pointed to by *src* to the string pointed to by *dest*, overwriting the termination character ("\0") character at the end of the *dest* string, and then adds a termination character. The strings may not overlap and the *dest* string must have enough space for the resulting concatenated string.

Dependencies

memcpy on page 39

See Also

strncat on page 51

4.1.33 strchr

C Specification

```
#include <string.h>
char * strchr(const char *s, int c)
```

```
#include <strchr.h>
inline char * _strchr(const char *s, int c)
```

Description

The *strchr* subroutine returns a pointer to the first occurrance of the character *c* in the string pointed to by *s*. If *c* is not found, then NULL is returned.

Dependencies

See Also

memchr on page 37
strrchr on page 55

4.1.34 strcmp

C Specification

```
#include <string.h>
int strcmp(const char *s1, const char *s2)
```

```
#include <strcmp.h>
inline int _strcmp(const char *s1, const char *s2)
```

Aliases

```
#include <string.h>
int strcoll(const char *s1, const char *s2)
```

Description

The *strcmp* subroutine compares two strings pointed to by parameters *s1* and *s2*. It returns an integer less than, equal to, or greater thatn zero if string specified by *s1* is found, respectively, to be less than, to match, or to be greater than the string specified by *s2*.

Dependencies

See Also

memcmp on page 38
strncmp on page 52

4.1.35 strcpy

C Specification

#include <string.h>
char * strcpy(char *dest, const char *src)

#include <strcpy.h>
inline char * _strcpy(char *dest, const char *src)

Description

The *strcpy* subroutine copies the string pointed to by *src* (including the termination character “\0”) to the array pointed to by *dest*. The strings may not overlap, and the destination string must be large enough to receive the copy.

Dependencies

See Also

memcpy on page 39
strncpy on page 53

4.1.36 strcspn

C Specification

```
#include <string.h>
size_t strcspn(const char *s, const char *reject)

#include <strcspn.h>
inline size_t _strcspn(const char *s, const char *reject)
```

Description

The *strcspn* subroutine calculates the length of the initial segment of string specified by *s* which consists entirely of characters not in string specified by *reject*.

Dependencies

See Also

strspn on page 56

4.1.37 strlen

C Specification

```
#include <string.h>
size_t strlen(const char *s)
```

```
#include <strlen.h>
inline size_t _strlen(const char*s)
```

Description

The *strlen* subroutine calculates the length of the string pointed to by the parameter *s*, not including the termination character “\0”.

Dependencies

See Also

4.1.38 strncat

C Specification

```
#include <string.h>
void * strncat(char *dest, const char *src, size_t n)
```

```
#include <strncat.h>
inline void * _strncat(char *dest, const char *src, size_t n)
```

Description

The *strncat* subroutine appends the first *n* characters of the string pointed to by *src* to the string pointed to by *dest* (overwriting the termination character at the end of *dest*). The strings may not overlap and the *dest* string must have enough space for the resul.

Dependencies

See Also

strcat on page 45
strcpy on page 48

4.1.39 strncmp

C Specification

```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n)

#include <strncmp.h>
inline int _strncmp(const char *s1. const char *s2, size_t n)
```

Description

The *strncmp* subroutine compares the strings pointed to by parameters *s1* and *s2*. It returns a integer less than, equal to, or greater than zero, if the first (at most) *n* characters of *s1* is found to be, respectively, to be less than, to match, or the be greater than *s2*.

Dependencies

See Also

memcmp on page 38
strcmp on page 47

4.1.40 strncpy

C Specification

#include <string.h>
void * strncpy(char *dest, const char *src, size_t n)

#include <strncpy.h>
inline void * _strncpy(char *dest, const char *src, size_t n)

Description

The *strncpy* subroutine copies at most *n* characters pf the string pointed to by *src* (include the termination character) to the array pointed to by *dest.* The strings may not overlap, and the destination string (*dest*) must be large enough to receive the copy. If there is no termination character in the first *n* bytes of *src*, the result will not be null-terminated.

Dependencies

See Also

memcpy on page 39
strcpy on page 48

4.1.41 strpbrk

C Specification

```
#include <string.h>
char * strpbrk(const char *s, const char *accept)
```

```
#include <strpbrk.h>
inline char * _strpbrk(cons char *s, const char *accept)
```

Description

The *strpbrk* subroutine locates the first occurrence in the string pointed to by *s* of any of the characters in the string pointed to by *accept*, returning the pointer to the first occurrence. If no character is found, then NULL is returned.

Dependencies

See Also

strcspn on page 49
strspn on page 56

4.1.42 strrchr

C Specification

```
#include <string.h>
char * strrchr(const char *s, int c)
```

```
#include <strrchr.h>
inline char * _strrchr(const char *s, int c)
```

Description

The *strrchr* subroutine returns a pointer to the last occurrence of the character *c* in the string pointed to by *s*. If the character does not occur in *s*, then NULL is returned.

Dependencies

See Also

memchr on page 37
strchr on page 46

4.1.43 strspn

C Specification

```
#include <string.h>
size_t strspn(const char *s, const char *accept)

#include <strspn.h>
inline size_t _strspn(const char *s, const char *accept)
```

Description

The *strspn* subroutine calculates the length of the initial segment of the string pointed to by *s* which consists entirely of character in the string pointed to by *accept*.

Dependencies

See Also

strcspn on page 49
strpbrk on page 54

4.1.44 strtod

C Specification

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr)

#include <strtod.h>
inline double _strtod(const char *nptr, char **endptr)
```

Aliases

```
#include <stdlib.h>
long double strtold(const char *nptr, char **endptr)
```

Description

The *strtod* subroutine converts the initial portion of the string pointed to by the *nptr* parameter to a double precision float. The expected form for initial portion of the string is optional whitespace (as recognized by *isspace*), an optional plus ("+") of minus ("-") sign, then either:

- a decimal number,
- a hexidecimal number,
- an infinitity, or
- a NAN (not-a-number).

A decimal number consists of a non-empty sequence of decimal digits possibly containing a radix character ("."), optionally followed by a decimal exponent. A decimal exponent consists of an "E" or "e", followed by an optional plus or minus sign, followed by a non-empty sequence of decimal digits. The exponent indicates a multiplication by a power of 10.

A hexidecimal number consists of a "0x" or "0X" followed by a non-empty sequence of hexidecimal digits possibly containing a radix character, optionally followed by a binary exponent. A binary exponent consists of a "P" or "p", followed by an optional plus or minus sign, followed by a non-empty sequence of decimal digits indicating a multiplication by a power of 2.

An infinity is either "INF" or "INFINITY", disregarding case.

A NAN is "NAN", disregarding case.

If the *endptr* parameter is not NULL, a pointer to the character after the last character of the conversion is stored at the location referenced by *endptr*.

This implementation does nothing special to handle overflow or underflow and as such does **not** set errno.

Inlining this function still requires linking with the C library in order to resolve the static control table - strtod_values.

Dependencies

isspace on page 33
toupper on page 66
exp2 on page 201

See Also

strtof on page 59
strtol on page 62

4.1.45 strtof

C Specification

```
#include <stdlib.h>
float strtof(const char *nptr, char **endptr)
```

```
#include <strtof.h>
inline float _strtof(const char *nptr, char **endptr)
```

Description

The *strtof* subroutine converts the initial portion of the string pointed to by the *nptr* parameter to a single precision float. The expected form for initial portion of the string is optional whitespace (as recognized by *isspace*), an optional plus ("+") of minus ("-") sign, then either:

- a decimal number,
- a hexidecimal number,
- an infinitity, or
- a NAN (not-a-number).

A decimal number consists of a non-empty sequence of decimal digits possibly containing a radix character ("."), optionally followed by a decimal exponent. A decimal exponent consists of an "E" or "e", followed by an optional plus or minus sign, followed by a non-empty sequence of decimal digits. The exponent indicates a multiplication by a power of 10.

A hexidecimal number consists of a "0x" or "0X" followed by a non-empty sequence of hexidecimal digits possibly containing a radix character, optionally followed by a binary exponent. A binary exponent consists of a "P" or "p", followed by an optional plus or minus sign, followed by a non-empty sequence of decimal digits indicating a multiplication by a power of 2.

An infinity is either "INF" or "INFINITY", disregarding case.

A NAN is "NAN", disregarding case.

If the *endptr* parameter is not NULL, a pointer to the character after the last character of the conversion is stored at the location referenced by *endptr*.

This implementation does nothing special to handle overflow or underflow and as such does **not** set errno.

Inlining this function still requires linking with the C library in order to resolve the static control table - strtod_values.

Dependencies

isspace on page 33
toupper on page 66
exp2 on page 201

See Also

strtod on page 57
strtol on page 62

4.1.46 strtok

C Specification

#include <string.h>
char * strtok(char *s, const char *delim)

#include <strtok.h>
inline char * _strtok(char *s, const char *delim)

Description

The *strtok* subroutine can be used to parse the string pointed to by *s* into tokens. The first call to *strtok* should have *s* as its first argument. Subsequent calls should have the first argument set to NULL. Each call routines a pointer to the next token, or NULL when no more tokens are found.

If a token ends with a delimiter, this delimiter character is over-written with a termination character ("\0") and a pointer to the next character is saved for the next call to *strtok*. The delimiter string (*delim*) may be different for each call.

Note: A token is a non-empty string of characters not occurring in the string specifief by *delim*, followed by a termination character ("\0") or by a character occuring in *delim*.

Dependencies

strcspn on page 49
strspn on page 56

See Also

4.1.47 strtol

C Specification

```
#include <stdlib.h>
long int strtol(const char *nptr, char **endptr, int base)

#include <strtol.h>
inline long int _strtol(const char *nptr, char **endptr, int base)
```

Description

The *strtol* subroutine converts the initial portion of the string pointed to by the *nptr* parameter to a long integer according to the base specified by the *base* parameter. *base* must be between 2 and 36 inclusive, or be a special value 0.

The string must begin with a artibrary amount of white space (as defined by *isspace*) followed by a single optional "+" or "-" sign. If the *base* is 0 or 16, the string then may include a "0x" prefix, and the number will be read in base 16. Otherwise, a zero base will be taken as 10 (decimal) unless the next character is "0", in which case it is taken as 8 (octal).

The remainder of the strings is converted to a long integer value, stopping at the first character in which it is not a valid digit in the specified base.

Note: In *bases* above 10, the letter "A", either lower or upper case, represents 10, "B" represents 11, and so forth through "Z" representing 36.

If the *endptr* parameter is not NULL, a pointer to the first invalid character is stored at the location referenced by *endptr*.

If the conversion results in a value that is out-of-range, then errno is set to ERANGE and either LONG_MIN or LONG_MAX is return, depending on the sign of the value.

Inlining this function still requires linking with the C library in order to resolve the static control tables - strtol_values and strtol_limits.

Dependencies

See Also

atoi on page 16
isspace on page 33
strtoll on page 63

4.1.48 strtoll

C Specification

```
#include <stdlib.h>
long long int strtoll(const char *nptr, char **endptr, int base)

#include <strtolk.h>
inline long long int _strtoll(const char *nptr, char **endptr, int base)
```

Description

The *strtoll* subroutine converts the initial portion of the string pointed to by the *nptr* parameter to a long long integer according to the base specified by the *base* parameter. *base* must be between 2 and 36 inclusive, or be a special value 0.

The string must begin with a artibrary amount of white space (as defined by *isspace*) followed by a single optional "+" or "-" sign. If the *base* is 0 or 16, the string then may include a "0x" prefix, and the number will be read in base 16. Otherwise, a zero base will be taken as 10 (decimal) unless the next character is "0", in which case it is taken as 8 (octal).

The remainder of the strings is converted to a long long integer value, stopping at the first character in which it is not a valid digit in the specified base.

Note: In *bases* above 10, the letter "A", either lower or upper case, represents 10, "B" represents 11, and so forth through "Z" representing 36.

If the *endptr* parameter is not NULL, a pointer to the first invalid character is stored at the location referenced by *endptr*.

If the conversion results in a value that is out-of-range, then errno is set to ERANGE and either LLONG_MIN orL LONG_MAX is return, depending on the sign of the value.

Inlining this function still requires linking with the C library in order to resolve the static control tables - strtoll_values and strtoll_limits.

Dependencies

See Also

atoll on page 17
isspace on page 33
strtol on page 62

4.1.49 strxfrm

C Specification

```
#include <string.h>
size_t strxfrm(char *dest, const char *src, size_t n)

#include <strxfrm.h>
inline size_t _strxfrm(char *dest, const char *src, size_t n)
```

Description

The *strxfrm* subroutine transforms the string pointed to by *src* into a form such that the result of *strcmp* on two strings that have been transformed with *strxfrm* is the same as the result of *strcoll* on the two strings before their transformation. The first *n* characters of the transformed string are placed in *dest.* The number of characters required to store the transformed string is returned. If the value returned is *n* or more, then the contents of *dest* are unchanged.

Since the SPE C library only supports the “C” local, this function is equivalent to:

```
size_t len;
len = strlen(src);
if (n > len) (void)memcpy((void *)dest, (void *)src, n);
return len;
```

Dependencies

memcpy on page 39
strlen on page 50

See Also

strcmp on page 47

4.1.50 tolower

C Specification

```
#include <ctype.h>
int tolower(int c)
```

```
#include <tolower.h>
inline int _tolower(int c)
```

Description

The *tolower* subroutine convert the letter *c* to lower case, if possible.

4.1.51 toupper

C Specification

```
#include <ctype.h>
int toupper(int c)
```

```
#include <toupper.h>
inline int _toupper(int c)
```

Description

The *toupper* subroutine convert the letter *c* to upper case, if possible.

4.2 PPE Serviced SPE C Library Functions

This section documents the SPE C library functions that are serviced by the PPE control plane processor. These functions have been provided to improved SPE functionality and should not be used for high performance applications.

These subroutines all operate in the same basic fashion.

1. The SPE constructs a local store image of the input and outout parameters.
2. The SPE creates a 32-bit message consisting of an opcode and pointer to the local store parameter image array.
3. The SPE executes a Stop and Signal instruction.
4. The PPE detects the Stop and Signal.
5. The PPE invoke a specialist to service the request according to the message opcode.
6. The PPE services the requested function.
7. The PPE returns results in the local store image array.
8. The PPE resumes SPE execution.
9. The SPE returns that results back to the caller.

The PPE services to support the standard function documented in this section have been incorporated in the SPE runtime library - libspe. As such, no special PPE programming need be done in order to facilitate these functions.

4.2.1 Assisted C99 Subroutines

The following C99 (ISO/IEC 9899:1999) subroutines have been supported. Consult the standard for complete functional descriptions.

Function	**Syntax**	***Description***
clearerr	void clearerr(FILE *stream)	Clear end-of-file and error indicators
fclose	int fclose(FILE *stream)	Close a stream
feof	int feof(FILE *stream)	Test for end-of-file on stream
ferror	int ferror(FILE *steam)	Test for error on stream
fflush	int fflush(FILE *stream)	Flush a stream
fgetc getc getchar	int fgetc(FILE *stream) int getc(FILE *stream) int getchar(void)	Read character from stream
fgetpos	int fgetpos(FILE *stream, fpos_t *pos)	Get current file position
fgets gets	char *fgets(char *s, int size, FILE *stream) char *gets(char *s)	Read string from stream
fopen	FILE *fopen(const char *path, const char (*mod))	Open a stream
fprintf printf snprintf sprintf vfprintf vsprintf vsnprintf	int fprintf(FILE *stream, const char *format, ...) int printf(const char *format, ...) int snprintf(char *str, size_t size, const char *format ...) int sprintf(char *str, const char *format, ...) int vfprintf(FILE *stream, const char *format, va_list ap) int vsprintf(char *str, const char *format, va_list ap) int vsnprintf(char *str, size_t size, const char *format, va_list ap)	Formated printf

Function	Syntax	*Description*
fputc putc putchar	int fputc(int c, FILE *stream) int putc(int c, FILE *stream) int putchar(int c)	Write a character to stream
fputs puts	int fputs(const char *s, FILE *stream) int puts(const char *s)	Write a string to stream
fread	size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)	Binary stream read
freopen	FILE *freopen(const char *path, const char *mode, FILE *stream)	Re-open a stream
fscanf scanf sscanf vscanf vsscanf vfscanff	int fscanf(FILE *stream, const char *format, ...) int scanf(const char *format, ...) int sscanf(const char *str, const char *format, ...) int vscanf(const char *format, va_list ap) int vsscanf(const char *src, const char *format, va_list ap) int vfscanf(FILE *stream, const char *format, va_list ap)	Stream format conversion
fseek	int fseek(FILE *stream, long offset, int whence)	Position a stream
fsetpos	int fsetpos(FILE *stream, fpos_t *pos)	Set current file position
ftell	long ftell(FILE *stream)	Get current stream position
fwrite	size_t fwrite(const void *ptr, size_t, size_t nmemb, FILE *stream)	Binary stream write
perror	void perror(const char *s)	Print a system error message
remove	int remove(const char *pathname)	Remove a file
rename	int rename(const char *oldpath, const char *newpath)	Rename file
rewind	void rewind(FILE *stream)	Rewind a stream
setbuf setvbuf	void setbuf(FILE *stream, char *buf) int setvbuf(FILE *stream, char *buf, int mode, size_t size)	Set stream buffering
system	int system(const char *string)	Execute a shell command
tmpfile	FILE *tmpfile(void)	Create temporary file
tmpnam	char *tmpnam(char *s)	Create a name for a temporary file
ungetc	int ungetc(int c, FILE *stream)	Put character back on stream

4.2.2 Assisted POSIX Subroutines

The following POSIX.1 subroutines have been supported. Consult the standard for complete functional descriptions.

Some of these subroutines accept or return system memory pointers, as apposed to local store pointers. These subroutines has *_ea* appended to their names and the system memory pointer is declared of type eaddr_t, which is defined to be a unsigned long long..

Function	Syntax	*Description*
adjtimex	int adjtimex(struct timex *buf)	Tune kernel clock
close	int close(int fd)	Close a file
creat	int creat(const char *pathname, mode_t mode)	Create a file
ftok	key_t ftok(const char *pathnane, int proj_id)	Convert path and identifier to a key
getpagesize	int getpagesize(void)	Get memory page size
gettimeofday	int gettimeofday(struct timeval *tv, struct timezone *tz)	Get time of day

Function	Syntax	*Description*
kill	int kill(pid_t pid, int sig)	Send signal to a process
lseek	off_t lseek(int filedes, off_t offset, int whence)	Reposition read/write file offset
mmap_ea	eaddr_t mmap_ea(eaddr_t start, size_t length, int prot, int flags, int fd, off_t offset)	Map file into memory
mremap_ea	eaddr_t mremap_ea(eaddr_t old_address, size_t old_size, size_t new_size, unsigned long flags)	Remap a virtual memory address
msync_ea	int msync_ea(eaddr_t start, size_t length, int flags)	Synchronize a file with a memory map
munmap_ea	int munmap_ea(eaddr_t start, size_t length)	Unmap file from memory
open	int open(const char *pathname, int flags, mode_t mode)	Open a file
read	size_t read(int fd, void *buf, size_t count)	Read from a file
shm_open	void *shm_open(const char *name, int oflag, mode_t mode)	Open a shared memory objects
shm_unlink	int shm_unlink(const char *name)	Remove shared memory object
shmat_ea	void *shmat(int shmid, eaddr_t shmaddr, int shmflg)	Shared memory attach
shmctl_ea	ubt shmctl(int shmid, int cmd, eaddr_t buf)	Shared memory control
shmdt_ea	int shmdt(eaddr_t shmaddr)	Shared memory detach
shmget	int shmget(key_t key, int size, int shmflg)	Allocate a shared memory segment
stat fstat lstat	int stat(const char *filename, struct stat *buf) int fstat(int filedes, struct stat *buf) int lstat(char char *filename, struct stat *buf)	Get file status
unlink	int unlink(const char *pathname)	Delete a file system name
wait waitpid	pid_t wait(int *status) pid_t waitpid(pid_t pid, int *status, int options)	Wait for process termination
write	size_t write(int fd, const void *buf, size_t count)	Write to a file

4.3 SPE Local Storage Memory Allocation

The SPE memory allocation package implements a simple *malloc* interface that allows SPE programs to dynamically allocate memory from a "heap" region of local storage memory.

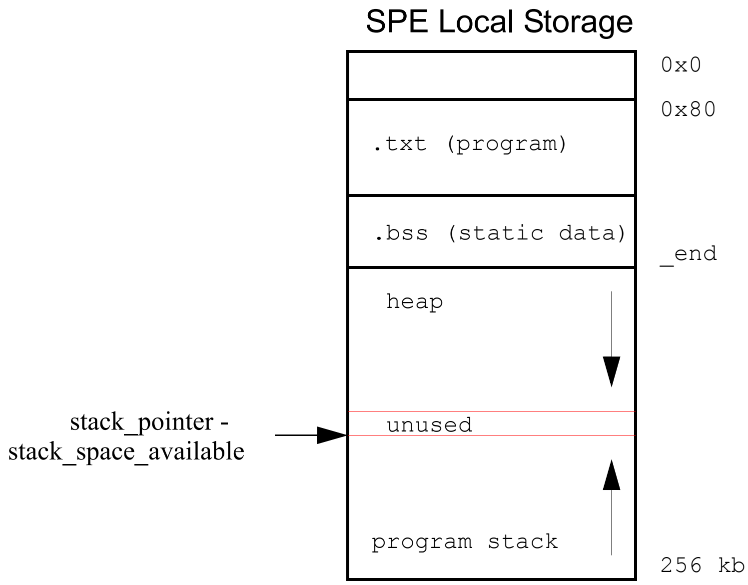


Figure 1: SPE Local Storage Memory Map

Implementation Details

The end location of the static data section (.bss) is referenced using the externally defined *_end* symbol, which is resolved at link time. Heap addresses are assumed to increase from *_end*, while stack addresses decrease from the end of local store memory (256 kb). The unused guard band is a 128 byte region that lies between the heap and stack.

The local store memory heap is initialized the first time a memory heap allocation routine is called. If there is no space available for the heap to be allocated, then the initialization fails and the heap is allocation is deferred until another heap allocation routine is called and there is heap space available. The heap space is established either during runtime initialization or extended using *brk* or *sbrk*.

The SPE malloc service internally divides the heap space into fixed size blocks of **4 KB** (*Programmer Note*: The internal block size can be reset at compile time by defining **BLOCKSIZE** to be any even power of 2 between 2048 and 16384).

Allocation requests that are greater than or equal to **BLOCKSIZE** can be satisfied by concatenating one or more contiguous blocks of memory. Smaller allocation requests can be satisfied by breaking a block into smaller fragments. Each block can only contain fragments of one size. All fragments are power of 2. Because of the quadword alignment rule, the smallest size of fragment is **16 bytes** (one can't allocate anything smaller than 16 bytes) .

Each block of memory has an information structure (header) associated with it. If the block is free, then the header has pointers to the next and previous free clusters of memory. If the block is busy and broken into fragments, then the header has information about the size of the fragments, number of free fragments and pointer to the first free fragment of this block. If this block is allocated as a whole, this header stores the number of blocks in this cluster. The size of each structure should be less than 16 bytes, however, because of the quadword alignment rule, we're reserving 16 bytes for each info struct.

In addition to keeping information about each block of memory, for each type of fragment (there should be about 8 different types: 16, 32, 64, 128, 256, 512, 1024, 2048), we also keep a linked list of all the free fragments for that size. Since the free fragments are not used, the linked list has pointers directly to the free fragments.

Once a fragment is freed, it's put on the free fragment list of a particular size and the free fragment counter for the block (in the block header) is decremented. If this is the last busy fragment in the block, the free fragments of the block are unlinked from the free fragment list and the block is available for other allocation request again.

4.3.1 brk

C Specification (SPE only)

```
#include <libmem.h>
int brk(void *end_data_segment)
```

Description

The *brk* subroutine sets the end of the data segment to the value specified by *end_data_segment* parameter.

If the *end_data_segment* value is reasonanble (i.e, the system has ample memory and does not exceed its max data size), the *brk* returns 0.

Note: Care should be taken when growing the memory heap. A large heap can severely restrict the available stack space limiting future subroutine call depth.

Dependencies

See Also

malloc on page 75
sbrk on page 77

4.3.2 calloc

C Specification (SPE only)

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size)
```

Description

The *calloc* subroutine attempts to allocate at least *size* bytes from local store memory heap.

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

Dependencies

See Also

free on page 74
malloc on page 75
calloc on page 73

4.3.3 free

C Specification (SPE only)

```
#include <stdlib.h>
void free(void *ptr)
```

Description

The *free* subroutine deallocates a block of local store memory previously allocated with *malloc, calloc, or realloc*. The memory to be freed is pointed to by *ptr*. If *ptr* is NULL, then no operation is performed.

Dependencies

See Also

calloc on page 73
malloc on page 75
realloc on page 76

4.3.4 malloc

C Specification (SPE only)

```
#include <stdlib.h>
void *malloc(unsigned int size)
```

Description

The *malloc* subroutine attempts to allocate at least *size* bytes from local store memory heap.

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

Dependencies

See Also

calloc on page 73
free on page 74
realloc on page 76

4.3.5 realloc

C Specification (SPE only)

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size)
```

Description

The re*malloc* subroutine changes the size of the memory block pointed to by *ptr* to *size* bytes. 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*(*size*). If *size* is equal to 0, then the call is equivalent to *free*(*ptr*). Unless *ptr* is NULL, its must have been returned by an earlier call to *malloc*, *calloc*, or *realloc*.

Dependencies

See Also

calloc on page 73
free on page 74
malloc on page 75

4.3.6 sbrk

C Specification (SPE only)

```
#include <stdlib.h>
void * sbrk(ptrdiff_t increment)
```

Description

The *sbrk* subroutine increases the local store memory data space by the number of bytes specified by the *increment* parameter. This routine returns a pointer to the start of the new area. If the *sbk* is unsuccessful, then NULL is returned.

Calling *sbrk* with an *increment* of 0 returns the current location of the program break. It can be used to find the current location of the program break.

Note: Care should be taken when growing the memory heap. A large heap can severely restrict the available stack space limiting future subroutine call depth.

Dependencies

See Also

malloc on page 75
brk on page 72

5. Audio Resample Library

The audio resample library supports a variety of forms of audio resampling that include the following:

- Both monophonic and stereophonic audio data
- Unsigned short or floating-point samples (both input and output, independently)
- Single and double precision (hiprec) computation

Note: This library is supported on both the PPE and SPE. However, double precision (high precision) resampling is only supported on the SPE.

Name(s)

libaudio_resample.a

Header File(s)

<libaudio_resample.h>
<audio_types.h>

5.1 init_resample_struct

C Specification

```
#include <init_resample_struct.h>
inline void _init_resample_struct(resample_struct *rd, int log2_lobes)

#include <libaudio.h>
void init_resample_struct(resample_struct *rd, int log2_lobes)
```

Descriptions

The *init_resample_struct* subroutine initializes the audio resampling data structure used by the resample subroutines. The resampling data structure is specified by the *rd* parameter and is initialized according to the *log2_lobes* parameter. The *log2_lobes* is the log, base 2, of the number “lobes” of the filter that is used when resampling an input stream. *log2_lobes* must be at least 5, corresponding to 32 lobes, and no larger than 12, corresponding to 4096 lobes. Roughly speaking, one can expected a 40dB signal-to-noise ratio using 32 lobes, and a 70dB signal-to-noise ratio using 1024 lobes. Greater than 1024 lobes does not appreciably add to the accuracy of the resampled result.

Dependencies

sin on page 259

See Also

resample_mono on page 82
resample_mono_hiprec on page 84
resample_stereo on page 86

5.2 Resample Routines

The *resample* subroutines resample an audio input stream specified by the input parameter *sample_in* and stores the resampled output into the *samples_out* array. If the audio signal is monophonic, the data is packed in adjacent entries of the given size (either unsigned short or float). If the audio signal is stereophonic, the data is interleaved left and right successive samples (L0, R0, L1, R1, L2, R2, etc.).

The input and output frequencies are encoded in the input parameters *cycle* and *freq_ratio*. The *cycle* is the denominator after reducing the fraction *frequency_in/frequency_out* to its lowest term. The *freq_ratio* is the floating point ratio *frequency_in/frequency_out*. For example, resampling CD quality (44.1 KHz) to DAT quality (48 KHz) has a frequency ratio of 44100/48000. This ratio reduces to 147/160. Therefore, the appropriate *cycle* and *freq_ratio* parameters are 160 and 0.91875, respectively.

The quality of the resample data is controlled by the resample data structure *rd*. The resample data is initialized according to the *log2_lobes* parameter specified in the *init_resample_struct* subroutine.

The input and output streams, *samples_in* and *samples_out* are coded as unsigned shorts or single precision floats depending upon the specific subroutine. As such, all samples in in the range 0 to 65535, even for floating-point inputs and outputs.

The *resample* subroutines uses the following filter function.

$$f(x) = \sin(\pi x)/(\pi x)$$

This function crosses the x axis at each non-zero integer, creating a series of positive and negative “lobes”. It runs from -infinity to +infinity and is clamped to zero outside the range [-*n*, *n*], where *n* is one half the number of “lobes”. For example, if *log2_lobes* is 7, the number of lobes is 128, and the non-zero region of the filter is [-64, 64].

Since the filter is 2**n* samples wide, the resampling algorithm requires *n* samples before in the input sample stream (*samples_in*), and n samples following the sample stream in order to construct the resampled output stream (*samples_out*). Failure to ensure that the *n* samples before and after the input stream are present can result in either unexpected results or even a protection exception.

The *sample_count* must be an integer multiple of *cycle* and specifies the number of output samples to be produced.

5.2.1 resample_mono

C Specification

```
#include <resample_mono.h>
inline void _resample_mono_F_F(int cycle, float freq_ratio,
                               float *samples_in, float *samples_out,
                               int sample_count, resample_struct *rd)

#include <resample_mono.h>
inline void _resample_mono_F_US(int cycle, float freq_ratio,
                               float *samples_in, unsigned short *samples_out,
                               int sample_count, resample_struct *rd)

#include <resample_mono.h>
inline void _resample_mono_US_F(int cycle, float freq_ratio,
                               unsigned short *samples_in, float *samples_out,
                               int sample_count, resample_struct *rd)

#include <resample_mono.h>
inline void _resample_mono_F_F(int cycle, float freq_ratio,
                               unsigned short *samples_in,
                               unsigned short *samples_out,
                               int sample_count, resample_struct *rd)


#include <libaudio.h>
void resample_mono_F_F(int cycle, float freq_ratio,
                               float *samples_in, float *samples_out,
                               int sample_count, resample_struct *rd)

#include <libaudio.h>
void resample_mono_F_US(int cycle, float freq_ratio,
                               float *samples_in, unsigned short *samples_out,
                               int sample_count, resample_struct *rd)

#include <libaudio.h>
void resample_mono_US_F(int cycle, float freq_ratio,
                               unsigned short *samples_in, float *samples_out,
                               int sample_count, resample_struct *rd)

#include <libaudio.h>
void resample_mono_F_F(int cycle, float freq_ratio,
                               unsigned short *samples_in,
                               unsigned short *samples_out, int sample_count,
                               resample_struct *rd)
```

Descriptions

The *resample_mono* subroutines resample a monophonic audio input stream specified by the input parameter *sample_in* and stores the resampled output into the *samples_out* array. Four combinations of input and output audio types are supported. Internal calculations are performed using single precision.

Subroutine	Input Type	Output Type
resample_mono_F_F	float	float
resample_mono_F_US	float	unsigned short
resample_mono_US_F	unsigned short	float
resample_mono_US_US	unsigned short	unsigned short

Dependencies

divide (floating point) on page 196
fabs on page 203
load_vec_unaligned on page 291
sin on page 259
sum_across_float on page 414

See Also

init_resample_struct on page 80
resample_mono_hiprec on page 84
resample_stereo on page 86

5.2.2 resample_mono_hiprec

C Specification

```
#include <resample_mono_hiprec.h>
inline void _resample_mono_F_F_hiprec(int cycle, float freq_ratio,
                                      float *samples_in, float *samples_out,
                                      int sample_count, resample_struct *rd)

#include <resample_mono_hiprec.h>
inline void _resample_mono_F_US_hiprec(int cycle, float freq_ratio,
                                      float *samples_in, unsigned short *samples_out,
                                      int sample_count, resample_struct *rd)

#include <resample_mono_hiprec.h>
inline void _resample_mono_US_F_hiprec(int cycle, float freq_ratio,
                                      unsigned short *samples_in, float *samples_out,
                                      int sample_count, resample_struct *rd)

#include <resample_mono_hiprec.h>
inline void _resample_mono_F_F_hiprec(int cycle, float freq_ratio,
                                      unsigned short *samples_in,
                                      unsigned short *samples_out, int sample_count,
                                      resample_struct *rd)


#include <libaudio.h>
void resample_mono_F_F_hiprec(int cycle, float freq_ratio,
                                      float *samples_in, float *samples_out,
                                      int sample_count, resample_struct *rd)

#include <libaudio.h>
void resample_mono_F_US_hiprec(int cycle, float freq_ratio,
                                      float *samples_in, unsigned short *samples_out,
                                      int sample_count, resample_struct *rd)

#include <libaudio.h>
void resample_mono_US_F_hiprec(int cycle, float freq_ratio,
                                      unsigned short *samples_in, float *samples_out,
                                      int sample_count, resample_struct *rd)

#include <libaudio.h>
void resample_mono_F_F_hiprec(int cycle, float freq_ratio,
                                      unsigned short *samples_in,
                                      unsigned short *samples_out, int sample_count,
                                      resample_struct *rd)
```

Descriptions

The *resample_mono_hiprec* subroutines resample a monophonic audio input stream specified by the input parameter *sample_in* and stores the resampled output into the *samples_out* array. Four combinations of input and output audio types are supported. Internal calculations are performed using IEEE double precision.

Subroutine	Input Type	Output Type
resample_mono_F_F_hiprec	float	float
resample_mono_F_US_hiprec	float	unsigned short
resample_mono_US_F_hiprec	unsigned short	float
resample_mono_US_US_hiprec	unsigned short	unsigned short

Note: These subroutines are only supported on the SPE.

Dependencies

divide (floating point) on page 196
fabs on page 203
load_vec_unaligned on page 291
sin on page 259
sum_across_float on page 414

See Also

init_resample_struct on page 80
resample_mono_hiprec on page 84
resample_stereo on page 86

5.2.3 resample_stereo

C Specification

```
#include <resample_stereo.h>
inline void _resample_stereo_F_F(int cycle, float freq_ratio,
                                 float *samples_in, float *samples_out,
                                 int sample_count, resample_struct *rd)

#include <resample_stereo.h>
inline void _resample_stereo_F_US(int cycle, float freq_ratio,
                                 float *samples_in,unsigned short *samples_out,
                                 int sample_count, resample_struct *rd)

#include <resample_stereo.h>
inline void _resample_stereo_US_F(int cycle, float freq_ratio,
                                 unsigned short *samples_in, float *samples_out,
                                 int sample_count, resample_struct *rd)

#include <resample_stereo.h>
inline void _resample_stereo_F_F(int cycle, float freq_ratio,
                                 unsigned short *samples_in,
                                 unsigned short *samples_out, int sample_count,
                                 resample_struct *rd)


#include <libaudio.h>
void resample_stereo_F_F(int cycle, float freq_ratio,
                                 float *samples_in, float *samples_out,
                                 int sample_count, resample_struct *rd)

#include <libaudio.h>
void resample_stereo_F_US(int cycle, float freq_ratio,
                                 float *samples_in, unsigned short *samples_out,
                                 int sample_count, resample_struct *rd)

#include <libaudio.h>
void resample_stereo_US_F(int cycle, float freq_ratio,
                                 unsigned short *samples_in, float *samples_out,
                                 int sample_count, resample_struct *rd)

#include <libaudio.h>
void resample_stereo_F_F(int cycle, float freq_ratio,
                                 unsigned short *samples_in,
                                 unsigned short *samples_out, int sample_count,
                                 resample_struct *rd)
```

Descriptions

The *resample_stereo* subroutines resample a stereophonic audio input stream specified by the input parameter *sample_in* and stores the resampled output into the *samples_out* array. Four combinations of input and output audio types are supported. Internal calculations are performed using single precision.

Subroutine	Input Type	Output Type
resample_stereo_F_F	float	float
resample_stereo_F_US	float	unsigned short
resample_stereo_US_F	unsigned short	float
resample_stereo_US_US	unsigned short	unsigned short

Dependencies

divide (floating point) on page 196
fabs on page 203
load_vec_unaligned on page 291
sin on page 259
sum_across_float on page 414

See Also

init_resample_struct on page 80
resample_mono_hiprec on page 84
resample_stereo on page 86

6. Curves and Surfaces Library

The curves and surfaces library consists of a set of support routines for evaluating curves and surfaces. At this time the only supported curves are quadratic and cubic Bezier curves. The only supported surfaces are biquadric and bicubic Bezier surfaces, and curved point-normal triangles.

This library is supported on both the PPE and SPE.

Name(s)

libsurface.a

Header File(s)

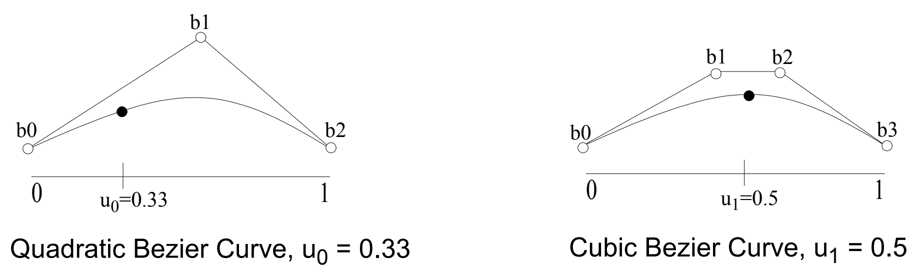
<libsurface.h>
<pn_types.h>
<cubic_fd_defs.h>
<btab_defs.h>

6.1 Quadratic & Cubic Bezier Curves

Quadratic and cubic Bezier curves are special cases of the Bezier spline function, and are implemented in this library due to their wide applicability in graphics and simulation.

Bezier curves are based on linear interpolation of control data points. Quadratic Bezier curves are described by a set of three Bezier control data points [$\mathbf{b_0}$,$\mathbf{b_1}$,$\mathbf{b_2}$] and may be evaluated at a parameterized **u** coordinate in the range [0..1]. Similarly, cubic Bezier curves are described by a set of four Bezier control data points [$\mathbf{b_0}$,$\mathbf{b_1}$,$\mathbf{b_2}$,$\mathbf{b_3}$] and may be evaluated at a parameterized **u** coordinate in the range [0..1].

Figure 6-1. Evaluation of Quadratic Bezier Curve and Cubic Bezier Curve



Some properties of Bezier curves include:

- Bezier curves are **affinely invariant**, which means that the following operations are equivelent: (1) affinely transform the control points, then evaluate the curve; (2) evaluate the curve, and then affinely transform the evaluated points.
- The Bezier control points [$\mathbf{b_0}$,$\mathbf{b_1}$,...,$\mathbf{b}_{n\mathbf{-1}}$] form a **convex bounding hull** for all parameter points **u** in the range of [0..1].
- The Bezier curve of degree ***n*** **interpolates** (passes through) the **end points** $\mathbf{b_0}$ and $\mathbf{b}_{n\mathbf{-1}}$.

Bezier curves can be directly evaluated in a variety of ways, including (but not limitted to): *Bernstein polynomials*, *de Casteljau's method*, and *forward differencing*. Each of these techniques are employed within this library, and each has implications on the performance and accuracy of the evaluation. Programmers should make their own decisions about the applicability of each method, but the following guidelines may be applied:

- Bernstein polynomials are a commonly used method for evaluating Bezier curves, and have simple (if inefficient) implementations for quadratic and cubic curves.
- *de Casteljau's method* for evaluating Bezier curves is considered to be numerically stable, as it combines a recursive sequence of linear interpolations.
- *forward differencing method* makes use of the Newton form of a polynomial to incrementally compute a sequence of points along the curve, and is usually several times faster than either de Casteljau or Bernstein evaluation. However, forward differencing propogates floating point round-off errors, which may be significant when the sampling frequency is high.

6.1.1 comp_cubic_bezier_coeffs_fd

C Specification

```
#include <comp_cubic_bezier_coeffs_fd.h>
inline void _comp_cubic_bezier_coeffs_fd(fdCoeffs *coeffs, int nsteps)

#include <libsurface.h>
void comp_cubic_bezier_coeffs_fd(fdCoeffs *coeffs, int nsteps)
```

Descriptions

The *comp_cubic_bezier_coeffs_fd* subroutine computes the coefficients necessary for evaluating a cubic Bezier curve or a bicubic Bezier surface using the forward differencing method. The coefficients are returned in the structure specified by the *coeffs* parameter.

The forward differencing coefficients may be precomputed once for each desired number of steps. The minimum step size is 1, while no upper limit is imposed. Due to floating point round off, applications should avoid using a large number of steps.

Dependencies

See Also

eval_cubic_bezier_curve_fd on page 95
eval_bicubic_bezier_surf_fd on page 108

6.1.2 eval_cubic_bezier_curve

C Specification

```
#include <eval_cubic_bezier_curve.h>
inline vector float _eval_cubic_bezier_curve(vector float u, vector float *P)

#include <libsurface.h>
vector float eval_cubic_bezier_curve(vector float u, vector float *P)
```

Descriptions

The *eval_cubic_bezier_curve* subroutine evaluates a cubic Bezier curve defined by the four control data points in the *P* array. The *P* array contains four component vector data. The curve is evaluated at the paramaterized *u* coordinate using the Bernstein polynomial method. A four component vector is returned.

Provided that the parameterized *u* coordinate is in the range [0..1], the result will lie on the cubic Bezier curve defined by *P*. For coordinates outside the range [0..1] the results are undefined.

Programmer Note

Each element in the *P* array represents a four component vector of the form as shown in *Figure 6-2*:

Figure 6-2. P Array Elements

x	**y**	**z**	**w**

To evaluate each vector component at the same parameterized *u* coordinate, the *u* coordinate should be replicated across its vector channels, as shown in the *Figure 6-3*:

Figure 6-3. U Coordinate Replication Across Vector Channels

u	**u**	**u**	**u**

To evaluate each vector component at a different paramaterized *u* coordinate, a different *u* value may be stored to each channel. This makes it possible to evaluate four curve points simultaneously. For more information, see *eval_cubic_bezier_curve_v* on page 96.

While vertex positions are most frequently used, other four component vector data including normals, colors and texture coordinates may be substituted.

Dependencies

See Also

eval_cubic_bezier_curve_v on page 96
eval_bicubic_bezier_surf on page 106

6.1.3 eval_cubic_bezier_curve_dc

C Specification

```
#include <eval_cubic_bezier_curve_dc.h>
inline vector float _eval_cubic_bezier_curve_dc(vector float u, vector float *P)

#include <libsurface.h>
vector float eval_cubic_bezier_curve_dc(vector float u, vector float *P)
```

Descriptions

The *eval_cubic_bezier_curve_dc* subroutine evaluates a cubic Bezier curve defined by the four control data points in the *P* array. The P array contains four component vector data. The curve is evaluated at the parameterized *u* coordinate using de Casteljau's method. A four component vector is returned.

Provided that the parameterized coordinate *u* is in the range [0..1], the result will lie on the cubic Bezier curve defined by *P*. For coordinates outside the range [0..1] the results are undefined.

Dependencies

See Also

eval_cubic_bezier_curve_dc_v on page 98
eval_bicubic_bezier_surf_dc_v on page 111

6.1.4 eval_cubic_bezier_curve_fd

C Specification

```
#include <eval_cubic_bezier_curve_fd.h>
inline void _eval_cubic_bezier_curve_fd(vector float *Q, vector float *P, fdCoeffs *coeffs)

#include <libsurface.h>
void eval_cubic_bezier_curve_fd(vector float *Q, vector float *P, fdCoeffs *coeffs)
```

Descriptions

The *eval_cubic_bezier_curve_fd* subroutine evaluates a cubic Bezier curve defined by the four control data points in the *P* array. The *P* array contains four component vector data. The curve is evaluated using the forward differencing method, and is sampled over the parameterized coordinate range [0..1]. The pre-calculated forward difference coefficients are specified by the *coeffs* parameter.

The resulting four component vectors are returned in the *Q* array. The number of vectors returned is *coeffs->nsteps+1*.

Dependencies

See Also

comp_cubic_bezier_coeffs_fd on page 91
eval_bicubic_bezier_surf_fd on page 108

6.1.5 eval_cubic_bezier_curve_v

C Specification

```
#include <eval_cubic_bezier_curve_v.h>
inline void _eval_cubic_bezier_curve_v(vector float *vx, vector float *vy,
                                       vector float *vz, vector float u,
                                       vector float *Px, vector float *Py,
                                       vector float *Pz)

#include <libsurface.h>
void eval_cubic_bezier_curve_v(vector float *vx, vector float *vy,
                                       vector float *vz, vector float u,
                                       vector float *Px, vector float *Py,
                                       vector float *Pz)
```

Descriptions

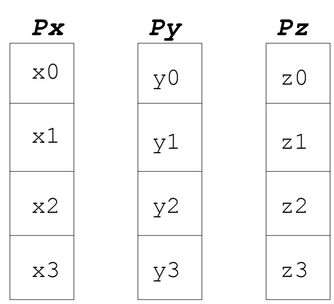
The *eval_cubic_bezier_curve_v* subroutine evaluates a cubic Bezier curve defined by the four control data points in the parallel arrays *Px*, *Py*, and *Pz*. The curve is evaluated at the paramaterized *u* coordinate using the Bernstein polynomial method. The resulting vectors are returned in *vx*, *vy*, and *vz*.

Provided that the parameterized *u* coordinate is in the range [0..1], the result will lie on the cubic Bezier curve defined by *Px, Py,* and *Pz*. For coordinates outside the range [0..1] the results are undefined.

Programmer Note

The four control data points that represent the cubic Bezier curve are stored in the parallel arrays *Px*, *Py*, and *Pz*. The vector components for the elements are stored separately as shown in *Figure 6-4* on page 96:

Figure 6-4. Parallel Array Storage - Vector Components



Information for four data elements is returned, again with the vector components separately stored stored in *vx*, *vy*, and *vz*.

While vertex positions are most frequently used, other three component vector data including normals, colors, and texture coordinates may be substituted.

Dependencies

See Also

eval_cubic_bezier_curve on page 92
eval_bicubic_bezier_surf_v on page 110

6.1.6 eval_cubic_bezier_curve_dc_v

C Specification

```
#include <eval_cubic_bezier_curve_dc_v.h>
inline void _eval_cubic_bezier_curve_dc_v(vector float *vx, vector float *vy,
                                          vector float *vz, vector float u,
                                          vector float *Px, vector float *Py,
                                          vector float *Pz)

#include <libsurface.h>
void eval_cubic_bezier_curve_dc_v(vector float *vx, vector float *vy,
                                  vector float *vz, vector float u,
                                  vector float *Px, vector float *Py,
                                  vector float *Pz)
```

Descriptions

The *eval_cubic_bezier_curve_dc_v* subroutine evaluates a cubic Bezier curve defined by the four control data points in the parallel arrays *Px*, *Py*, and *Pz*. The curve is evaluated at the paramaterized *u* coordinate using de Casteljau's method. The resulting vectors are returned in *vx*, *vy*, and *vz*.

Provided that the parameterized *u* coordinate is in the range [0..1], the result will lie on the cubic Bezier curve defined by *Px, Py,* and *Pz*. For coordinates outside the range [0..1] the results are undefined.

Dependencies

See Also

eval_cubic_bezier_curve_dc on page 94
eval_bicubic_bezier_surf_dc_v on page 111

6.1.7 eval_quadratic_bezier_curve

C Specification

```
#include <eval_quadratic_bezier_curve.h>
inline vector float _eval_quadratic_bezier_curve(vector float u, vector float *P)

#include <libsurface.h>
vector float eval_quadratic_bezier_curve(vector float u, vector float *P)
```

Descriptions

The *eval_quadratic_bezier_curve* subroutine evaluates a quadratic Bezier curve defined by the three control data points contained in the *P* array. The *P* array contains four component vector data. The curve is evaluated at the parameterized *u* coordinate using the Bernstein polynomial method. A four component vector is returned.

Provided that the parameterized *u* coordinate is in the range [0..1], the result will lie on the quadratic Bezier curve defined by *P*. For coordinates outside the range [0..1] the results are undefined.

Dependencies

See Also

eval_quadratic_bezier_curve_v on page 101
eval_bicubic_bezier_surf on page 106

6.1.8 eval_quadratic_bezier_curve_dc

C Specification

```
#include <eval_quadratic_bezier_curve_dc.h>
inline vector float _eval_quadratic_bezier_curve_dc(vector float u, vector float *P)

#include <libsurface.h>
vector float eval_quadratic_bezier_curve_dc(vector float u, vector float *P)
```

Descriptions

The *eval_quadratic_bezier_curve_dc* subroutine evaluates a quadratic Bezier curve defined by the three control data points in the *P* array. The P array contains four component vector data. The curve is evaluated at the paramaterized *u* coordinate using de Casteljau's method. A four component vector is returned.

Provided that the parameterized *u* coordinate is in the range [0..1], the result will lie on the quadratic Bezier curve defined by *P*. For coordinates outside the range [0..1] the results are undefined.

Dependencies

See Also

eval_quadratic_bezier_curve_dc_v on page 102
eval_bicubic_bezier_surf_dc on page 107

6.1.9 eval_quadratic_bezier_curve_v

C Specification

```
#include <eval_quadratic_bezier_curve_v.h>
inline vector float _eval_quadratic_bezier_curve_v(vector float *vx, vector float *vy,
                                                    vector float *vz, vector float u,
                                                    vector float *Px, vector float *Py,
                                                    vector float *Pz)

#include <libsurface.h>
vector float eval_quadratic_bezier_curve_v(vector float *vx, vector float *vy,
                                            vector float *vz, vector float u,
                                            vector float *Px, vector float *Py,
                                            vector float *Pz)
```

Descriptions

The *eval_quadratic_bezier_curve_v* subroutine evaluates a quadratic Bezier curve defined by the three control data points in the parallel arrays *Px, Py,* and *Pz.* The curve is evaluated at the paramaterized surface point *u* coordinate using the Bernstein polynomial method. The resulting vectors are returned in *vx*, *vy*, and *vz*.

Provided that the parameterized *u* coordinate is in the range [0..1], the result will lie on the quadratic Bezier curve defined by *Px, Py,* and *Pz*. For coordinates outside the range [0..1] the results are undefined.

Dependencies

See Also

eval_quadratic_bezier_curve on page 99
eval_biquadric_bezier_surf_v on page 114

6.1.10 eval_quadratic_bezier_curve_dc_v

C Specification

```
#include <eval_quadratic_bezier_curve_dc_v.h>
inline vector float _eval_quadratic_bezier_curve_dc_v(vector float *vx, vector float *vy,
                                                       vector float *vz, vector float u, vector float *Px,
                                                       vector float *Py, vector float *Pz)

#include <libsurface.h>
vector float eval_quadratic_bezier_curve_dc_v(vector float *vx, vector float *vy,
                                               vector float *vz, vector float u, vector float *Px,
                                               vector float *Py, vector float *Pz)
```

Descriptions

The *eval_quadratic_bezier_curve_dc_v* subroutine evaluates a quadratic Bezier curve defined by the three control data points in the parallel arrays *Px, Py,* and *Pz.* The curve is evaluated at the paramaterized *u* coordinate using de Casteljau's method. The resulting vectors are returned in *vx*, *vy*, and *vz*.

Provided that the parameterized *u* coordinate is in the range [0..1], the result will lie on the quadratic Bezier curve defined by *Px, Py,* and *Pz*. For coordinates outside the range [0..1] the results are undefined.

Dependencies

See Also

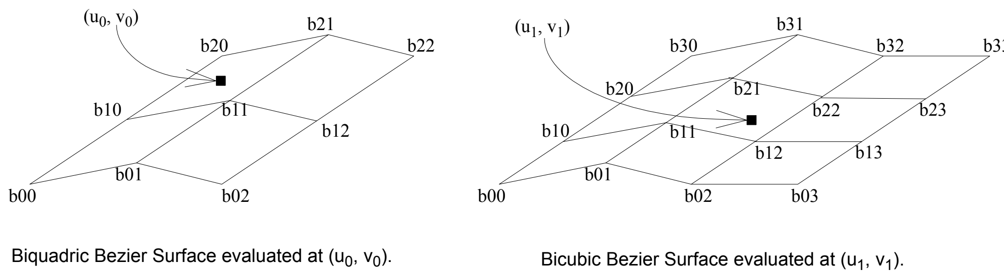
eval_quadratic_bezier_curve on page 99
eval_biquadric_bezier_surf_v on page 114

6.2 Biquadric & Bicubic Bezier Surfaces

Biquadric and bicubic Bezier surfaces are special cases of the tensor product patches, and are implemented in this library due to their wide applicability to graphics and simulation.

Where Bezier curves are based on linear interpolation of control points, Bezier surfaces are based on bi-linear interpolation of a control patch. Biquadric Bezier surfaces are described by a 3x3 patch of control data points, and may be evaluated at a parameterized **(u,v)** coordinate in the range of $[0..1]^2$. Similarly, bicubic Bezier surfaces are described by a 4x4 patch of control data points, and may be evaluated at a parameterized **(u,v)** coordinate in the range of $[0..1]^2$.

Figure 6-5. Biquadric and Bicubic Bezier Surface Evaluation

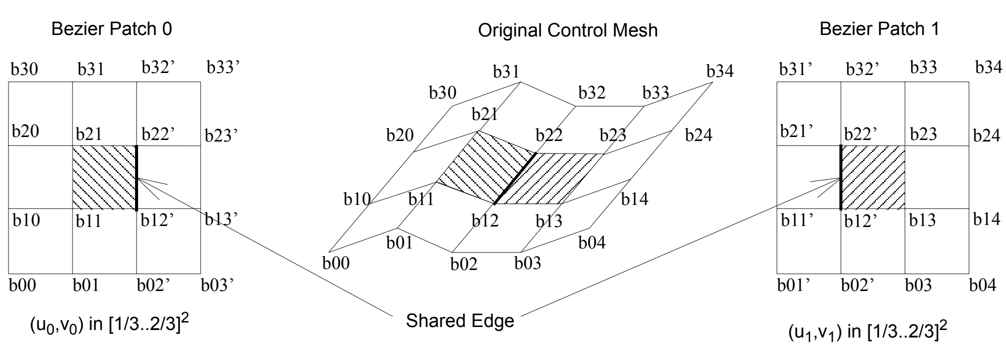


As with curves, Bezier surfaces are **affinely invariant**, **interpolate** their **corner points**, and lie within the **convex bounding hull** formed by the control points.

An important topic with surfaces is continuity, specifically geometric continuity (**G**) and tangent plane continuity (**C**). The degree of geometric continuity ($\mathbf{G}^N$) indicates how many times the function describing a surface can be differentiated, while the degree of tangent plane continuity ($\mathbf{C}^N$) indicates how many times the function describing the tangent planes can be differentiated. In general, $\mathbf{G}^N = \mathbf{C}^{N-1}$, which should come as no surprise as the tangent planes are computed by evaluating a surface's first order partial derivitaves. Biquadric Bezier surfaces have $\mathbf{G}^2$ and $\mathbf{C}^1$ continuity, while bicubic Bezier surfaces have $\mathbf{G}^3$ and $\mathbf{C}^2$ continuity.

Bezier evaluation forms the basis for certain types of approximating subdivision. Through knot insertion, bicubic Bezier patches can be made to match the evaluation rules for Catmull-Clark subdivision along the shared edge of two adjacent patches. This fact can be exploited to independently evaluate portions of two adjacent bicubic Bezier patches, while maintaining the assurance of geometric and tangent plane continuity.

Figure 6-6. Bezier Patch Evaluation



As illustrated in *Figure 6-6* on page 104, two knot insertions along each row are performed in order to maintain $\mathbf{C}^2$ continuity. A single knot insertion along each row can be performed if $\mathbf{C}^1$ continuity is sufficient. See *Curves and Surfaces for CAGD, a Practical Guide* (5th ed) for more information.

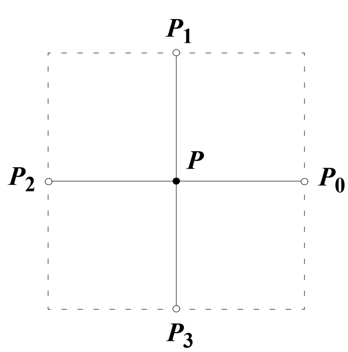
For regular valence meshes (such as terrain maps), direct Bezier evaluation has two important advantages over general purpose subdivision. First, the 1-neighborhood information can be implied from the mesh, without requiring additional storage overhead. Second, the Bezier patch can be efficiently evaluated at the desired sampling frequency, where most general purpose subdivision schemes are implemented with recursion and require additional storage for intermediate subdivision levels.

Bezier curves and surfaces are typically sampled at regular intervals, as indicated by the desired number of sampling points (**du** = **dv** = **1/*nsteps***, for instance). When the end points are taken into account, such a sampling actually produces ***nsteps***+1 points along a curve, or $(\textbf{\textit{nsteps}}+1)^2$ points on a surface. Face splitting subdivision schemes, on the other hand, divide edges $2^{\boldsymbol{n}}-1$ times, where ***n*** indicates the desired level of subdivison,. And when the end points are taken into account, face splitting subdivision produces $2^{\boldsymbol{n}}+1$ points along an edge, or $(2^{\boldsymbol{n}}+1)^2$ points on a surface. Provided that ***nsteps*** = $2^{\boldsymbol{n}}$ for integer values of ***n***, it follows that the number of Bezier sampling points will be consistent with the number of points produced by a face splitting subdivision scheme like Catmull-Clark.

Another important topic for surfaces is generation of **surface normals**. With Bezier surfaces, a number of techniques can be used to derive surface normals at a parameterized (**u,v**) coordinate, including:

1. If the surface normals for the control vertex positions are known, then the normals themselves can be smoothly interpolated using biquadric or bicubic Bezier interpolation on the normal data. Of course a final renormalization step is required to ensure that the resulting surface normals are of unit length.
2. A surface normal can be approximated at a position (**u,v**) by evaluating the surface locally at positions [(**u+du, v**),(**u,v+dv**)], assuming reasonably small values for **du** and **dv**. The differential positions can be used to form a cross product and thus determine an approximation to the local surface normal. This step could be performed for just the control vertex positions, while surface normals for the intermediate positions could be interpolated using the method described in 1) above.
3. Given the 1-neighborhood of a vertex in a mesh, it is possible to construct a local surface normal using the cross product of the two tangent plane masks:

Figure 6-7. Construction of a Local Surface Normal (Using the Cross Product of Two Tangent Plane Masks)



$$t1(P) = \sum \boldsymbol{P}_{\mathbf{i}} * \cos(2*\pi*\mathbf{i})/4$$

$$t2(P) = \sum \boldsymbol{P}_{\mathbf{i}} * \sin(2*\pi*\mathbf{i})/4$$

$$\text{norm}(P) = \frac{t1(P) \times t2(P)}{|t1(P) \times t2(P)|}$$

Again, this step could be performed for the control vertex positions, while surface normals for the intermediate positions can be interpolated using the method described in 1) above.

4. By modifying the coefficients used in the Bernstein polynomial evaluation, partial derivitives can be computed for Bezier curves, instead of vertex positions. As before, surface normals for intermediate positions can be interpolated from normals computed for the control vertices.

6.2.1 eval_bicubic_bezier_surf

C Specification

```
#include <eval_bicubic_bezier_surf.h>
inline vector float _eval_bicubic_bezier_surf(vector float u, vector float v,
                                              vector float *P)

#include <libsurface.h>
vector float eval_bicubic_bezier_surf(vector float u, vector float v, vector float *P)
```

Descriptions

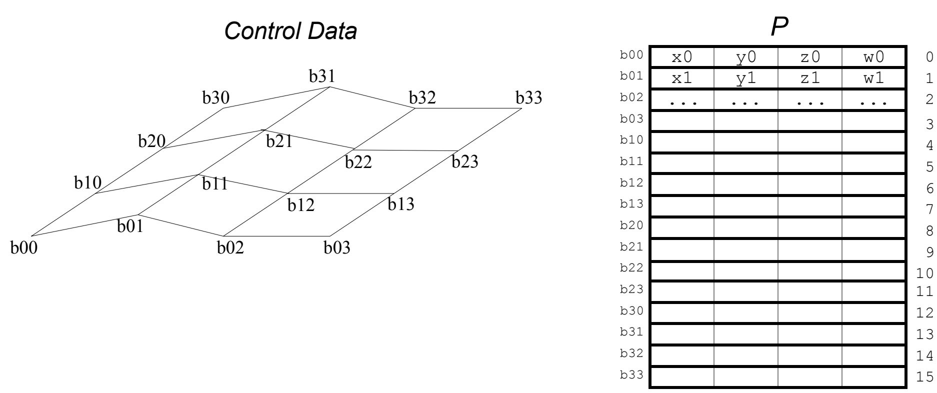
The *eval_bicubic_bezier_surf* subroutine evaluates a bicubic Bezier surface defined by the sixteen control data points in the *P* array. The *P* array contains four component vector data. The surface is evaluated at the paramaterized (u,v) coordinate using the Bernstein polynomial method. A four component vector is returned.

Provided that the parameterized (u,v) coordinate is in the range $[0..1]^2$, the result will lie on the bicubic Bezier surface defined by *P*. For coordinates outside the range $[0..1]^2$ the results are undefined.

Programmer Note

The sixteen control data points in the P array are arranged in memory as shown in *Figure 6-8*:

Figure 6-8. Data Control Points in the P Array



While vertex positions are most frequently used, other four component vector data including normals, colors, and texture coordinates may be substituted.

Dependencies

See Also

eval_bicubic_bezier_surf_v on page 110
eval_cubic_bezier_curve on page 92

6.2.2 eval_bicubic_bezier_surf_dc

C Specification

```
#include <eval_bicubic_bezier_surf_dc.h>
inline vector float _eval_bicubic_bezier_surf_dc(vector float u, vector float v,
                                                 vector float *P)

#include <libsurface.h>
vector float eval_bicubic_bezier_surf_dc(vector float u, vector float v, vector float *P)
```

Descriptions

The *eval_bicubic_bezier_surf_dc* subroutine evaluates a bicubic Bezier surface defined by the sixteen control data points in the *P* array. The *P* array contains four component vector data. The surface is evaluated at the paramaterized (*u,v*) coordinate using de Casteljau's method. A four component vector is returned.

Provided that the parameterized (*u,v*) coordinate is in the range $[0..1]^2$, the result will lie on the bicubic Bezier surface defined by *P*. For coordinates outside the range $[0..1]^2$ the results are undefined.

Dependencies

See Also

eval_bicubic_bezier_surf_dc_v on page 111
eval_cubic_bezier_curve_dc on page 94

6.2.3 eval_bicubic_bezier_surf_fd

C Specification

```
#include <eval_bicubic_bezier_surf_fd.h>
inline vector float _eval_bicubic_bezier_surf_fd(vector float *Q, vector float *P,
                                                 fdCoeffs *coeffs)

#include <libsurface.h>
vector float eval_bicubic_bezier_surf_fd(vector float *Q, vector float *P, fdCoeffs *coeffs)
```

Descriptions

The *eval_bicubic_bezier_surf_fd* subroutine evaluates a bicubic Bezier surface defined by the sixteen control data points in the *P* array. The *P* array contains four component vector data. The curve is evaluated using the forward differencing method, and is sampled over the parameterized coordinate range $[0..1]^2$. The pre-calculated forward difference coefficients are specified by the *coeffs* parameter.

The resulting four component vectors are returned in the *Q* array. The number of vectors returned is *(coeffs->nsteps+1)*(coeffs->nsteps+1)*.

Dependencies

See Also

eval_cubic_bezier_curve_fd on page 95
comp_cubic_bezier_coeffs_fd on page 91

6.2.4 eval_bicubic_bezier_surfnorm_fd

C Specification

```
#include <eval_bicubic_bezier_surfnorm_fd.h>
inline vector float _eval_bicubic_bezier_surfnorm_fd(vector float *Q, vector float *P,
                                                      fdCoeffs *coeffs)

#include <libsurface.h>
vector float eval_bicubic_bezier_surfnorm_fd(vector float *Q, vector float *P,
                                              fdCoeffs *coeffs)
```

Descriptions

The *eval_bicubic_bezier_surfnorm_fd* subroutine is identical to the *eval_bicubic_bezier_surf_fd* routine except that a re-normalization step is performed inside the forward differencing loop. The resulting three component vectors are of unit length; the fourth component is ignored during the re-normalization step.

The resulting unit vectors are returned in the *Q* array. The number of vectors returned is *(coeffs->nsteps+1)*(coeffs->nsteps+1)*.

Dependencies

See Also

eval_bicubic_bezier_surf_fd on page 108
comp_cubic_bezier_coeffs_fd on page 91

6.2.5 eval_bicubic_bezier_surf_v

C Specification

```
#include <eval_bicubic_bezier_surf_v.h>
inline void _eval_bicubic_bezier_surf_v(vector float *vx, vector float *vy, vector float *vz,
                                        vector float u, vector float v, vector float *Px,
                                        vector float *Py, vector float *Pz)
```

```
#include <libsurface.h>
void eval_bicubic_bezier_surf_v(vector float *vx, vector float *vy, vector float *vz,
                                vector float u, vector float v, vector float *Px,
                                vector float *Py, vector float *Pz)
```

Descriptions

The *eval_bicubic_bezier_surf_v* subroutine evaluates a bicubic Bezier surface defined by the sixteen control data points in the parallel arrays *Px*, *Py*, and *Pz*. The surface is evaluated at the paramaterized (*u*,*v*) coordinate using the Bernstein polynomial method. The resulting vectors are returned in *vx*, *vy*, and *vz*.

Provided that the parameterized (*u*,*v*) coordinate is in the range $[0..1]^2$, the result will lie on the bicubic Bezier surface defined by *Px, Py,* and *Pz*. For coordinates outside the range $[0..1]^2$ the results are undefined.

Dependencies

See Also

eval_bicubic_bezier_surf on page 106
eval_cubic_bezier_curve_v on page 96

6.2.6 eval_bicubic_bezier_surf_dc_v

C Specification

```
#include <eval_bicubic_bezier_surf_dc_v.h>
inline void _eval_bicubic_bezier_surf_dc_v(vector float *vx, vector float *vy,
                                           vector float *vz, vector float u, vector float *Px,
                                           vector float *Py, vector float *Pz)

#include <libsurface.h>
void eval_bicubic_bezier_surf_dc_v(vector float *vx, vector float *vy, vector float *vz,
                                   vector float u, vector float *Px, vector float *Py,
                                   vector float *Pz)
```

Descriptions

The *eval_bicubic_bezier_surf_dc_v* subroutine evaluates a bicubic Bezier surface defined by the sixteen control data points in the parallel arrays *Px*, *Py*, and *Pz*. The surface is evaluated at the paramaterized (u,v) coordinate using de Casteljau's method. The resulting vectors are returned in *vx*, *vy*, and *vz*.

Provided that the parameterized (u,v) coordinate is in the range $[0..1]^2$, the result will lie on the bicubic Bezier surface defined by *Px, Py,* and *Pz*. For coordinates outside the range $[0..1]^2$ the results are undefined.

Dependencies

See Also

eval_bicubic_bezier_surf_dc on page 107
eval_cubic_bezier_curve_dc_v on page 98

6.2.7 eval_biquadric_bezier_surf

C Specification

```
#include <eval_biquadric_bezier_surf.h>
inline vector float _eval_biquadric_bezier_surf(vector float u, vector float v,
                                                 vector float *P)

#include <libsurface.h>
vector float eval_biquadric_bezier_surf(vector float u, vector float v, vector float *P)
```

Descriptions

The eval_biquadric_bezier_surf subroutine evaluates a biquadric Bezier surface defined by the nine control data points in the P array. The P array contains four component vector data. The surface is evaluated at the paramaterized (u,v) coordinate using the Bernstein polynomial method. A four component vector is returned.

Provided that the parameterized (u,v) coordinate is in the range $[0..1]^2$, the result will lie on the biquadric Bezier surface defined by P. For coordinates outside the range $[0..1]^2$ the results are undefined.

Dependencies

See Also

eval_biquadric_bezier_surf_v on page 114
eval_quadratic_bezier_curve on page 99

6.2.8 eval_biquadric_bezier_surf_dc

C Specification

```
#include <eval_biquadric_bezier_surf_dc.h>
inline vector float _eval_biquadric_bezier_surf_dc(vector float u, vector float v,
                                                   vector float *P)

#include <libsurface.h>
vector float eval_biquadric_bezier_surf_dc(vector float u, vector float v, vector float *P)
```

Descriptions

The eval_biquadric_bezier_surf_dc subroutine evaluates a biquadric Bezier surface defined by the nine control data points in the P array. The P array contains four component vector data. The surface is evaluated at the paramaterized (u,v) coordinate using de Casteljau’s method. A four component vector is returned.

Provided that the parameterized (u,v) coordinate is in the range $[0..1]^2$, the result will lie on the biquadric Bezier surface defined by P. For coordinates outside the range $[0..1]^2$ the results are undefined.

Dependencies

See Also

eval_biquadric_bezier_surf_dc on page 113
eval_quadratic_bezier_curve_dc on page 100

6.2.9 eval_biquadric_bezier_surf_v

C Specification

```
#include <eval_biquadric_bezier_surf_v.h>
inline void _eval_biquadric_bezier_surf_v(vector float *vx, vector float *vy,
                                          vector float *vz, vector float u, vector float v,
                                          vector float *Px, vector float *Py, vector float *Pz)

#include <libsurface.h>
void eval_biquadric_bezier_surf_v(vector float *vx, vector float *vy, vector float *vz,
                                  vector float u, vector float v, vector float *Px,
                                  vector float *Py, vector float *Pz)
```

Descriptions

The *eval_biquadric_bezier_surf_v* subroutine evaluates a biquadric Bezier surface defined by the nine control data points in the parallel arrays *Px*, *Py*, and *Pz*. The surface is evaluated at the paramaterized (u,v) coordinate using the Bernstein polynomial method. The resulting vectors are returned in *vx*, *vy*, and *vz*.

Provided that the parameterized (u,v) coordinate is in the range $[0..1]^2$, the result will lie on the bicubic Bezier surface defined by *Px, Py,* and *Pz*. For coordinates outside the range $[0..1]^2$ the results are undefined.

Dependencies

See Also

eval_biquadric_bezier_surf on page 112
eval_quadratic_bezier_curve_v on page 101

6.2.10 eval_biquadric_bezier_surf_dc_v

C Specification

```
#include <eval_biquadric_bezier_surf_dc_v.h>
inline void _eval_biquadric_bezier_surf_dc_v(vector float *vx, vector float *vy,
                                             vector float *vz, vector float u, vector float v,
                                             vector float *Px, vector float *Py, vector float *Pz)

#include <libsurface.h>
void eval_biquadric_bezier_surf_dc_v(vector float *vx, vector float *vy, vector float *vz,
                                     vector float u, vector float v, vector float *Px,
                                     vector float *Py, vector float *Pz)
```

Descriptions

The *eval_biquadric_bezier_surf_dc_v* subroutine evaluates a biquadric Bezier surface defined by the nine control data points in the parallel arrays *Px*, *Py*, and *Pz*. The surface is evaluated at the paramaterized (*u*,*v*) coordinate using de Casteljau's method. The resulting vectors are returned in *vx*, *vy*, and *vz*.

Provided that the parameterized (*u*,*v*) coordinate is in the range $[0,1]^2$, the result will lie on the bicubic Bezier surface defined by *Px, Py,* and *Pz*. For coordinates outside the range $[0,1]^2$ the results are undefined.

Dependencies

See Also

eval_biquadric_bezier_surf_dc on page 113
eval_quadratic_bezier_curve_dc_v on page 102

6.3 Curved Point-Normal Triangles

Curved point-normal, known as P-N triangles and n-patches, are the current poor-man's subdivision surfaces. Flat triangles, consisting of three points and their respective normals, are subdivided into three-sided cubic Bezier patches with either quadratically or linearly varying normals for Gouraud shading. These curved point-normal triangles require minimal, or no changes to existing authoring tools or hardware designs. In exchange for this simplicity, P-N triangle surfaces suffer from the following limitations.

- Tangent plane continuity between adjacent triangles is not guaranteed. Therefore, in the limit, the surface may have artifacts with specular highlights, spotlights, or environment/cube-mapped textures.
- Only one normal per vertex. Therefore, extended rules for creases, corners, cones, etc. are not supported. This can result in unexpected subdivision and/or cracks in surfaces.
- Clipping P-N triangles before subdivision could be difficult. The geometry alone does not act as the bounding hull for the subdivided triangles.
- Animating a mesh can be tricky. High quality vertex normals must be generated before subdivision. Vertex blending would have to include normals and would further require a matrix/weight per neighbor. Alternatively, vertex normals must be regenerated after mesh animation.
- Wavelet technology is undefined for P-N triangles. Other techniques for adding surface peturbations (e.g., noise) also presents problems because they change the curvature of the surface and thus must be applied prior to subdivision.

A quality introduction to the mathematics of P-N triangles can be found in a paper entitled "Curved PN Triangles" by Alex Vlachos, Jorg Peters, Chas Boyd, and Jason Mitchell.

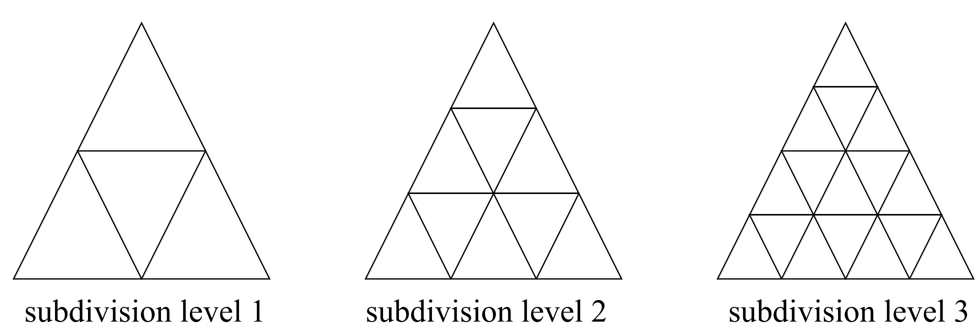
Producing Curved Surfaces

For each triangle, the process of rendering curved surfaces using the P-N triangle subroutines is as follows:

- Compute the coefficients of the geometry and each interpolated vertex datum. The geometry coefficients are computed using the *compute_cubic_pn_coeffs* subroutines. Normal coefficients are computed using the *compute_quadratic_pn_coeffs* subroutines (to produce quadratically varying normals) or the *compute_linear_pn_coeffs* subroutines (to produce linearly varying normals). All other interpolated vertex data (e.g., colors, textures, fog factors, etc...) coefficients can be computed using the *compute_linear_pn_coeffs* subroutines.
- Evaluate the subdivided vertices and vertex data using the *eval_cubic_pn_vtx* subroutines for the vertices, the *eval_quadratic_pn_vtx* subroutines for quadratically varying normals, and *eval_linear_pn_vtx* for linearly vertex data. Vertices are generally evaluated on regularly space intervals according to the subdivision level.

A representation of how this process renders curved surfaces is shown in *Figure 6-9* on page 117.

Figure 6-9. Example Rendering Curved Surfaces Using the P-N Triangle Subroutines



6.3.1 compute_cubic_pn_coeffs

C Specification

```
#include <compute_cubic_pn_coeffs.h>
inline void _compute_cubic_pn_coeffs(pnCubicCoeffs *coeffs,
                    vector float p1, vector float p2, vector float p3,
                    vector float n1, vector float n2, vector float n3)

#include <compute_cubic_pn_coeffs_v.h>
inline void _compute_cubic_pn_coeffs_v(pnCubicCoeffs_v *coeffs,
                    vector float p1X, vector float p1Y, vector float p1Z,
                    vector float p2X, vector float p2Y, vector float p2Z,
                    vector float p3X, vector float p3Y, vector float p3Z,
                    vector float n1X, vector float n1Y, vector float n1Z,
                    vector float n2X, vector float n2Y, vector float n2Z,
                    vector float n3X, vector float n3Y, vector float n3Z)

#include <libsurface.h>
void compute_cubic_pn_coeffs(pnCubicCoeffs *coeffs, vector float p1,
                    vector float p2, vector float p3, vector float n1,
                    vector float n2, vector float n3)

#include <libsurface.h>
void compute_cubic_pn_coeffs_v(pnCubicCoeffs_v *coeffs,
                    vector float p1X, vector float p1Y, vector float p1Z,
                    vector float p2X, vector float p2Y, vector float p2Z,
                    vector float p3X, vector float p3Y, vector float p3Z,
                    vector float n1X, vector float n1Y, vector float n1Z,
                    vector float n2X, vector float n2Y, vector float n2Z,
                    vector float n3X, vector float n3Y, vector float n3Z)
```

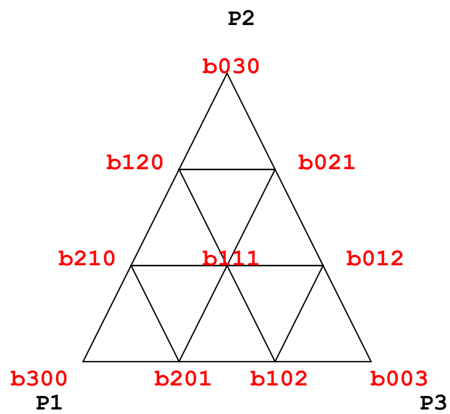
Descriptions

The *compute_cubic_pn_coeffs* subroutine computes the cubic bezier coefficients for a point-normal triangle defined by the vertices (points), *p1*, *p2* & *p3*, with unit normals, *n1*, *n2* & *n3*. The resulting coefficients are returned in the structure specified by the *coeffs* parameter. The coeficients are used when evaluating vertices of the patch using the *eval_cubic_pn_vtx* subroutine.

The *compute_cubic_pn_coeffs_v* subroutine computes the cubic bezier coefficients for a set of 4 point-normal triangles defined by the parallel array vertices (i.e., points), *p1X*, *p1Y*, *p1Z*, *p2X*, *p2Y*, *p2Z*, *p3X*, *p3Y* & *p3Z*, with unit normals, *n1X*, *n1Y*, *n1Z*, *n2X*, *n2Y*, *n2Z*, *n3X*, *n3Y* & *n3Z*. The resulting coefficients are returned in the structure specified by the *coeffs* parameter. The coeficients are used when evaluating vertices of the patch using *eval_cubic_pn_vtx_v* subroutine.

A pictorial representation (control net) of the coefficients (i.e., control points) of the triangular bezier patch is shown in *Figure 6-10* on page 119.

Figure 6-10. Example of a Control Net of the Coefficients of a Triangular Bezier Patch



Dependencies

dot_product on page 400

See Also

compute_linear_pn_coeffs on page 120
compute_quadratic_pn_coeffs on page 121
eval_cubic_pn_vtx on page 123

6.3.2 compute_linear_pn_coeffs

C Specification

```
#include <compute_linear_pn_coeffs.h>
inline void _compute_linear_pn_coeffs(pnLinearCoeffs *coeffs,
                                      vector float v1, vector float v2, vector float v3)

#include <compute_linear_pn_coeffs_v.h>
inline void _compute_cubic_pn_coeffs_v(pnCubicCoeffs_v *coeffs,
                                      vector float v1X, vector float v1Y, vector float v1Z,
                                      vector float v2X, vector float v2Y, vector float v2Z,
                                      vector float v3X, vector float v3Y, vector float v3Z)

#include <libsurface.h>
void compute_linear_pn_coeffs(pnLinearCoeffs *coeffs, vector float v1,
                                      vector float v2, vector float v3)

#include <libsurface.h>
void compute_linear_pn_coeffs_v(pnLinearCoeffs_v *coeffs,
                                      vector float v1X, vector float v1Y, vector float v1Z,
                                      vector float v2X, vector float v2Y, vector float v2Z,
                                      vector float v3X, vector float v3Y, vector float v3Z)
```

Descriptions

The *compute_linear_pn_coeffs* subroutine computes the linear coefficients for a point-normal triangle defined by the 4-component vertex data *v1*, *v2*, & *v3*. The resulting coefficients are returned in the structure specified by the *coeffs* parameter. This subroutine is suitable for any vertex data to be linearly computed across the P-N triangle bezier patch. This includes colors, normals, textures, etc... The coeficients are used when evaluating vertex data of the patch using the *eval_linear_pn_vtx* subroutine.

The compute_linear_pn_coeffs_v subroutine computes the linear coefficients for a set of 4 point-normal triangle 3-component vertex data defined by the parallel array vertices data *v1X*, *v1Y*, *v1Z*, *v2X*, *v2Y*, *v2Z*, *v3X*, *v3Y* & *v3Z*. The resultingcoefficients are returned in the structure specified by the *coeffs* parameter. This subroutine is suitable for any 3-D vertex data to be linearly computed across the P-N triangle bezier patch. This includes colors, normals, textures, etc... The coeficients are used when evaluating vertex data of the patch using the *eval_linear_pn_vtx_v* subroutine.

Dependencies

See Also

compute_cubic_pn_coeffs on page 118
compute_quadratic_pn_coeffs on page 121
eval_linear_pn_vtx on page 124

6.3.3 compute_quadratic_pn_coeffs

C Specification

```
#include <compute_quadratic_pn_coeffs.h>
inline void _compute_quadratic_pn_coeffs(pnQuadraticCoeffs *coeffs,
                                          vector float p1, vector float p2, vector float p3,
                                          vector float n1, vector float n2, vector float n3)

#include <compute_quadratic_pn_coeffs_v.h>
inline void _compute_quadratic_pn_coeffs_v(pnQuadraticCoeffs_v *coeffs,
                                          vector float p1X, vector float p1Y, vector float p1Z,
                                          vector float p2X, vector float p2Y, vector float p2Z,
                                          vector float p3X, vector float p3Y, vector float p3Z,
                                          vector float n1X, vector float n1Y, vector float n1Z,
                                          vector float n2X, vector float n2Y, vector float n2Z,
                                          vector float n3X, vector float n3Y, vector float n3Z)

#include <libsurface.h>
void compute_quadratic_pn_coeffs(pnQuadraticCoeffs *coeffs,
                                          vector float p1, vector float p2, vector float p3,
                                          vector float n1, vector float n2, vector float n3)

#include <libsurface.h>
void compute_quadratic_pn_coeffs_v(pnQuadraticCoeffs_v *coeffs,
                                          vector float p1X, vector float p1Y, vector float p1Z,
                                          vector float p2X, vector float p2Y, vector float p2Z,
                                          vector float p3X, vector float p3Y, vector float p3Z,
                                          vector float n1X, vector float n1Y, vector float n1Z,
                                          vector float n2X, vector float n2Y, vector float n2Z,
                                          vector float n3X, vector float n3Y, vector float n3Z)
```

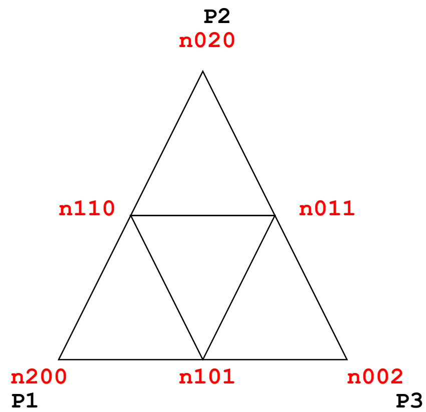
Descriptions

The *compute_quadratic_pn_coeffs* subroutine computes the quadratic normal coefficients for a point-normal triangle defined by the vertices (points), *p1*, *p2*, & *p3* with unit 3-D normals, *n1*, *n2*, & *n3*. The resulting coefficients are returned in the structure specified by the *coeffs* parameter. The coeficients are used when evaluating vertex normals of the patch using the *eval_quadratic_pn_vtx* subroutine.

The *compute_quadratic_pn_coeffs_v* subroutine computes the quadratic normal coefficients for a set of 4 point-normal triangles defined by the parallel array vertices (points), *p1X*, *p1Y*, *p1Z*, *p2X*, *p2Y*, *p2Z*, *p3X*, *p3Y* & *p3Z*, with 3-D unit normals, *n1X*, *n1Y*, *n1Z*, *n2X*, *n2Y*, *n2Z*, *n3X*, *n3Y* & *n3Z*. The resulting coefficients are returned in the structure specified by the *coeff* parameter. The coeficients are used when evaluating vertice normals of the patch using *eval_quadratic_pn_vtx_v* subroutine.

A pictorial representation (control net) of the coefficients (i.e., control points) of the triangular bezier patch is shown in *Figure 6-11* on page 122:

Figure 6-11. Example of a Control Net of the Coefficients of a Triangular Bezier Patch



Dependencies

divide (floating point) on page 196
dot_product on page 400
normalize on page 411

See Also

compute_linear_pn_coeffs on page 120
eval_cubic_pn_vtx on page 123
eval_quadratic_pn_vtx on page 125

6.3.4 eval_cubic_pn_vtx

C Specification

```
#include <eval_cubic_pn_vtx.h>
inline vector float _eval_cubic_pn_vtx(pnCubicCoeffs *coeffs, vector float w1,
                                       vector float w2, vector float w3)

#include <eval_cubic_pn_vtx_v.h>
inline void _eval_cubic_pn_vtx_v(vector float *vx, vector float *vy, vector float *vz, pnCubicCoeffs_v *coeffs,
vector float w1, vector float w2, vector float w3)

#include <libsurface.h>
vector float eval_cubic_pn_vtx(pnCubicCoeffs *coeffs, vector float w1, vector float w2,
                                       vector float w3)

#include <libsurface.h>
void eval_cubic_pn_vtx_v(vector float *vx, vector float *vy, vector float *vz,
                                       pnCubicCoeffs_v *coeffs, vector float w1,
                                       vector float w2, vector float w3)
```

Descriptions

The *eval_cubic_pn_vtx* subroutine evaluates a vertex of a cubic bezier P-N triangle patch corresponding to the baricentric coordinate *w1*, *w2*, *w3*. The coefficients of the triangle patch is specified by the *coeffs* parameter. The resulting vertex is returned as a packed 128-bit, floating-point vector.

The *eval_cubic_pn_vtx_v* subroutine evaluates a set of 4 vertices of 4 cubic bezier P-N triangle patches corresponding to the baricentric coordinate *w1*, *w2*, *w3*. The coefficients of the 4 triangle patches is specified by the *coeffs* parameter. The resulting vertices are returned in parallel array format in the memory pointed to by parameters *vx*, *vy*, and *vz*.

The baricentric coordinate (*w1*, *w2*, *w3*) are vertex weighting factors andtypically sum to 1.0 with each component of the weighting factors being equal.

Dependencies

See Also

compute_cubic_pn_coeffs on page 118
eval_linear_pn_vtx on page 124
eval_quadratic_pn_vtx on page 125

6.3.5 eval_linear_pn_vtx

C Specification

```
#include <eval_linear_pn_vtx.h>
inline vector float _eval_linear_pn_vtx(pnLinearCoeffs *coeffs, vector float w1,
                                        vector float w2, vector float w3)

#include <eval_linear_pn_vtx_v.h>
inline void _eval_linear_pn_vtx_v(vector float *vx, vector float *vy, vector float *vz,
                                        pnLinearCoeffs_v *coeffs, vector float w1,
                                        vector float w2, vector float w3)

#include <libsurface.h>
vector float eval_linear_pn_vtx(pnLinearCoeffs *coeffs, vector float w1, vector float w2,
                                        vector float w3)

#include <libsurface.h>
void eval_linear_pn_vtx_v(vector float *vx, vector float *vy, vector float *vz,
                                        pnLinearCoeffs_v *coeffs, vector float w1,
                                        vector float w2, vector float w3)
```

Descriptions

The *eval_linear_pn_vtx* subroutine evaulates linear interpolated vertex data of a P-N triangle patch corresponding to the baricentric coordinate *w1*, *w2*, & *w3*. The coefficients of the vertex data are specified by the *coeffs* parameter. The resulting vertex 4-D data is returned as a 128-bit, floating-point vector.

The *eval_linear_pn_vtx_v* evaulates a set of 4 linear interpolated vertex data of 4 cubic bezier P-N triangle patches corresponding to the baricentric coordinate w1, w2, w3. The coefficients of the 4 vertex datum is specified by the *coeffs* parameter. The resulting vertex data is returned in parallel array format in the memory pointed to by *vx*, *vy*, and *vz*.

The baricentric coordinate (*w1*, *w2*, *w3*) are vertex weighting factors and typically sum to 1.0 with each component of the weighting factors being equal.

Dependencies

See Also

compute_linear_pn_coeffs on page 120
eval_cubic_pn_vtx on page 123
eval_quadratic_pn_vtx on page 125

6.3.6 eval_quadratic_pn_vtx

C Specification

```
#include <eval_quadratic_pn_vtx.h>
inline vector float _eval_quadratic_pn_vtx(pnQuadraticCoeffs *coeffs, vector float w1,
                                           vector float w2, vector float w3)

#include <eval_quadratic_pn_vtx_v.h>
inline void _eval_quadratic_pn_vtx_v(vector float *nx, vector float *ny, vector float *nz,
                                     pnQuadraticCoeffs_v *coeffs, vector float w1,
                                     vector float w2, vector float w3)

#include <libsurface.h>
vector float eval_quadratic_pn_vtx(pnQuadraticCoeffs *coeffs, vector float w1,
                                   vector float w2, vector float w3)

#include <libsurface.h>
void eval_quadratic_pn_vtx_v(vector float *vx, vector float *vy, vector float *vz,
                             pnQuadraticCoeffs_v *coeffs, vector float w1,
                             vector float w2, vector float w3)
```

Descriptions

The *eval_quadratic_pn_vtx* subroutine evaluates a quadratically interpolated normal of a cubic bezier P-N triangle patch corresponding to the baricentric coordinate *w1*, *w2*, *w3*. The normal coefficients of the patch are specified by the *coeffs* parameter. The resulting non-unit normals is returned in the 3 most significant slots of a 128-bit, floating-point vector.

The *eval_quadratic_pn_vtx_v* subroutine evaluates a set of 4 quadratically interpolated normals of 4 cubic bezier P-N triangle patches corresponding to the baricentric coordinate *w1*, *w2*, *w3*. The normal coefficients are specified by the *coeffs* parameter. The resulting non-unit normals are returned in parallel array format to the memory pointed to by *nx*, *ny*, and *nz*.

The baricentric coordinate (*w1*, *w2*, *w3*) are vertex weighting factors and typically sum to 1.0 with each component of the weighting factors being equal.

Dependencies

See Also

compute_quadratic_pn_coeffs on page 121
eval_cubic_pn_vtx on page 123
eval_linear_pn_vtx on page 124

7. 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.a

Header File(s)

<libfft.h>

7.1 fft_1d_r2

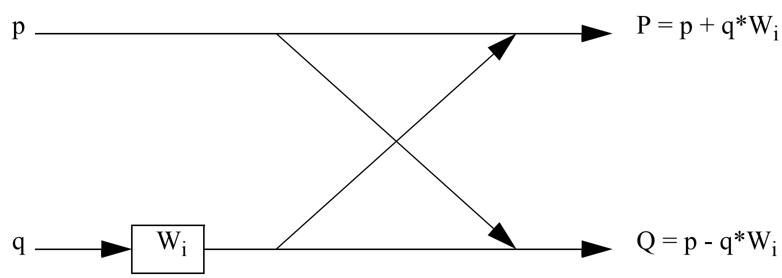
C Specification

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

#include <libfft.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.

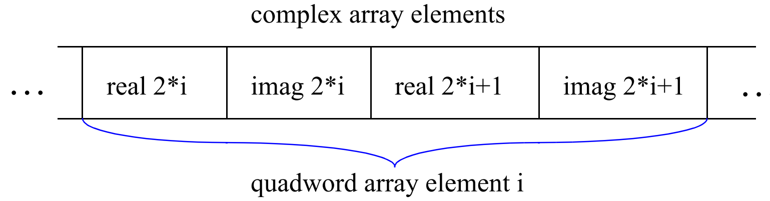
This routine requires the caller to provide pre-computed twiddle factors, *W*. *W* is an array of single-precision complex numbers of length $2^{(log2\_size-2)}$ entries and is computes are follows for forward (time domain to frequency domain):

```
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 130

7.2 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.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 282

See Also

init_fft_2d on page 132
fft_1d_r2 on page 128

7.3 init_fft_2d

C Specification

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

#include <libfft.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

cos on page 189
sin on page 259

See Also

fft_2d on page 130

8. 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>

8.1 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 139

8.2 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 138
unpack_color8 on page 143

8.3 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 144

8.4 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 145
pack_color8 on page 136

8.5 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 134

8.6 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

inverse on page 231

See Also

spec9 on page 142

8.7 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 141

8.8 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 136
unpack_rgba8 on page 145

8.9 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 137

8.10 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, $1.0\text{-}2^{23}$ is produced.

Dependencies

See Also

pack_rgba8 on page 138
unpack_color8 on page 143

9. 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>

9.1 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’).

9.1.1 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 150
conv3x3_4ub, conv5x5_4ub, conv7x7_4ub, conv9x9_4ub on page 152

9.1.2 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 148
conv3x3_4ub, conv5x5_4ub, conv7x7_4ub, conv9x9_4ub on page 152

9.1.3 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 even 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 even power of two (common for many blurring or averaging filters), the ‘scale’ and ‘shift’ values may be computed as follows:

```
scale = 2**log2(sum) * 65535 / sum
shift = 16 + 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 148
conv3x3_1us, conv5x5_1us, conv7x7_1us, conv9x9_1us on page 150

9.2 Histograms

9.2.1 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

10. 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>

10.1 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 157

10.2 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 156

10.3 lu2_decomp

C Specification

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

Description

The *lu2_decomp* 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 vector *ipiv* must be quadword aligned

This is the right-looking Level 2 BLAS version of the algorithm. This 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. This subroutine should not be used for general large square matrix since it is not very efficient. One should use subroutine *lu_decomp_3* instead.

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 156
scale_vector on page 169
swap_vectors on page 175
nmsub_number_vector on page 165

See Also

lu3_decomp_block on page 160

10.4 lu3_decomp_block

C Specification

```
#include <liblarge_matrix.h>
int lu3_decomp_block (int m, int n, float *A, int lda, int *ipiv)
```

Description

The *lu3_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 158
swap_matrix_rows on page 174

solve_unit_lower on page 171
nmsub_matrix_matrix on page 163

See Also

lu2_decomp on page 158

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*)

10.5 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. *a*, *b*, and *c* 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 column major order with a stride of *lda*
lda	stride of matrix *A*
B	an *n* by *p* matrix arranged in column major order with a stride of *ldb*
ldb	stride of matrix *B*
C	an *m* by *p* matrix arranged in column major order. On exit, the matrix *C* is overwritten by the resulting matrix

Dependencies

See Also

nmsub_matrix_matrix on page 163
madd_vector_matrix on page 166

10.6 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. *A*, *B*, and *C* 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 column major order with a stride of *lda*
lda	stride of matrix *A*
B	an *n* by *p* matrix arranged in column major order with a stride of *ldb*
ldb	stride of matrix B
C	an *m* by *p* matrix arranged in column 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 162
madd_vector_matrix on page 166

10.7 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 165
madd_vector_vector on page 167
madd_vector_matrix on page 166
scale_vector on page 169

10.8 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 164
nmsub_vector_vector on page 168
scale_vector on page 169

10.9 madd_vector_matrix

C Specification

```
#include <liblarge_matrix.h>
void madd_vector_matrix(int m, int n, float *A, int lda, float *x, float *y)
```

Description

The *madd_vector_matrix* subroutine performs the matrix-vector operation:

$$y = A*x + y$$

where *x* and *y* are vectors, and *A* is a matrix

Vectors *x* and y and matrix *A* must be quadword aligned; *m* and *n* must be multiples of 4.

Parameters

m	size of arrays *x* and *y*
n	size of arrays *x* and *y*
A	an *m* by *n* matrix arranged in column major order
x	*n*-element array
y	*n*-element array

Dependencies

See Also

madd_vector_vector on page 167
madd_matrix_matrix on page 162

10.10 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 + row*col

where *A* is an *m* by *n* matrix, *row* is a *n* elements vector and *col* is an *m* elements 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 168
madd_vector_matrix on page 166
madd_number_vector on page 164

10.11 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 - row*col

where *A* is an *m* by *n* matrix, *row* is a *n* elements vector and *col* is an *m* elements 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 167
nmsub_number_vector on page 165

10.12 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 170
madd_number_vector on page 164
nmsub_number_vector on page 165

10.13 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*.

Dependencies

See Also

scale_vector on page 169
madd_number_vector on page 164
nmsub_number_vector on page 165

10.14 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 172
solve_upper_1 on page 173
solve_linear_system_1 on page 176

10.15 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 171
solve_upper_1 on page 173
solve_linear_system_1 on page 176

10.16 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 171
solve_unit_lower_1 on page 172
solve_linear_system_1 on page 176

10.17 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 175

10.18 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 174

10.19 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*
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

lu3_decomp_block on page 160
swap_matrix_rows on page 174
solve_unit_lower_1 on page 172
solve_upper_1 on page 173

See Also

lu3_decomp_block on page 160
swap_matrix_rows on page 174
solve_unit_lower_1 on page 172
solve_upper_1 on page 173

10.20 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

11. Math Library

The math library consists of a set of general purpose math routines. Many of the routines mimic those found in the standard system math library except these have be tuned to exploit the SIMD features and generally only support single precision. The *Game Math Library* provides some math functions of less than single precision accuracy.

This library is supported on both the PPE and SPE, however, not all functions are implemented on the PPE.

Name(s)

libmath.a

Header File(s)

<libmath.h>

11.1 acos

C Specification

```
#include <acosf.h>
inline float _acosf(float x)

#include <acosf_v.h>
inline vector float _acosf_v(vector float x)

#include <libmath.h>
float acosf(float x)

#include <libmath.h>
vector float acosf_v(vector float x)
```

Descriptions

The *acosf* subroutine computes the arc-cosine of the input, specified by the parameter *x*, to an accuracy of approximately single precision floating-point. The arc-cosine is the angle in radians in the range [0.0, π] whose cosine is *x*. The results are undefined if *x* is outside the range [-1.0, 1.0].

The *acosf_v* subroutine computes the arc-cosine on a vector (4 independent) of inputs.

Dependencies

asin on page 182

See Also

cos on page 189

11.2 acot

C Specification

```
#include <acotf.h>
inline float _acotf(float x)

#include <acotf_v.h>
inline vector float _acotf_v(vector float x)

#include <libmath.h>
float acotf(float x)

#include <libmath.h>
vector float acotf_v(vector float x)
```

Descriptions

The *acotf* subroutine computes the arc-cotangent of the input, specified by the parameter *x*, to an accuracy of approximately single precision floating-point. The arc-cotangent is the angle in radians in the range [0.0, π] whose cotangent is *x*.

The *acotf_v* subroutine computes the arc-cotangent on a vector (4 independent) of inputs.

Dependencies

inverse on page 231

See Also

cot on page 191
atan on page 183

11.3 asin

C Specification

```
#include <asinf.h>
inline float _asinf(float x)

#include <asinf_v.h>
inline vector float _asinf_v(vector float x)

#include <libmath.h>
float asinf(float x)

#include <libmath.h>
vector float asinf_v(vector float x)
```

Descriptions

The *asin*f subroutine computes the arc-sine of the input, specified by the parameter *x*, to an accuracy of approximately single precision floating-point. The arc-sine is the angle in radians in the range [-π/2, π/2] whose sine is *x*. The results are undefined if *x* is outside the range [-1.0, 1.0].

The *asinf_v* subroutine computes the arc-sine on a vector (4 independent) of inputs.

Dependencies

divide (floating point) on page 196
sqrt on page 261

See Also

sin on page 259
acos on page 180

11.4 atan

C Specification

<include <atanf.h>
inline float _atanf(float x)

#include <atanf_v.h>
inline vector float _atanf_v(vector float x)

#include <libmath.h>
float atanf(float x)

#include <libmath.h>
vector float atanf_v(vector float x)

Descriptions

The *atan*f subroutine computes the arc-tangent of the input, specified by the parameter *x*, to an accuracy of approximately single precision floating-point. The arc-tangent is the angle in radians in the range [π-/2, π/2] whose tangent is *x*.

The arc-tangent function is computed using an approximating polynomial (B. Carlson, M. Golstein, Los Alamos Scientific Laboratory, 1955).

$$\operatorname{atan} f(x) = \sum_{i=0}^{8} (C_i \times x^{(2 \times i + 1)})$$

for *x* in the range [-1.0, 1.0]. If *x* is in the range [-*infinity*, -1.0), then atan(x) = -π/2 + atan(-1/x). If *x* is in the range (1.0, *infinity*], then atan(x) = π/2 + atan(-1/x).

The *atanf_v* subroutine computes the arc-tangent on a vector (4 independent) of inputs.

Dependencies

inverse on page 231

See Also

tan on page 263
acot on page 181

11.5 cbrt

C Specification (SPE only)

```
#include <cbrtf.h>
inline float _cbrtf(float x)

#include <cbrt.h>
inline double _cbrt(double x)

#include <cbrtf_v.h>
inline vector float _cbrtf_v(vector float x)

#include <cbrtf_vfast.h>
inline vector float _cbrtf_vfast(vector float x)

#include <cbrt_v.h>
inline vector double _cbrt_v(vector double x)

#include <libmath.h>
float cbrtf(float x)

#include <libmath.h>
double cbrt(double x)

#include <libmath.h>
vector float cbrtf_v(vector float x)

#include <libmath.h>
vector float cbrtf_vfast(vector float x)

#include <libmath.h>
vector double cbrt_v(vector double x)
```

Aliases (SPE only)

```
#include <libmath.h>
long double cbrtl(long double x)
```

Descriptions

The *cbrt*f subroutine computes the cube root, specified by the parameter *x*, to an accuracy of single precision floating point. The *cbrt* computes the cube root to a accuracy of double precision floating point.

The *cbrtf_v* and *cbrt_v* subroutine computes a vector of cube roots of both single and double precision respectively. *cbrtf_v* computes 4 independent results, while *cbrt_v* computes 2 independent results.

The *cbrtf_vfast* subroutine computes a fast cube root on a vector (4 independent) floating point inputs. The version is guaranteed to be accurate to -8 ulp's (units of least precision) to 7 ulp's and has been provided to applications not requiring full accuracy, yet needing improved performance.

The *cbrtl* subroutine is aliased to the *cbrt* subroutine.

Inlined forms of these functions still require linkage with *math* library to resolve references to the cube root factor table - *cbrt_factors*.

Dependencies

See Also

sqrt on page 261

11.6 ceil

C Specification

```
#include <ceilf.h>
inline float _ceilf(float value)

#include <ceilf_v.h>
inline vector float _ceilf_v(vector float value)

#include <libmath.h>
float ceilf(float value)

#include <libmath.h>
vector float ceilf_v(vector float value)
```

C Specification (SPE only)

```
#include <ceil.h>
inline double _ceil(double value)

#include <ceil_v.h>
inline vector double _ceil(vector double value)

#include <libmath.h>
double ceil(double value)

#include <libmath.h>
vector double ceil_v(vector double value)
```

Aliases (SPE only)

```
#include <libmath.h>
long double ceill(long double value)
```

Descriptions

The *ceil*f and *ceilf_v* subroutines round the floating-point input (or set of inputs) specified by the input parameter *value* upwards to the nearest integer returning the result(s) in floating-point. Two forms of the ceiling function are provided - full range and limited (integer) range.

The full range form (default) provides ceiling computation on all IEEE floating-point values. The ceiling of NANs and inifinities remain unchanged. The ceiling of denorms results in zero, regardless of its sign.

The limited range form (selected during compilation by defining CEIL_INTEGER_RANGE), computes the ceiling for all floating-point values within the 32-bit signed integer range. Values outside this range get clamped.

The *ceill* subroutine is aliased to the *ceil* subroutine.

The full range form is the only form supported on the PPE.

The *ceil* and *ceil_v* subroutines round the double precision input (or pair of inputs) specified by the input parameter *value* upwards to the nearest integer returning the result(s) in double precision float. These functions are only supported on the SPE.

Dependencies

See Also

floor on page 217

11.7 copysign

C Specification (SPE only)

#include <copysignf.h>
inline float _copysignf(float x, float y)

#include <copysign.h>
inline double _copysign(double x, double y)

#include <copysignf_v.h>
inline vector float _copysignf_v(vector float x, vector float y)

#include <copysign_v.h>
inline vector double _copysign_v(vector double x, vector double y)

#include <libmath.h>
float copysignf(float x, float y)

#include <libmath.h>
double copysign(double x, double y)

#include <libmath.h>
vector float copysignf_v(vector float x, vector float y)

#include <libmath.h>
vector double _opysign_v(vector double x, vector double y)

Aliases (SPE only)

#include <libmath.h>
long double copysignl(long double x, long double y)

Descriptions

The *copysign* subroutines returns a value whose absolute value matches that of *x*, but whose sign matches that of *y*. Subroutines are provided to handle both single-presion and double-precision floating-point inputs as well as scalar and vector inputs.

The *copysignl* subroutine is aliased to the *copysign* subroutine.

Dependencies

See Also

fabs on page 203

11.8 cos

C Specification

```
#include <cosf.h>
inline float _cosf(float angle)

#include <cosf_v.h>
inline vector float _cosf_v(vector float angle)

#include <libmath.h>
float cosf(float angle)

#include <libmath.h>
vector float cosf_v(vector float angle)
```

C Specification (SPE only)

```
#include <cos.h>
inline double _cos(double angle)

#include <cos_v.h>
inline vector double _cos_v(vector double angle)

#include <libmath.h>
double cos_(double angle)

#include <libmath.h>
vector double cos_v(vector double angle)
```

Aliases (SPE only)

```
#include <libmath.h>
long double cosl(long double angle)
```

Descriptions

The *cos* subroutines, *cosf* and *cos*, computes the cosine of the input angle, specified by the parameter *angle*, to an accuracy of the specified input, single precision and double precision respectively. The input angle is expressed in radians.

The *cosf_v* and *cos_v* subroutines computes the cosine on a vector of single precision and double precision radian angles respectively.

The *cosl* subroutine is aliased to the *cos* subroutine.

Dependencies

See Also

sin on page 259

cos8, cos14, cos18 on page 134

11.9 cot

C Specification

#include <cotf.h>
inline float _cotf(float angle)

#include <cotf_v.h>
inline vector float _cotf_v(vector float angle)

#include <libmath.h>
float cotf(float angle)

#include <libmath.h>
vector float cotf_v(vector float value)

C Specification (SPE only)

#include <cot.h>
inline double _cot(double angle)

#include <cot_v.h>
inline vector double _cot_v(vector double angle)

#include <libmath.h>
double cot_(double angle)

#include <libmath.h>
vector double cot_v(vector double angle)

Aliases (SPE only)

#include <libmath.h>
long double cotl(long double angle)

Descriptions

The *cot* subroutines, *cotf* and *cot*, computes the cotangent of the input angle, specified by the parameter *angle*, to an accuracy of the specified input, single precision and double precision respectively. The input angle is expressed in radians.

The *cotf_v* and *cot_v* subroutines computes the cotangent on a vector of single precision and double precision radian angles respectively.

The *cotl* subroutine is aliased to the *cot* subroutine.

Dependencies

See Also

tan on page 263

acot on page 181

11.10 div

C Specification

```
#include <div.h>
inline div_t _div(int numer, int denom)

#include <div_v.h>
inline div_t_v _div_v(vector signed int numer, vector signed int denom)

#include <libmath.h>
div_t div(int numer, int denom)

#include <libmath.h>
div_t_v div_v(vector signed int numer, vector signed int denom)
```

Descriptions

The *div* subroutine computes the signed quotient and remainder of *numer* divided by *denom*. The results are returned in a *div_t* structure. The div_t structure is defined in div_t.h as follows:

```
typedef struct {
    int quot;
    int rem;
} div_t;
```

The *div_v* subroutine computes 4 simultaneous signed quotients and remainders for each of the components of the *numer* parameter divided by the corresponding *denom* parameter. The results are returned in a *div_t_v* structure. The div_t_v structure is defined in div_t.h as follows:

```
typedef struct {
    vector signed int quot;
    vector signed int rem;
} div_t_v;
```

Dependencies

See Also

divide (integer) on page 194
divide (floating point) on page 196

11.11 divide (integer)

C Specification

```
#include <divide_i.h>
inline signed int _divide_i(signed int dividend, signed int divisor)

#include <divide_ui.h>
inline unsigned int _divide_ui(unsigned int dividend, unsigned int divisor)

#include <divide_ll.h>
inline signed long long _divide_ll(signed long long dividend, signed long long divisor)

#include <divide_ull.h>
inline unsigned long long _divide_ull(unsigned long long dividend, unsigned long long divisor)

#include <divide_i_v.h>
inline vector signed int _divide_i_v(vector signed int dividend, vector signed int divisor)

#include <divide_ui_v.h>
inline vector unsigned int _divide_ui_v(vector unsigned int dividend, vector unsigned int divisor)

#include <divide_ll_v.h>
inline vector signed long long _divide_ll_v(vector signed long long dividend, vector signed long long divisor)

#include <divide_ull_v.h>
inline vector unsigned long long _divide_ull_v(vector unsigned long long dividend,
                                               vector unsigned long long divisor)

#include <libmath.h>
inline signed int _divide_i(signed int dividend, signed int divisor)

#include <libmath.h>
unsigned int divide_ui(unsigned int dividend, unsigned int divisor)

#include <libmath.h>
signed long long divide_ll(signed long long dividend, signed long long divisor)

#include <libmath.h>
unsigned long long divide_ull(unsigned long long dividend, unsigned long long divisor)

#include <libmath.h>
vector signed int divide_i_v(vector signed int dividend, vector signed int divisor)

#include <libmath.h>
vector unsigned int divide_ui_v(vector unsigned int dividend, vector unsigned int divisor)

#include <libmath.h>
vector signed long long divide_ll_v(vector signed long long dividend, vector signed long long divisor)
```

```
#include <divide_ull_v.h>
vector unsigned long long divide_ull_v(vector unsigned long long dividend, vector unsigned long long divisor)
```

Descriptions

The *divide_i* subroutine computes then signed integer quotient of *dividend* / *divisor*. If the divisor is 0, then a quotient of 0 is produced. 0 is also produced when 0x80000000 is divided by -1.

The *divide_ui* subroutine computes then unsigned integer quotient of *dividend* / *divisor.*

The *divide_ll* and *divide_ull* subroutines compute the signed and unsigned long long precision quotients respectively.

The *divide_i_v*, *divide_ui_v*, *divide_ll_v*, and *divide_ull_v* subroutines compute a vector of quotients by dividing each component of *dividend* by the corresponding component of *divisor.*

Dependencies

See Also

div on page 193
divide (floating point) on page 196
mod on page 245

11.12 divide (floating point)

C Specification

```
#include <divide.h>
inline float _divide(float dividend, float divisor)

#include <divide_v.h>
inline vector float _divide_v(vector float dividend, vector float divisor)

#include <libmath.h>
float divide(float dividend, float divisor)

#include <libmath.h>
vector float divide_v(vector float dividend, vector float divisor)
```

C Specification (SPE only)

```
#include <divide_d.h>
inline double _divide_d(double dividend, double divisor)

#include <divide_dv.h>
inline vector double _divide_dv(vector double dividend, vector double divisor)

#include <libmath.h>
double divide_d(double dividend, double divisor)

#include <libmath.h>
vector double divide_dv(vector double dividend, vector double divisor)
```

Descriptions

The *divide* subroutine divides the floating-point dividend by the floating-point divisor and returns the floating-point quotient. Computation is similar to taking the reciprocal of the divisor and multiplying it by the dividend, except this routine produces slightly better accuracy and is slightly more efficient.

Computation is performed using the processor's reciprocal estimate and interpolate instructions to produce and estimate accurate to approximately 12 bits. One iteration of Newton-Raphson is performed to produce a result accurate to single precision floating-point. Double precision results are obtained by further casting the single precision result and performing two additional double precision Newton-Raphson iterations.

Dependencies

See Also

divide (integer) on page 194
inverse on page 231

11.13 exp

C Specification

```
#include <expf.h>
inline float _expf(float x)

#include <expf_v.h>
inline vector float _expf_v(vector float x)

#include <libmath.h>
float expf(float x)

#include <libmath.h>
vector float expf_v(vector float x)
```

C Specification (SPE only)

```
#include <exp.h>
inline double _exp(double x)

#include <exp_v.h>
inline vector double _exp_v(vector double x)

#include <libmath.h>
double exp(double x)

#include <libmath.h>
vector double exp_v(vector double x)
```

Aliases (SPE only)

```
#include <libmath.h>
long double expl(long double x)
```

Descriptions

The *exp* subroutines computes *e* (the base of the natural logarithms) rasied to the input parameter *x* (i.e., e^x). *exp* is computed using *exp2* as follows:

$$e^x = 2^{(\log_2(e) \times x)}$$

Both single precision (*expf)* and double precision (*exp*) forms are provided. In addition, vectored forms are also provided. *expf_v* subroutine computes *e* raised to the *x* for a vector of 4 independent single precision values and *exp_v* compute *e* raised for a vector of 2 independent double precision values.

The *expl* subroutine is aliased to the *exp* subroutine.

Dependencies

exp2 on page 201

See Also

exp2 on page 201
exp10 on page 199
log on page 237
pow on page 249

11.14 exp10

C Specification

```
#include <exp10f.h>
inline float _exp10f(float x)

#include <exp10f_v.h>
inline vector float _exp10f_v(vector float x)

#include <libmath.h>
float exp10f(float x)

#include <libmath.h>
vector float exp10f_v(vector float x)
```

C Specification (SPE only)

```
#include <exp10.h>
inline double _exp10(double x)

#include <exp10_v.h>
inline vector double _exp10_v(vector double x)

#include <libmath.h>
double exp10(double x)

#include <libmath.h>
vector double exp10_v(vector double x)
```

Aliases (SPE only)

```
#include <libmath.h>
long double exp10l(long double x)
```

Descriptions

The *exp10* subroutines compute 10 rasied to the input parameter *x* (i.e., 10^x). *exp10* is computed using *exp2* as follows:

$$10^{x} = 2^{(\log_2(10) \times x)}$$

Both single precision (*exp10f)* and double precision (*exp10*) forms are provided. In addition, vectored forms are also provided. *exp10f_v* subroutine computes *10* raised to the *x* for a vector of 4 independent single precision values and *exp10_v* compute *10* raised for a vector of 2 independent double precision values.

The *exp10l* subroutine is aliased to the *exp10* subroutine.

Dependencies

exp2 on page 201

See Also

exp on page 197
log10 on page 239
pow on page 249

11.15 exp2

C Specification

```
#include <exp2f.h>
inline float _exp2f(float x)

#include <exp2f_v.h>
inline vector float _exp2f_v(vector float x)

#include <libmath.h>
float exp2f(float x)

#include <libmath.h>
vector float exp2f_v(vector float x)
```

C Specification (SPE only)

```
#include <exp2.h>
inline double _exp2(double x)

#include <exp2_v.h>
inline vector double _exp2_v(vector double x)

#include <libmath.h>
double exp2(double x)

#include <libmath.h>
vector double exp2_v(vector double x)
```

Aliases (SPE only)

```
#include <libmath.h>
long double exp2l(long double x)
```

Descriptions

The *exp2* subroutines compute 2 rasied to the input parameter *x* (i.e., 2^x). Computation is performed by observing that $2^{(a+b)} = 2^a * 2^b$. *x* is decomposed into *a* and *b* by letting a=ceil(x) and b=x-a. 2^a is easily computed by placing *a* into the exponent of a floating-point number whose mantissa is a zeros. 2^b is computed to floatinf-point precision using a *n*-order approximating polynomial of the form (C. Hastings Jr., 1955).

$$2^{(-x)} = \sum_{i=1}^{n} (C_i \times x^i)$$

For single precision accuracy a 7th order polynomial is used. For double precision accuracy a 13th order polynomial is used.

Both single precision (*exp2f)* and double precision (*exp2*) forms are provided. In addition, vectored forms are also provided. *exp2f_v* subroutine computes *2* raised to the *x* for a vector of 4 independent single precision values and *exp2_v* compute *2* raised for a vector of 2 independent double precision values.

The *exp2l* subroutine is aliased to the *exp2* subroutine.

Dependencies

See Also

exp on page 197
exp10 on page 199
log2 on page 241
pow on page 249

11.16 fabs

C Specification

```
#include <fabsf.h>
inline float _fabsf(float value)

#include <fabsf_v.h>
inline vector float _fabsf_v(vector float value)

#include <libmath.h>
float fabsf(float value)

#include <libmath.h>
vector float fabsf_v(vector float value)
```

C Specification (SPE only)

```
#include <fabs.h>
inline double _fabs(double value)

#include <fabs_v.h>
inline vector double _fabs_v(vector double value)

#include <libmath.h>
double fabs(double value)

#include <libmath.h>
vector double fabs_v(vector double value)
```

Aliases (SPE only)

```
#include <libmath.h>
long double fabsl(long double value)
```

Descriptions

The *fabs* subroutines return the absolute value of the floating-point input specified by the parameter *value.*

Both single precision (*fabsf)* and double precision (*fabs*) forms are provided. In addition, vectored forms are also provided. *fabsf_v* subroutine computes the absolute value of a vector of 4 independent single precision values and *fabs_v* computes the absolute value of a vector of 2 independent double precision values.

The *fabsl* subroutine is aliased to the *fabs* subroutine.

Dependencies

See Also

copysign on page 188

11.17 fdim

C Specification

```
#include <fdimf.h>
inline float _fdimf(float x, float y)

#include <fdimf_v.h>
inline vector float _fdimf_v(vector float x, vector float y)

#include <libmath.h>
float fdimf(float x, float y)

#include <libmath.h>
vector float fdimf_v(vector float x, vector float y)
```

C Specification (SPE only)

```
#include <fdim.h>
inline double _fdim(double x, double y)

#include <fdim_v.h>
inline vector double _fdim_v(vector double x, vector double y)

#include <libmath.h>
double fdim(double x, double y)

#include <libmath.h>
vector double fdim_v(vector double x, vector double y)
```

Aliases (SPE only)

```
#include <libmath.h>
long double fdiml(long double x, long double y)
```

Descriptions

The *fdim* subroutines compute the positive difference of inputs *x* and *y.*

Both single precision (*fdimf)* and double precision (*fdim*) forms are provided. In addition, vectored forms are also provided. *fdimf_v* subroutine computes the positive difference of a pair of 4 elements single precision vectors - *x* and *y* and *fdim_v* computes the positive difference of a pair of 2 element double precision vectors.

The *fdiml* subroutine is aliased to the *fdim* subroutine.

Dependencies

See Also

fabs on page 203

11.18 feclearexcept

C Specification (SPE only)

```
#include <feclearexcept.h>
inline void _feclearexcept(int excepts)
```

```
#include <libmath.h>
void _feclearexcept(int excepts)
```

Descriptions

The *feclearexcept* subroutine clears the supported exceptions represented by the bits in the *except* argument. The supported exceptions are: FE_DIVBYZERO, FE_OVERFLOW, FE_UNDERFLOW, FE_INEXACT, FE_INVALID, FE_NC_NAN, FE_NC_DENORM, FE_DIFF_SINGL. Consult the SPE ISA for complete description on these execptions.

Dependencies

See Also

fegetexceptflag on page 208
feraiseexcept on page 211
fesetexceptflag on page 213
fetestexcept on page 215

11.19 fegetenv

C Specification (SPE only)

```
#include <fegetenv.h>
inline void _fegetenv(fenv_t *envp)
```

```
#include <libmath.h>
void _fegetenv(fenv_t *envp)
```

Descriptions

The *fegetenv* subroutine saves the current floating point environment in the object pointed to by *envp*.

Dependencies

See Also

feholdexcept on page 210
fesetenv on page 212
feupdateenv on page 216

11.20 fegetexceptflag

C Specification (SPE only)

#include <fegetexceptflag.h>
inline void _fegetexceptflag(fexcept_t *flagp, int excepts)

#include <libmath.h>
void _fegetexceptflag(fexcept_t *flagp, int excepts)

Descriptions

The *fegetexceptflag* subroutine stores are representation of the state of the exception flags specified by the *excepts* parameter in the opaque objec pointed to by *flagp*. The supported exceptions are: FE_DIVBYZERO, FE_OVERFLOW, FE_UNDERFLOW, FE_INEXACT, FE_INVALID, FE_NC_NAN, FE_NC_DENORM, FE_DIFF_SINGL. Consult the SPE ISA for complete description on these execptions.

Dependencies

See Also

feclearexcept on page 206
feraiseexcept on page 211
fesetexceptflag on page 213
fetestexcept on page 215

11.21 fegetround

C Specification (SPE only)

```
#include <fegetround.h>
inline int _fegetround(void)
```

```
#include <libmath.h>
int _fegetround(void)
```

Descriptions

The *fegetround* subroutine return the value corresponding to the current double precision rounding mode. Supported double precision rounding modes include: FE_TONEAREST, FE_TOWARDZERO, FE_UPWARD, and FE_DOWNWARD.

Note: This routine only returns the scalar (element 0) rounding mode.

Dependencies

See Also

fesetround on page 214

11.22 feholdexcept

C Specification (SPE only)

#include <feholdexcept.h>
inline int _feholdexcept(fenv_t *envp)

#include <libmath.h>
int _feholdexcept(fenv_t *envp)

Descriptions

The *feholdexcept* subroutine saves the current floating point environment in the object pointed to by *envp* and clears all the exception flags. Zero is returned on successful completion.The supported exceptions are: FE_DIVBYZERO, FE_OVERFLOW, FE_UNDERFLOW, FE_INEXACT, FE_INVALID, FE_NC_NAN, FE_NC_DENORM, FE_DIFF_SINGL. Consult the SPE ISA for complete description on these execptions.

Dependencies

See Also

fegetenv on page 207
fesetenv on page 212
feupdateenv on page 216

11.23 feraiseexcept

C Specification (SPE only)

#include <feraiseexcept.h>
inline void _feraiseexcept(int excepts)

#include <libmath.h>
void _feraiseexcept(int excepts)

Descriptions

The *feraiseexcept* subroutine raises the exceptions represented by the bits in the *except* argument. The supported exceptions are: FE_DIVBYZERO, FE_OVERFLOW, FE_UNDERFLOW, FE_INEXACT, FE_INVALID, FE_NC_NAN, FE_NC_DENORM, FE_DIFF_SINGL. Consult the SPE ISA for complete description on these execptions.

Dependencies

See Also

feclearexcept on page 206
fegetexceptflag on page 208
fesetexceptflag on page 213
fetestexcept on page 215

11.24 fesetenv

C Specification (SPE only)

```
#include <fesetenv.h>
inline void _fesetenv(const fenv_t *envp)
```

```
#include <libmath.h>
void _fesetenv(const fenv_t *envp))
```

Descriptions

The *fesetenv* subroutine restores the floating point environment from the object specified by *envp*. This object must be known to be valid, for example, the result of a call to *fegetenv* or *feholdexcept* or equal to FE_DFL_ENV.

Dependencies

See Also

fegetenv on page 207
feholdexcept on page 210
feupdateenv on page 216

11.25 fesetexceptflag

C Specification (SPE only)

```
#include <fesetexceptflag.h>
inline void _fesetexceptflag(const fexcept_t *flagp, int excepts)
```

```
#include <libmath.h>
void _fesetexceptflag(const fexcept_t *flagp, int excepts)
```

Descriptions

The *fesetexceptflag* subroutine sets the complete status for the exceptions represented by *excepts* to the value specified by the *flagp*. The *flagp* must have been obtained by an earlier call to *fegetexceptflag* with an argument that contains all the bits in *excepts.* The supported exception are: FE_DIVBYZERO, FE_OVERFLOW, FE_UNDERFLOW, FE_INEXACT, FE_INVALID, FE_NC_NAN, FE_NC_DENORM, FE_DIFF_SINGL. Consult the SPE ISA for complete description on these execptions.

Dependencies

See Also

feclearexcept on page 206
fegetexceptflag on page 208
feraiseexcept on page 211
fetestexcept on page 215

11.26 fesetround

C Specification (SPE only)

```
#include <fesetround.h>
inline int _fesetround(int rounding_mode)

#include <libmath.h>
int _fesetround(int rounding_mode)
```

Descriptions

The *fesetround* subroutine sets the double precision rounding mode to the rounding mode specified by the *rounding_mode* parameter and return 0 upon success. Valid double precision rounding modes include: FE_TONEAREST, FE_TOWARDZERO, FE_UPWARD, and FE_DOWNWARD.

Note: This routine only sets the scalar (element 0) rounding mode.

Dependencies

See Also

fegetround on page 209

11.27 fetestexcept

C Specification (SPE only)

```
#include <fetestexcept.h>
inline int _fetestexcept(int excepts)
```

```
#include <libmath.h>
int _fetestexcept(int excepts)
```

Descriptions

The *fetestexcept* subroutine returns an integer in which the bits set are set that were set in the *excepts* argument and for which the the corresponding exception is currently set. The supported exception are: FE_DIVBYZERO, FE_OVERFLOW, FE_UNDERFLOW, FE_INEXACT, FE_INVALID, FE_NC_NAN, FE_NC_DENORM, FE_DIFF_SINGL. Consult the SPE ISA for complete description on these execptions.

Dependencies

See Also

feclearexcept on page 206
fegetexceptflag on page 208
feraiseexcept on page 211
fesetexceptflag on page 213

11.28 feupdateenv

C Specification (SPE only)

```
#include <feupdateenv.h>
inline void _feupdateenv(const fenv_t *envp)
```

```
#include <libmath.h>
void _feupdateenv(const fenv_t *envp)
```

Descriptions

The *feupdateenv* subroutine installs the floating-point environment represented by the object specified by the *envp* parameter, except the currenly raised exceptions are not cleared. After calling *feupdateenv* the raised exceptions will be a bitwise OR of those previously set with those in **envp*. The object **envp* must be known to be valid - a result of calling *fegetenv* or *feholdexcept.*

Dependencies

See Also

fegetenv on page 207
feholdexcept on page 210
fesetenv on page 212

11.29 floor

C Specification

```
#include <floorf.h>
inline float _floorf(float value)

#include <floorf_v.h>
inline vector float _floorf_v(vector float value)

#include <libmath.h>
float floorf(float value)

#include <libmath.h>
vector float floorf_v(vector float value)
```

C Specification (SPE only)

```
#include <floor.h>
inline double _floor(double value)

#include <floor_v.h>
inline vector double _floor_v(vector double value)

#include <libmath.h>
double floor(double value)

#include <libmath.h>
vector double floor_v(vector double value)
```

Aliases (SPE only)

```
#include <libmath.h>
long double floorl(long double value)
```

Descriptions

The *floorf* and *floorf_v* subroutines round the floating-point input (or set of inputs) specified by the input parameter *value* downwards to the nearest integer returning the result(s) in floating-point. Two forms of the floor function are provided - full range and limited (integer) range.

The full range form (default) provides floor computation on all IEEE floating-point values. The floor of NANs and inifinities remain unchanged. The floor of denorms results in zero, regardless of its sign.

The limited range form (selected during compilation by defining FLOOR_INTEGER_RANGE), computes the floor for all floating-point values within the 32-bit signed integer range. Values outside this range get clamped.

The *floorl* subroutine is aliased to the *floor* subroutine.

The full range form is the only form supported on the PPE.

The *floor* and *floor_v* subroutines round the double precision input (or pair of inputs) specified by the input parameter *value* downwards to the nearest integer returning the result(s) in double precision float. These functions are only supported on the SPE.

Dependencies

See Also

ceil on page 186

11.30 fma

C Specification

```
#include <fmaf.h>
inline float _fmaf(float x, float y, float z)

#include <fmaf_v.h>
inline vector float _fmaf_v(vector float x, vector float y, vector float z)

#include <libmath.h>
float fmaf(float x, float y, float y)

#include <libmath.h>
vector float fmaf_v(vector float x, vector float y, vector float z)
```

C Specification (SPE only)

```
#include <fma.h>
inline double _fma(double x, double y, double z)

#include <fma_v.h>
inline vector double _fma_v(vector double x, vector double y, vector double z)

#include <libmath.h>
double fma(double x, double y, double z)

#include <libmath.h>
vector double fma_v(vector double x, vector double y, vector double z)
```

Aliases (SPE only)

```
#include <libmath.h>
long double fmal(long double x, long double y, long double z)
```

Descriptions

The *fma* subroutines computes the floating-point multiple and add of $x*y$ + z rounded in a single ternary operation. Subroutines are provided to handle both single-presion and double-precision floating-point inputs as well as scalar and vector inputs.

The *fmal* subroutine is aliased to the *fma* subroutine.

Dependencies

See Also

11.31 fmax

C Specification

#include <fmaxf.h>
inline float _fmaxf(float x, float y)

#include <fmaxf_v.h>
inline vector float _fmaxf_v(vector float x, vector float y)

#include <libmath.h>
float fmaxf(float x, float y)

#include <libmath.h>
vector float fmaxf_v(vector float x, vector float y)

C Specification (SPE only)

#include <fmax.h>
inline double _fmax(double x, double y)

#include <fmax_v.h>
inline vector double _fmax_v(vector double x, vector double y)

#include <libmath.h>
double fmax(double x, double y)

#include <libmath.h>
vector double fmax_v(vector double x, vector double y)

Aliases (SPE only)

#include <libmath.h>
long double fmaxl(long double x, long double y)

Descriptions

The *fmax* subroutines computes maximum numeric value of their arguments - MAX(x, y). Subroutines are provided to handle both single-presion and double-precision floating-point inputs as well as scalar and vector inputs. The vectored forms compute the maximum numeric value their arugments on a element by element basis.

The *fmaxl* subroutine is aliased to the *fmax* subroutine.

Dependencies

See Also

fmin on page 221

11.32 fmin

C Specification

#include <fminf.h>
inline float _fminf(float x, float y)

#include <fminf_v.h>
inline vector float _fminf_v(vector float x, vector float y)

#include <libmath.h>
float fminf(float x, float y)

#include <libmath.h>
vector float fminf_v(vector float x, vector float y)

C Specification (SPE only)

#include <fmin.h>
inline double _fmin(double x, double y)

#include <fmin_v.h>
inline vector double _fmin_v(vector double x, vector double y)

#include <libmath.h>
double fmin(double x, double y)

#include <libmath.h>
vector double fmin_v(vector double x, vector double y)

Aliases (SPE only)

#include <libmath.h>
long double fminl(long double x, long double y)

Descriptions

The *fmin* subroutines computes minumum numeric value of their arguments - MIN(x, y). Subroutines are provided to handle both single-presion and double-precision floating-point inputs as well as scalar and vector inputs. The vectored forms compute the minumum numeric value their arugments on a element by element basis.

The *fminl* subroutine is aliased to the *fmin* subroutine.

Dependencies

See Also

fmax on page 220

11.33 fmod

C Specification

#include <fmodf.h>
inline float _fmodf(float x, float y)

#include <fmodf_v.h>
inline vector float _fmodf_v(vector float x, vector float y)

#include <libmath.h>
float fmodf(float x, float y)

#include <libmath.h>
vector float fmodf_v(vector float x, vector float y)

C Specification (SPE only)

#include <fmod.h>
inline double _fmod(double x, double y)

#include <fmod_v.h>
inline vector double _fmodf_v(vector double x, vector double y)

#include <libmath.h>
double fmod(double x, double y)

#include <libmath.h>
vector double fmod_v(vector double x, vector double y)

Aliases (SPE only)

#include <libmath.h>
long double fmodl(long double x, long double y)

Descriptions

The *fmod* subroutines compute the remainder of *x* divided by *y*. The return value is *x* - *n*y*, where *n* is the quotient of *x* and *y* (*x*/*y*), rounded towards 0.

Figure 11-1. fmod(x,y) : y>0

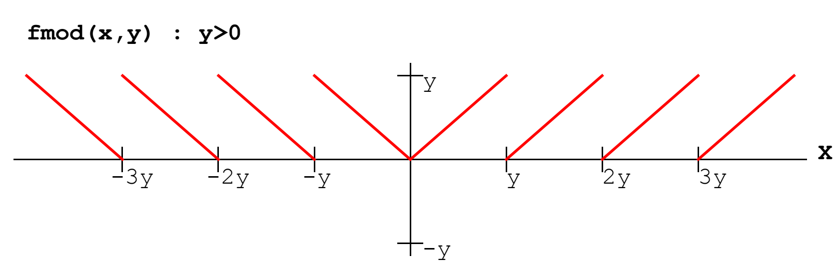
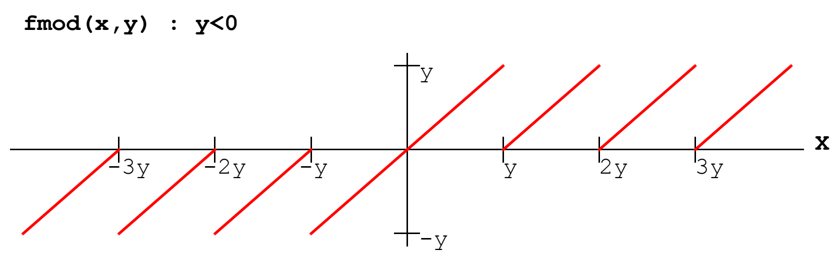


Figure 11-2. fmod(x,y) : y<0



Subroutines are provided to handle both single-presion and double-precision floating-point inputs as well as scalar and vector inputs. The vectored forms compute the floating-point remainder on a element by element basis.

Two forms of the fmodf function are provided - full range and limited (integer) range. The full range form (default) provides fmodf computation on all single precision IEEE floating-point values of *x*/*y*. The limited range form (selected during compilation by defining FMODF_INTEGER_RANGE), computes the fmod for all floating-point values of *x*/*y* within the 32-bit signed integer range. Values outside this range get clamped.

The *fmodl* subroutine is aliased to the *fmod* subroutine.

The full range form is the only form supported on the PPE.

Dependencies

divide (floating point) on page 196
fabs on page 203

See Also

fmodfs on page 224
mod on page 245
remainder on page 251

11.34 fmodfs

C Specification

#include <fmodfs.h>
inline float _fmodfs(float x, float y)

#include <fmodfs_v.h>
inline vector float _fmodfs_v(vector float x, vector float y)

#include <libmath.h>
float fmodfs(float x, float y)

#include <libmath.h>
vector float fmodfs_v(vector float x, vector float y)

Descriptions

The *fmodfs* subroutines the remainder of *x* divided by *y*. The return value is *x* - *n*y*, where *n* is the quotient of *x* and *y* (*x*/*y*), rounded towards negative infinity. The *fmodfs* subroutine is similar to *fmod*f except the result is cyclicly continuous and is more applicable for procedural texture generation.

Figure 11-3. fmodfs (x,y) : y>0

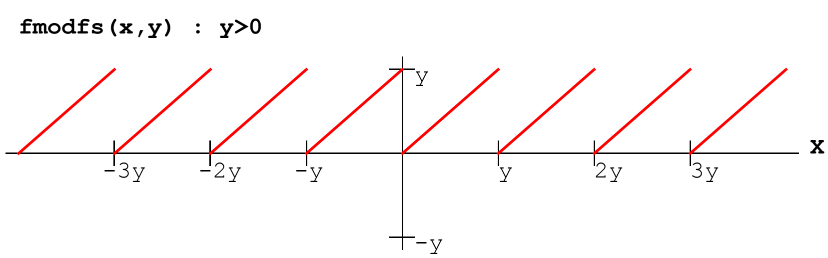
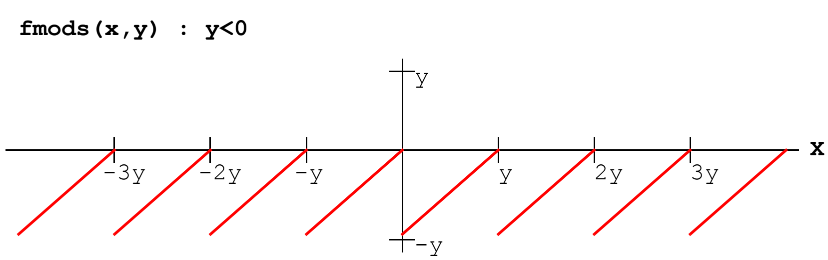


Figure 11-4. fmods (x,y) : y<0



Two forms of the fmods function are provided - full range and limited (integer) range. The full range form (default) provides fmod computation on all IEEE floating-point values of *x*/*y*. The limited range form (selected during compilation by defining FMODFS_INTEGER_RANGE), computes the fmods for all floating-point values of *x*/*y* within the 32-bit signed integer range. Values outside this range get clamped.

The full range form is the only form supported on the PPE.

Dependencies

divide (floating point) on page 196
fabs on page 203

See Also

fmod on page 222
mod on page 245

11.35 frexp

C Specification (SPE only)

```
#include <frexpf.h>
inline float _frexpf(float x, int *pexp)

#include <frexpf_v.h>
inline vector float _frexpf_v(vector float x, vector signed int *pexp)

#include <frexp.h>
inline double _frexp(double x, int *pexp)

#include <frexp_v.h>
inline vector double _frexp_v(vector double x, vector signed int *pexp)

#include <libmath.h>
float frexpf(float x, int *pexp)

#include <libmath.h>
vector float frexpf_v(vector float x, vector signed int *pexp)

#include <libmath.h>
double frexp(double x, int *pexp)

#include <libmath.h>
vector double frexp_v(vector double x, vector signed int *pexp)
```

Aliases (SPE only)

```
#include <libmath.h>
long double frexpl(long double x, int *pexp)
```

Descriptions

The *frexp* subroutines split the number *x* into a normalized fraction and an exponent. If *x* is not zero, the normalized fraction is *x* times a power of two exponent, and is always in the half-open range [0.5, 1.0). The normalized fraction is returned and the power of two exponents is returned in the memory pointed to by *pexp.* If *x* is zero, then the normalized fraction is zero and zero is stored in **pexp*.

Subroutines are provided to handle both single-presion and double-precision floating-point inputs as well as scalar and vector inputs. The vectored forms split the elements of the number *x* on a element by element basis.

The results are undefined for double precision infinities and NaNs. The two exponents returned by *fexp_v* are returned in the even elements of the vector pointed to by *pexp*. The odd elements are zeroed.

The *frexp* subroutine is aliased to the *fmax* subroutine.

Dependencies

See Also

fmod on page 222
ldexp on page 233

11.36 ilog2

C Specification

```
#include <ilog2.h>
inline signed int _ilog2(signed int x)

#include <ilog2_v.h>
inline vector signed int _ilog2_v(vector signed int x)

#include <libmath.h>
signed int ilog2(signed int x)

#include <libmath.h>
vector signed int ilog2_v(vector signed int x)
```

Descriptions

The *ilog2* subroutine computes the ceiling of the base-2 logarithm of the signed integer input parameter *x*.

$$\mathrm{ilog2}(x) = \mathrm{ceil}(\log_2(x))$$

The *ilog2_v* subroutine computes the ceiling of the base-2 logarithm for a vector of 4 independent signed integer values.

The i*log2* subroutines assume that *x* is a non-zero positive value. Undefined results will occur for values outside this domain.

Dependencies

See Also

ceil on page 186
log2 on page 241

11.37 ilogb

C Specification (SPE only)

```
#include <ilogbf.h>
inline int _ilogbf(float x)

#include <ilogbf_v.h>
inline vector signed int _ilogb_v(vector float x)

#include <ilogb.h>
inline int _ilogb(double x)

#include <ilogb_v.h>
inline vector signed int _ilogb_v(vector double x)

#include <libmath.h>
int ilogbf(float x)

#include <libmath.h>
vector signed int ilogbf_v(vector float x)

#include <libmath.h>
int ilogb(double x)

#include <libmath.h>
vector signed int ilogb_v(vector double x)
```

Aliases (SPE only)

```
#include <libmath.h>
int ilogbl(long double x)
```

Descriptions

The *ilogb* subroutines return the signed exponent of the floating-point input *x.* The

Subroutines are provided to handle both single-presion and double-precision floating-point inputs as well as scalar and vector inputs. The vectored forms return the exponent for each element of the input vector *x*.

Single-precision denorms are treated like zero and return FP_ILOGB0 (INT_MIN).

Double precision infinities and NaNs return FP_ILOGBNAN (INT_MAX). A double precision zero returns FP_ILOGB0 (INT_MIN). The two exponents returned by *ilogb_v* are returned in the even elements. The odd elements are zeroed.

The *ilogbl* subroutine is aliased to the *ilogb* subroutine.

Dependencies

See Also

log2 on page 241

11.38 inverse

C Specification

```
#include <inverse.h>
inline float _inverse(float value)

#include <inverse_v.h>
inline vector float _inverse_v(vector float value)

#include <libmath.h>
float inverse(float value)

#include <libmath.h>
vector float inverse_v(vector float value)
```

C Specification (SPE only)

```
#include <inverse_d.h>
inline double _inverse_d(double value)

#include <inverse_dv.h>
inline vector double _inverse_dv(vector double value)

#include <libmath.h>
double inverse_d(double value)

#include <libmath.h>
vector double inverse_dv(vector double value)
```

Descriptions

The *inverse* subroutine computes the reciprocal of the floating-point input value (specified by the *value* parameter). Computation is performed using the processor's reciprocal estimate and interpolate instructions to produce and estimate accurate to approximately 12 bits. One iteration of Newton-Raphson is performed to produce a result accurate to single precision floating-point.

Double precision results are obtained by further casting the single precision result and performing two additional double precision Newton-Raphson iterations.

Dependencies

See Also

divide (floating point) on page 196

11.39 inv_sqrt

C Specification

#include <inv_sqrt.h>
inline float _inv_sqrt(float value)

#include <inv_sqrt_v.h>
inline vector float _inv_sqrt_v(vector float value)

#include <libmath.h>
float inv_sqrt(float value)

#include <libmath.h>
vector float inv_sqrt_v(vector float value)

C Specification (SPE only)

#include <inv_sqrt_d.h>
inline double _inv_sqrt(double value)

#include <inv_sqrt_dv.h>
inline vector double _inv_sqrt_v(vector double value)

#include <libmath.h>
double inv_sqrt_d(double value)

#include <libmath.h>
vector double inv_sqrt_dv(vector double value)

Descriptions

The *inv_sqrt* subroutine computes the reciprocal square root of the number (or vector of numbers) specified by the *value* parameter. Computation is performed using the floating-point reciprocal square root estimate and interpolate (SPE only) instructions to generate an estimate accurate to 12 bits. One iteration of a Newton-Raphson is performed to improve accuracy to single precision floating-point.

Double precision results are obtained by further casting the single precision result and performing two additional double precision Newton-Raphson iterations.

Dependencies

See Also

sqrt on page 261

11.40 ldexp

C Specification (SPE only)

```
#include <ldexpf.h>
inline float _ldexpf(float x, int exp)

#include <ldexpf_v.h>
inline vector float _ldexpf_v(vector float x, vector signed int exp)

#include <ldexp.h>
inline double _ldexp(double x, int exp)

#include <ldexp_v.h>
inline vector double _ldexp_v(vector double x, vector signed int exp)

#include <libmath.h>
float ldexpf(float x, int exp)

#include <libmath.h>
vector float ldexpf_v(vector float x, vector signed int exp)

#include <libmath.h>
double ldexp(double x, int exp)

#include <libmath.h>
vector double ldexp_v(vector double x, vector signed int exp)
```

Aliases (SPE only)

```
#include <libmath.h>
long double ldexpl(long double x, int exp)
```

Descriptions

The *ldexp* subroutines return the result of multiplying the floating point number(s) of *x* by 2 raised to the power *exp*. The result is force to 0 on underflow, and FLT_MAX on overflow.

The double precision functions correctly handle underflow, overflow, and denorms by breaking the problem into the following sequences:

exp = MAX(exp, -2044);
exp = MIN(exp, 2046);
exp1 = exp / 2;
exp2 = exp - exp1;
result = x * 2^{exp1} * 2^{exp2};

The *ldexp_v* subroutines uses the even elements of *exp*.

The *ldexpl* subroutine is aliased to the *ldexp* subroutine.

Dependencies

See Also

fmod on page 222
frexp on page 226

11.41 llrint

C Specification (SPE only)

#include <llrintf.h>
inline long long int _llrintf(float x)

#include <llrint.h>
inline long long int _llrint(double x)

#include <llrint_v.h>
inline vector long long _llrint_v(vector double x)

#include <libmath.h>
long long int llrintf(float x)

#include <libmath.h>
long long int llrint(double x)

#include <libmath.h>
vector long long llrint_v(vector double x)

Aliases (SPE only)

#include <libmath.h>
long long int llrintl(long double x)

Descriptions

The *llrint* subroutines round the input(s) specified by the parameter *x* to the nearest long long integer value, using the current rounding direction. For single precision floating point functions, *llrintf*, the rounding direction is always toward zero. If *x* is infinite, NaN, or if the rounded value is outside the range of its return type, the numerical result is unspecified. The rounded long long integer value is returned.

The *llrintl* subroutine is aliased to the *llrint* subroutine.

Dependencies

See Also

ceil on page 186
floor on page 217
llround on page 236
lrint on page 243
nearbyint on page 248
round on page 256

11.42 llround

C Specification (SPE only)

#include <llroundf.h>
inline long long int _llroundf(float x)

#include <llround.h>
inline long long int _llround(double x)

#include <llround_v.h>
inline vector signed long long _llround_v(vector double x)

#include <libmath.h>
long long int llroundf(float x)

#include <libmath.h>
long long int llround(double x)

#include <libmath.h>
vector signed long long llround_v(vector double x)

Aliases (SPE only)

#include <libmath.h>
long long int llroundl(long double x)

Descriptions

The *llround* subroutines round the input(s) specified by the parameter *x* to the nearest long long integer value, round away from zero regardless of the current rounding direction. If *x* is infinite, NaN, or if the rounded value is outside the range of its return type, the numerical result is unspecified. The rounded long long integer value is returned.

The *llroundl* subroutine is aliased to the *llround* subroutine.

Dependencies

See Also

ceil on page 186
floor on page 217
llrint on page 235
lrint on page 243
lround on page 244
nearbyint on page 248
round on page 256

11.43 log

C Specification

```
#include <logf.h>
inline float _logf(float x)

#include <logf_v.h>
inline vector float _logf_v(vector float x)

#include <libmath.h>
float logf(float x)

#include <libmath.h>
vector float logf_v(vector float x)
```

C Specification (SPE only)

```
#include <log.h>
inline double _log(double x)

#include <log_v.h>
inline vector double _log_v(vector double x)

#include <libmath.h>
double log(double x)

#include <libmath_v.h>
vector float logf_v(vector float x)
```

Aliases (SPE only)

```
#include <libmath.h>
long double logl(long double x)
```

Descriptions

The *log* subroutines compute the natural logarithm of the input parameter *x*. *log* is computed using *log2* as follows:

$$\log(x) = \frac{\log_2(x)}{\log_2(e)}$$

The *logf_v* subroutine computes the natural logarithm for a vector of 4 independent single precision floating-point values. The *log_v* subroutine computes the natural logarithm for a vector of 2 independent double precision floating-point values.

The *logl* subroutine is an alias for the *log* subroutine.

Dependencies

log2 on page 241

See Also

log2 on page 241
log10 on page 239
exp on page 197
pow on page 249

11.44 log10

C Specification

#include <log10f.h>
inline float _log10f(float x)

#include <log10f_v.h>
inline vector float _log10f_v(vector float x)

#include <libmath.h>
float log10f(float x)

#include <libmath.h>
vector float log10f_v(vector float x)

C Specification (SPE only)

#include <log10.h>
inline double _log10(double x)

#include <log10_v.h>
inline vector double _log10_v(vector double x)

#include <libmath.h>
double log10(double x)

#include <libmath.h>
vector double log10_v(vector double x)

Aliases (SPE only)

#include <libmath.h>
long double log10l(long double x)

Descriptions

The *log10* subroutines compute the base-10 logarithm of the input parameter *x*. *log10f* is computed using *log2* as follows:

$$\log 10(x) = \frac{\log_2(x)}{\log_2(10)}$$

The *log10f_v* subroutine computes the base-10 logarithm for a vector of 4 independent single precision floating-point values. The *log10_v* subroutine computes the base-10 logarithm for a vector of 2 independent double precision floating-point values.

The *log10l* subroutine is an alias for the *log10* subroutine.

Dependencies

log2 on page 241

See Also

log2 on page 241
log on page 237
exp10 on page 199
pow on page 249

11.45 log2

C Specification

```
#include <log2f.h>
inline float _log2f(float x)

#include <log2f_v.h>
inline vector float _log2f_v(vector float x)

#include <libmath.h>
float log2f(float x)

#include <libmath.h>
vector float log2f_v(vector float x)
```

C Specification (SPE only)

```
#include <log2.h>
inline double_log2(double x)

#include <log2_v.h>
inline vector double _log2_v(vector double x)

#include <libmath.h>
double log2(double x)

#include <libmath.h>
vector double log2_v(vector double x)
```

Aliases (SPE only)

```
#include <libmath.h>
long double log2l(long double x)
```

Descriptions

The *log2* subroutine computes the base-2 logarithm of the input parameter *x*. Log base 2 of *x* is approximated to single precision floating-point using an 8th order polynomial (C. Hastings Jr., 1955)

$$\log 2(1 + x) = \sum_{i=1}^{8} (C_i \times x^i)$$

for *x* in the range [0.0, 1.0]. The *log2f* subroutine assumes that *x* is a non-zero positive value.

The *log2f_v* subroutine computes the base-2 logarithm for a vector of 4 independent single precision floating-point values. The *log2_v* subroutine computes the base-2 logarithm for a vector of 2 independent double precision floating-point values.

The *log2l* subroutine is an alias for the *log2* subroutine.

Dependencies

See Also

exp2 on page 201
ilog2 on page 228
log on page 237
log10 on page 239
pow on page 249

11.46 lrint

C Specification (SPE only)

#include <lrintf.h>
inline long int _lrintf(float x)

#include <lrint.h>
inline long int _lrint(double x)

#include <lrint_v.h>
inline vector signed int _lrint_v(vector double x)

#include <libmath.h>
long int lrintf(float x)

#include <libmath.h>
long int lrint(double x)

#include <libmath.h>
vector signed int lrint_v(vector double x)

Aliases (SPE only)

#include <libmath.h>
long int lrintl(long double x)

Descriptions

The *lrint* subroutines round the input(s) specified by the parameter *x* to the nearest long integer value, using the current rounding direction. For single precision floating point functions, *lrintf*, the rounding direction is always toward zero. If *x* is infinite, NaN, or if the rounded value is outside the range of its return type, the numerica result is unspecified. The rounded long integer value is returned.

The *lrint_v* subroutines returned the rounding pair of double precision inputs specified by *x* in the even (0, and 2) elements of the signed integer vector. The odd elements (1 and 3) are zeroed.

The *lrintl* subroutine is aliased to the *lrint* subroutine.

Dependencies

See Also

ceil on page 186
floor on page 217
llrint on page 235
llround on page 236
nearbyint on page 248
round on page 256

11.47 lround

C Specification (SPE only)

```
#include <lroundf.h>
inline long int _lroundf(float x)

#include <lround.h>
inline long int _lround(double x)

#include <lround_v.h>
inline vector signed int _lround_v(vector double x)

#include <libmath.h>
long int lroundf(float x)

#include <libmath.h>
long int lround(double x)

#include <libmath.h>
vector signed int lround_v(vector double x)
```

Aliases (SPE only)

```
#include <libmath.h>
long int lroundl(long double x)
```

Descriptions

The *lround* subroutines round the input(s) specified by the parameter *x* to the nearest long integer value, round away from zero regardless of the current rounding direction. If *x* is infinite, NaN, or if the rounded value is outside the range of its return type, the numerical result is unspecified. The rounded long integer value is returned.

The *lroundl* subroutine is aliased to the *lround* subroutine.

Dependencies

See Also

ceil on page 186
floor on page 217
llrint on page 235
llround on page 236
lrint on page 243
nearbyint on page 248
round on page 256

11.48 mod

C Specification

```
#include <mod_i.h>
inline signed int _mod_i(signed int dividend, signed int divisor)

#include <mod_ui.h>
inline unsigned int _mod_ui(unsigned int dividend,
                            unsigned int divisor)

#include <mod_i_v.h>
inline vector signed int _mod_i_v(vector signed int dividend,
                                  vector signed int divisor)

#include <mod_ui_v.h>
inline vector unsigned int _mod_ui_v(vector unsigned int dividend,
                                     vector unsigned int divisor)

#include <libmath.h>
inline signed int _mod_i(signed int dividend, signed int divisor)

#include <libmath.h>
unsigned int mod_ui(unsigned int dividend, unsigned int divisor)

#include <libmath.h>
vector signed int mod_i_v(vector signed int dividend,
                          vector signed int divisor)

#include <libmath.h>
vector unsigned int mod_ui_v(vector unsigned int dividend,
                             vector unsigned int divisor)
```

Descriptions

The *mod_i* subroutine computes then signed integer remainder of *dividend* / *divisor*. If the divisor is 0, then a reaminder equal to the dividend is produced. 0x80000000 is also when the dividend is 0x80000000 and the divisor is -1.

The *mod_ui* subroutine computes then unsigned integer remainder of *dividend* / *divisor*.

The *mod_i_v* and *mod_ui_v* subroutines compute a vector of remainders by dividing each component of *dividen* by the corresponding component of *divisor.*

Dependencies

See Also

divide (integer) on page 194
fmod on page 222

fmodfs on page 224

11.49 multiply

C Specification

#include <multiply_ui_v.h>
inline vector unsigned int _multiply_ui_v(vector unsigned int x, vecotor unsigned int y)

#include <multiply_ull.h>
inline unsigned long long _multiply_ull(unsigned long long x, unsigned long long y)

#include <libmath.h>
vector unsigned int multiply_ui_v(vector unsigned int x, vector unsigned int y)

#include <libmath.h>
unsigned long long multiply_ull(unsigned long long x, unsigned long long y)

C Specification (SPE only)

#include <multiply_ull_v.h>
inline vector unsigned long long _multiply_ull_v(vector unsigned long long x, vector unsigned long long y)

#include <libmath.h>
vector unsigned long long multiply_ull_v(vector unsigned long long x, vector unsigned long long y)

Descriptions

The *multiply_ui_v* subroutine computes a vector of products by multiplying each component of *x* with the corresponding component of *y*.

The *multiple_ull* subroutine computes the product of unsigned long long parameters *x* and *y*.

The *multiply_ull_v* subroutine computes a vector of products by multiplying each unsigned long long component of *x* with the corresponding component of *y*.

These routine can also be used for multiplying signed quantities.

Dependencies

See Also

divide (integer) on page 194

11.50 nearbyint

C Specification (SPE only)

```
#include <nearbyint.h>
inline double _nearbyint(double x)
```

```
#include <nearbyint_v.h>
inline vector double _nearbyint_v(vector double x)
```

```
#include <libmath.h>
double nearbyint(double x)
```

```
#include <libmath.h>
vector double nearbyint_v(vector double x)
```

Aliases (SPE only)

```
#include <libmath.h>
float nearbyintf(float x)
```

```
#include <libmath.h>
vector float nearbyintf_v(vector float x)
```

```
#include <libmath.h>
long double nearbyintl(long double x)
```

Descriptions

The *nearbyint* subroutines round the input(s) specified by the parameter *x* to the nearest integer value in floating-point format, using the current rounding direction. If *x* is infinite, NaN, or if the rounded value is outside the range of its return type, the numerical result is unspecified. The rounded long long integer value is returned.

For single precision floating point functions, *nearbyintf* and *nearbyintf_v*, the rounding direction is always toward zero. Therefore, these functions are aliased to *truncf* and *truncf_v* respectively.

The *nearbyint* subroutines are identical to *rint* except that it will set the inexact FPSCR bit.

The *nearbyintl* subroutine is aliased to the *nearbyint* subroutine.

Dependencies

See Also

floor on page 217
llround on page 236
lrint on page 243
rint on page 255
trunc on page 265

11.51 pow

C Specification

#include <powf.h>
inline float _powf(float x, float y)

#include <powf_v.h>
inline vector float _powf_v(vector float x, vector float y)

#include <libmath.h>
float powf(float x, float y)

#include <libmath.h>
vector float powf_v(vector float x, vector float y)

C Specification (SPE only)

#include <pow.h>
inline double _pow(double x, double y)

#include <pow_v.h>
inline vector double _pow_v(vector double x, vector double y)

#include <libmath.h>
double pow(double x, double y)

#include <libmath.h>
vector double pow_v(vector double x, vector double y)

Aliases (SPE only)

#include <libmath.h>
long double powl(long double x, long double y)

Descriptions

The *pow* subroutines compute the input parameter *x* raised to the input parameter *y* (i.e., x^y). The power function is computed using *exp2* and *log2* as follows:

$$\mathrm{pow}(x,y) = x^y = 2^{(y \times \log_2(x))}$$

The *powf_v* subroutine computes x^y for a vector of 4 independent single precision floating-point values. The *pow_v* subroutine computes x^y for a vector of 2 independent double precision floating-point values.

The *powl* subroutine is aliased to the *pow* subroutine.

Dependencies

exp2 on page 201

log2 on page 241

See Also

exp2 on page 201
log2 on page 241

11.52 remainder

C Specification (SPE only)

```
#include <remainderf.h>
inline float _remainderf(float x, float y)

#include <remainderf_v.h>
inline vector float _remainderf_v(vector float x, vector float y)

#include <remainder.h>
inline double _remainder(double x, double y)

#include <remainder_v.h>
inline vector double _remainder_v(vector double x, vector double y)

#include <libmath.h>
float remainderf(float x, float y)

#include <libmath.h>
vector float remainderf_v(vector float x, vector float y)

#include <libmath.h>
double remainder(double x, double y)

#include <libmath.h>
vector double remainder_v(vector double x, vector double y)
```

Aliases (SPE only)

```
#include <libmath.h>
long double remainderl(long double x, long double y)
```

Descriptions

The *remainder* subroutines compute the remainder of dividing *x* by *y*. The return value is *x* - n * *y*, where n is the value *x* / *y*, rounded to the nearest integer. If this fractional part of the quotient is 0.5, it is rounded to the nearest even number (independent of the current rounding mode). If the return value is 0, it has a sign of *x.* The result is unspecified if *y* is equal to 0.

The *remainderf_v* subroutine computes the remainder for each of the 4 elements of a vector of single precision floating-point values of *x* and *y*. The *remainder_v* subroutine computes the raminder for each of the 2 elements of a vector of double precision floating-point values of *x* and *y*.

The *remainderl* subroutine is aliased to the *remainder* subroutine.

Dependencies

See Also

fmod on page 222
remquo on page 253

11.53 remquo

C Specification (SPE only)

#include <remquof.h>
inline float _remquof(float x, float y, int *quo)

#include <remquof_v.h>
inline vector float _remquof_v(vector float x, vector float y, vector signed int *quo)

#include <remquo.h>
inline double _remquo(double x, double y, int *quo)

#include <remquo_v.h>
inline vector double _remquo_v(vector double x, vector double y, vector signed int *quo)

#include <libmath.h>
float remquof(float x, float y. int *quo)

#include <libmath.h>
vector float remquof_v(vector float x, vector float y, vector signed int quo)

#include <libmath.h>
double remquo(double x, double y, int *quo)

#include <libmath.h>
vector double remquo_v(vector double x, vector double y, vector signed int *quo)

Aliases (SPE only)

#include <libmath.h>
long double remquol(long double x, long double y, int *quo)

Descriptions

The *remquo* subroutines compute the remainder of dividing *x* by *y*. The return value is *x* - n * *y*, where n is the value *x* / *y*, rounded to the nearest integer. If this fractional part of the quotient is 0.5, it is rounded to the nearest even number (independent of the current rounding mode). If the return value is 0, it has a sign of *x.* The result is unspecified if *y* is equal to 0.

The object pointed to by *quo*, the value whose sign is the sign of *x* and whose magnitude is congruent modulo 2^n to the magnitude of the integral quotient of *x* / *y*, where n is 3.

The *remquof_v* subroutine computes the remquo for each of the 4 elements of a vector of single precision floating-point values of *x* and *y*. The *remquo_v* subroutine computes the raminder for each of the 2 elements of a vector of double precision floating-point values of *x* and *y*. *remquo_v* returns 2 quo's in the vector specified by the *quo* parameter. Word elements 0 and 1 contain the congruent modulo for elements 0 of the inputs, and word elements 2 and 2 contain the congruent modulo for elements 1 of the inputs.

The *remquol* subroutine is aliased to the *remquo* subroutine.

Dependencies

See Also

remainder on page 251

11.54 rint

C Specification (SPE only)

#include <nearbyint.h>
inline double _rint(double x)

#include <rint_v.h>
inline vector double _rint_v(vector double x)

#include <libmath.h>
double rint(double x)

#include <libmath.h>
vector double rint_v(vector double x)

Aliases (SPE only)

#include <libmath.h>
float rintf(float x)

#include <libmath.h>
vector float rintf_v(vector float x)

#include <libmath.h>
long double rintl(long double x)

Descriptions

The *rint* subroutines round the input(s) specified by the parameter *x* to the nearest integer value in floating-point format, using the current rounding direction. If *x* is infinite, NaN, or if the rounded value is outside the range of its return type, the numerical result is unspecified. The rounded long long integer value is returned.

For single precision floating point functions, *rintf* and *rintf_v*, the rounding direction is always toward zero. Therefore, these functions are aliased to *truncf* and *truncf_v* respectively.

The *rint* subroutines are identical to *nearbyint* except that it does not set the inexact FPSCR bit.

The *rintl* subroutine is aliased to the *rint* subroutine.

Dependencies

See Also

floor on page 217
llround on page 236
lrint on page 243
nearbyint on page 248
trunc on page 265

11.55 round

C Specification (SPE only)

```
#include <roundf.h>
inline float _roundf(float x)

#include <round.h>
inline double _round(double x)

#include <round_v.h>
inline vector double _round_v(vector double x)

#include <libmath.h>
float roundf(float x)

#include <libmath.h>
double round(double x)

#include <libmath.h>
vector long long round_v(vector double x)
```

Aliases (SPE only)

```
#include <libmath.h>
long long int roundl(long double x)
```

Descriptions

The *round* subroutines round the input(s) specified by the parameter *x* to the nearest integer value, but round half-way value away from zero (regardless of the current rounding direction). If *x* is infinite, *x* itself is returned.

The *roundl* subroutine is aliased to the *round* subroutine.

Dependencies

See Also

ceil on page 186
floor on page 217
lround on page 244
nearbyint on page 248
rint on page 255
trunc on page 265

11.56 scalbn

C Specification (SPE only)

```
#include <scalbnf.h>
inline float _scalbnf(float x, int exp)

#include <scalbn.h>
inline double _scalbn(double x, int exp)

#include <libmath.h>
float scalbnf(float x, int exp)

#include <libmath.h>
double scalbn(double x, int exp)

#include <libmath.h>
vector double scalbn_v(vector double x, vector signed int exp)
```

Aliases (SPE only)

```
#include <libmath.h>
float scalblnf(float x, long int exp)

#include <libmath.h>
vector float scalbnf_v(vector float x, vector signed int exp)

#include <libmath.h>
vector double float scalbn_v(vector double x, vector signed int exp)

#include <libmath.h>
long double scalbnl(long double x, int exp)
```

Descriptions

The *scalbn* subroutines return the result of multiplying the floating point number(s) of *x* by 2 raised to the power *exp*. The result is force to 0 on underflow, and FLT_MAX on overflow.

The double precision functions correctly handle underflow, overflow, and denorms by breaking the problem into the following sequences:

exp = MAX(exp, -2044);
exp = MIN(exp, 2046);
exp1 = exp / 2;
exp2 = exp - exp1;
result = x * 2^{exp1} * 2^{exp2};

The scalar single precision variant, *scalbnf*, computes the result without any floating-point operations and as such does not any floating point exception flags.

The *scalbn_v* and *scalnbf_v* subroutines are aliased to *ldexp_v* and *ldexpf_v*, respectively.

The scalblnf is aliased to *scalbnf*.

The *scalbnl* subroutine is aliased to the *scalbn* subroutine.

11.57 sin

C Specification

#include <sinf.h>
inline float _sinf(float angle)

#include <sinf_v.h>
inline vector float _sinf_v(vector float angle)

#include <libmath.h>
float sinf(float angle)

#include <libmath.h>
vector float sinf_v(vector float angle)

C Specification (SPE only)

#include <sin.h>
inline double _sin(double angle)

#include <sin_v.h>
inline vector double _sin_v(vector double angle)

#include <libmath.h>
double sin_(double angle)

#include <libmath.h>
vector double sin_v(vector double angle)

Aliases (SPE only)

#include <libmath.h>
long double sinl(long double angle)

Descriptions

The *sin* subroutines, *sinf* and *sin*, computes the sine of the input angle, specified by the parameter *angle*, to an accuracy of the specified input, single precision and double precision respectively. The input angle is expressed in radians.

The *sinf_v* and *sin_v* subroutines computes the sine on a vector of single precision and double precision radian angles respectively.

The *sinl* subroutine is aliased to the *sin* subroutine.

Dependencies

See Also

cos on page 189

sin8, sin14, sin18 on page 139

11.58 sqrt

C Specification

#include <sqrtf.h>
inline float _sqrtf(float value)

#include <sqrtf_v.h>
inline vector float _sqrtf_v(vector float value)

#include <libmath.h>
float sqrtf(float value)

#include <libmath.h>
vector float sqrtf_v(vector float value)

C Specification (SPE only)

#include <sqrt.h>
inline double _sqrt_d(double value)

#include <sqrt_v.h>
inline vector double _sqrt_v(vector double value)

#include <libmath.h>
double sqrt_d(double value)

#include <libmath.h>
vector double sqrt_dv(vector double value)

Aliases (SPE only)

#include <libmath.h>
long double sqrtl(long double value)

Descriptions

The *sqrt* subroutines compute the square root of the number (or vector of numbers) specified by the *value* input parameter. Computation of the square root exploits the reciprocal square root subroutine (*inv_sqrt*) since sqrt(x) = x * 1.0 / sqrt(x).

The result is accurate to single precision floating-point for the *sqrtf* and *sqrtf_v* subroutines.

The result is accurate to double precision floating-point for the *sqrt* and *sqrt_v* subroutines. In addition, special handling for exceptional values are provided. If the input angle is an infinity, then the result is infinity. If the input is less than 0 or a NaN (Not a Number), then the result is a NaN. If the input is a denorm, then the result is 0.

The *sqrtl* subroutine is aliased to the *sqrt* subroutine.

Dependencies

inv_sqrt on page 232

See Also

inv_sqrt on page 232

11.59 tan

C Specification

```
#include <tanf.h>
inline float _tanf(float angle)

#include <tanf_v.h>
inline vector float _tanf_v(vector float angle)

#include <libmath.h>
float tanf(float angle)

#include <libmath.h>
vector float tanf_v(vector float angle)
```

C Specification (SPE only)

```
#include <tan.h>
inline double _tan(double angle)

#include <tan_v.h>
inline vector double _tan_v(vector double angle)

#include <libmath.h>
double tan(double angle)

#include <libmath.h>
vector double tan_v(vector double angle)
```

Aliases (SPE only)

```
#include <libmath.h>
long double tanl(long double angle)
```

Descriptions

The *tanf* subroutine computes the tangent of the input angle (in radians), specified by the parameter *angle*, to an accuracy of single precision floating point. The tangent function is computed using a sign corrected ratio of approximating sine and cosine polynomials for the range of input angles $[0.0, \pi/4]$. The entire range of supported input angles is implemented by observing the symmetry of the tangent function over the region $[0, 2*\pi)$ and the cyclic nature over the Reals.

The *tanf_v* subroutine computes the tangent on a vector (4 independent) input angles.

The *tan* and *tan_v* compute the scalar and vector tangents, respectively, to an accuracy of double precision floating point.

The *tanl* subroutine is aliased to the *tan* subroutine.

Dependencies

cos on page 189
sin on page 259
divide (floating point) on page 196

See Also

cot on page 191
atan on page 183
cos on page 189
sin on page 259

11.60 trunc

C Specification (SPE only)

```
#include <truncf.h>
inline float _truncf(float x)

#include <truncf_v.h>
inline vector float _truncf_v(vector float x)

#include <trunc.h>
inline double _trunc(double x)

#include <trunc_v.h>
inline vector double _trunc_v(vector double x)

#include <libmath.h>
float truncf(float x)

#include <libmath.h>
vector float truncf_v(vector float x)

#include <libmath.h>
double trunc(double x)

#include <libmath.h>
vector double trunc_v(vector double x)
```

Aliases (SPE only)

```
#include <libmath.h>
long double truncl(long double x)
```

Descriptions

The *trunc* subroutines round the input(s) specified by the parameter *x* to the nearest integer not larger in aboslute value in floating-point format. If *x* is infinite, NaN, or integral, then *x* itself is returned.

The vectored subroutines, *truncf_v* and *trunc_v*, perform the truncation indendently on each 4 and 2 elements, respectively.

The *truncl* subroutine is aliased to the *trunc* subroutine.

Dependencies

See Also

floor on page 217
llround on page 236
lrint on page 243
nearbyint on page 248

12. 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.

msb			lsb
X	Y	Z	W

12.1 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

12.2 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

inverse on page 231

See Also

ortho_matrix4x4 on page 274
perspective_matrix4x4 on page 275

12.3 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:

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

Dependencies

See Also

12.4 inverse_matrix4x4

C Specification

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

#include <libmatrix.h>
void 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 imporvements over simple scalar code.

Dependencies

See Also

12.5 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.*

$$\left[out\right] = \left[m1\right] X \left[m2\right]$$

Both single precision and double precision matrix multiplies are supported.

Dependencies

See Also

12.6 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 quaterions *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 276
rot_matrix4x4_to_quat on page 278

12.7 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

inverse on page 231

See Also

frustum_matrix4x4 on page 269
perspective_matrix4x4 on page 275

12.8 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:

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

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

Dependencies

inverse on page 231
cot on page 191

See Also

ortho_matrix4x4 on page 274
perspective_matrix4x4 on page 275

12.9 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:

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

Dependencies

See Also

rot_matrix4x4_to_quat on page 278

12.10 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:

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

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

Dependencies

See Also

12.11 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 276

12.12 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.*

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

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

Dependencies

See Also

12.13 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:

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

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

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

Dependencies

sin on page 259
divide (floating point) on page 196
acos on page 180

See Also

rot_matrix4x4_to_quat on page 278
quat_to_rot_matrix4x4 on page 276

12.14 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

12.15 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

13. 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>

13.1 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 on page 73

See Also

free_align on page 290
malloc_align on page 292
realloc_align on page 304

13.2 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 286
clamp_minus1_to_1 on page 287

13.3 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 285
clamp_minus1_to_1 on page 287

13.4 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 286
clamp_0_to_1 on page 285

13.5 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 on page 39

See Also

copy_to_ls on page 289

13.6 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 on page 39

See Also

copy_from_ls on page 288

13.7 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 on page 74

See Also

calloc_align on page 284
malloc_align on page 292
realloc_align on page 304

13.8 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 305

13.9 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 on page 75

See Also

calloc_align on page 284
free_align on page 290
realloc_align on page 304

13.10 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 295
max_int_v on page 294
min_float_v on page 297

13.11 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 296
max_float_v on page 293
min_int_v on page 298

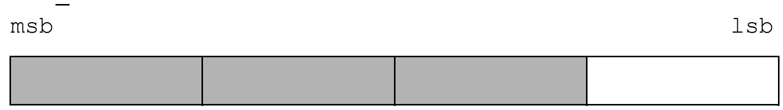
13.12 max_vec_float

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 296
max_vec_float on page 295

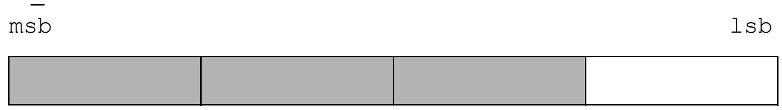
13.13 max_vec_int

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 295
min_vec_int on page 300

13.14 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 299
min_int_v on page 298
max_float_v on page 293

13.15 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 300
min_float_v on page 297
max_int_v on page 294

13.16 min_vec_float

C Specification

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

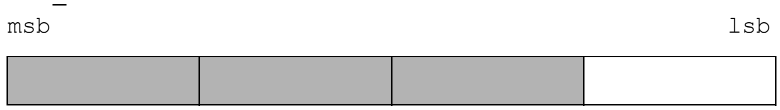
#include <min_vec_float4.h>
inline float _min_vec_float4(vector float v_in)

#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 300
max_vec_float on page 295

13.17 min_vec_int

C Specification

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

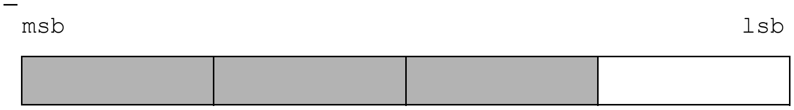
#include <min_vec_int4.h>
inline signed int _min_vec_float4(vector signed int v_in)

#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 299
max_vec_int on page 296

13.18 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 306
rand on page 42
rand_0_to_1 on page 303
rand_minus1_to_1 on page 302

13.19 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 42

See Also

srand on page 44
rand_0_to_1 on page 303

13.20 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 42

See Also

srand on page 44
rand_minus1_to_1 on page 302

13.21 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*.

Note: The contents of the buffer are preserved only if the requested alignment is the same as the alignment in which the buffer was originally allocated.

Dependencies

realloc on page 76

See Also

calloc_align on page 284
free_align on page 290
malloc_align on page 292

13.22 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 291

13.23 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.

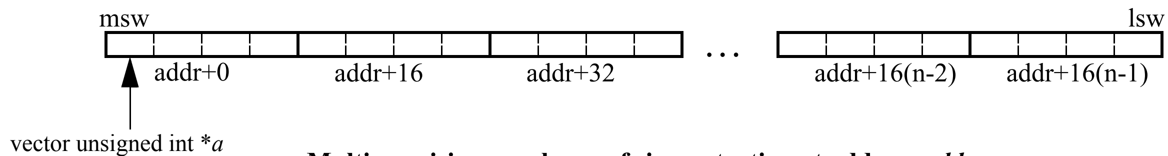
Dependencies

See Also

rand on page 301 or *rand* on page 42
rand_0_to_1 on page 303
rand_minus1_to_1 on page 302
srand on page 44

14. 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>

14.1 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 327

See Also

14.2 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 310
mpm_sub on page 330

14.3 mpm_add_partial

C Specification

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

#include <libmpm.h>
void mpm_add_partial(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 a output. All numbers are of

This function can be used to significantly improve the performance accumulating multiple multi-precision numbers. For example, the 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 309

14.4 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 312
mpm_cmpgt on page 313

14.5 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* quadwords. 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 311
mpm_cmpgt on page 313

14.6 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 311
mpm_cmpge on page 312

14.7 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 318
mpm_mul on page 324

14.8 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 313
mpm_sub on page 330

See Also

mpm_mod_exp on page 319
mpm_mod on page 318

14.9 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 313
mpm_mod on page 318

See Also

mpm_div on page 314

14.10 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 324
mpm_add on page 309
mpm_add_partial on page 310

14.11 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 313
mpm_sub on page 330

See Also

mpm_div on page 314

14.12 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 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 324
mpm_div on page 314
mpm_square on page 329
mpm_fixed_mod_reduction on page 315

See Also

mpm_mont_mod_exp on page 321

14.13 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,
                              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,
                              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 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_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 that the sliding window size is constant and equals MPM_MOD_EXP_MAX_K.

Dependencies

mpm_mod on page 318
mpm_mul_inv on page 325
mpm_mont_mod_mul on page 323

See Also

mpm_mod_exp on page 319

14.14 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 330

See Also

mpm_mod on page 318
mpm_mont_mod_exp on page 321

14.15 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 317
mpm_add_partial on page 310

14.16 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 $0 < mi < a$.

Dependencies

mpm_add on page 309
mpm_cmpge on page 312
mpm_cmpgt on page 313

mpm_div on page 314
mpm_mod on page 318
mpm_mul on page 324
mpm_sizeof on page 328
mpm_sub on page 330

See Also

14.17 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 308

14.18 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

14.19 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 324
mpm_add_partial on page 310

14.20 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 309

14.21 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

15. Noise LibraryPPE

The noise library is supported on both the PPE and SPE. The noise libraries provides functions for:

1. 1-D, 2-D, 3-D and 4-D noise
2. Lattice and non-latice noise
3. Turbulance.

Name(s)

libnoise.a

Header File(s)

<libnoise.h>

15.1 noise1, noise2, noise3, noise4

C Specification

```
#include <noise1.h>
inline float _noise1(float x)

#include <noise1_v.h>
inline vector float _noise1_v(vector float x)

#include <noise2.h>
inline float _noise2(float x, float y)

#include <noise2_v.h>
inline vector float _noise2_v(vector float x, vector float y)

#include <noise3.h>
inline float _noise3(float x, float y, float z)

#include <noise3_v.h>
inline vector float _noise3_v(vector float x, vector float y, vector float z)

#include <noise4.h>
inline float _noise4(float x, float y, float z, float w)

#include <noise4_v.h>
inline vector float _noise4_v(vector float x, vector float y, vector float z, vector float w)

#include <libnoise.h>
float noise1(float x)

#include <libnoise.h>
vector float noise1_v(vector float x)

#include <libnoise.h>
float noise2(float x, float y)

#include <libnoise.h>
vector float noise2_v(vector float x, vector float y)

#include <libnoise.h>
float noise3(float x, float y, float z)

#include <libnoise.h>
vector float _noise3_v(vector float x, vector float y, vector float z)

#include <libnoise.h>
float noise4(float x, float y, float z, float w)
```

```
#include <libnoise.h>
vector float noise4_v(vector float x, vector float y, vector float z, ector float w)
```

Descriptions

The *noise* subroutines implement coherent pseudo-random functions across 1-4 dimensions. The *noise* functions are repeatable, in that they return one result given the same set of inputs. The computed result is in the domain of [-0.7..0.7].

The *noise* subroutines implement a simple and efficient kind of noise known as *lattice noise*. The noise values pass through 0 when the input coordinates arrive at whole integer lattice points.

The *noise* subroutines are based on Perlin's original C implementation, but have been extended and modified as follows: (A) The format of the inputs and outputs are single precision floating point, where Perlin originally used double precision; (B) Noise values can be computed for 1-4 dimensions, where Perlin originally implemented 1-3; (C) Both scalar and vector versions of the routines are provided; (D) Perlin's original permutation and gradient tables have been replaced with hash functions that compute gradient vectors on the fly, which is better suited to SIMD computation and does not require storage overhead; (E) The *noise* subroutines compute pseudo-random gradient vectors from 16-bit integer seeds, where Perlin's original tables restricted the seeds to 8-bit integers.

Dependencies

See Also

Texturing and Modeling, A Procedural Approach (Ebert, et al, 2nd ed).
Ken Perlin's NYU home page: http://mrl.nyu.edu/perlin/
Slides on the history of Perlin noisee: http://noisemachine.com/talk1/

15.2 vlnoise1, vlnoise2, vlnoise3, vlnoise4

C Specification

```
#include <vlnoise1.h>
inline float _vlnoise1(float x)

#include <vlnoise1_v.h>
inline vector float _vlnoise1_v(vector float x)

#include <vlnoise2.h>
inline float _vlnoise2(float x, float y)

#include <vlnoise2_v.h>
inline vector float _vlnoise2_v(vector float x, vector float y)

#include <vlnoise3.h>
inline float _vlnoise3(float x, float y, float z)

#include <vlnoise3_v.h>
inline vector float _vlnoise3_v(vector float x, vector float y, vector float z)

#include <vlnoise4.h>
inline float _vlnoise4(float x, float y, float z, float w)

#include <vlnoise4_v.h>
inline vector float _vlnoise4_v(vector float x, vector float y,
                                vector float z, vector float w)

#include <libnoise.h>
float vlnoise1(float x)

#include <libnoise.h>
vector float vlnoise1_v(vector float x)

#include <libnoise.h>
float vlnoise2(float x, float y)

#include <libnoise.h>
vector float vlnoise2_v(vector float x, vector float y)

#include <libnoise.h>
float vlnoise3(float x, float y, float z)

#include <libnoise.h>
vector float vlnoise3_v(vector float x, vector float y, vector float z)

#include <libnoise.h>
float vlnoise4(float x, float y, float z, float w)
```

```
#include <libnoise.h>
vector float vlnoise4_v(vector float x, vector float y, vector float z, vector float w)
```

Descriptions

The *vlnoise* subroutines implement coherent pseudo-random functions across 1-4 dimensions. The *vlnoise* functions are repeatable, in that they return one result given the same set of inputs. The computed result is in the domain of [-0.7..0.7].

The *vlnoise* subroutines implement a kind of noise known as *variable lattice noise*. Instead of using whole integer lattice points as the noise function does, the *vlnoise* function generates pseudo-random lattice points in the range of [0..1]. The *vlnoise* function passes through 0 when the input coordinates arrive at the pseudo-random lattice points. vlnoise can be less “boxy” than traditional noise.

Dependencies

See Also

Texturing and Modeling, A Procedural Approach (Ebert, et al, 2nd ed).

15.3 fractalsum1, fractalsum2, fractalsum3, fractalsum4

C Specification

```
#include <fractalsum1.h>
inline float _fractalsum1(float x, float minfreq, float maxfreq)

#include <fractalsum1_v.h>
inline vector float _fractalsum1_v(vector float x, vector float minfreq,
                                   vector float maxfreq)

#include <fractalsum2.h>
inline float _fractalsum2(float x, float y, float minfreq, float maxfreq)

#include <fractalsum2_v.h>
inline vector float _fractalsum2_v(vector float x, vector float y, vector float minfreq,
                                   vector float maxfreq)

#include <fractalsum3.h>
inline float _fractalsum3(float x, float y, float z, float minfreq, float maxfreq)

#include <fractalsum3_v.h>
inline vector float _fractalsum3_v(vector float x, vector float y, vector float z,
                                   vector float minfreq, vector float maxfreq)

#include <fractalsum4.h>
inline float _fractalsum4(float x, float y, float z, float w, float minfreq, float maxfreq)

#include <fractalsum4_v.h>
inline vector float _fractalsum4(vector float x, vector float y, vector float z,
                                   vector float w, vector float float minfreq,
                                   vector float maxfreq)

#include <libnoise.h>
float fractalsum1(float x, float minfreq, float maxfreq)

#include <libnoise.h>
vector float fractalsum1_v(vector float x, vector float minfreq, vector float maxfreq)

#include <libnoise.h>
float fractalsum2(float x, float y, float minfreq, float maxfreq)

#include <libnoise.h>
vector float fractalsum2_v(vector float x, vector float y, vector float minfreq,
                                   vector float maxfreq)

#include <libnoise.h>
float fractalsum3(float x, float y, float z, float minfreq, float maxfreq)

#include <libnoise.h>
```

```
vector float fractalsum3_v(vector float x, vector float y, vector float z, vector float freq,
                           vector float ampl)

#include <libnoise.h>
float fractalsum4(float x, float y, float z, float w, float minfreq, float maxfreq)

#include <libnoise.h>
vector float fractalsum4_v(vector float x, vector float y, vector float z,vector float w,
                           vector float minfreq, vector float maxfreq)
```

Descriptions

The *fractalsum* subroutines implement the equivelent of Perlin's original *turbulence* function. The *fractalsum* subroutines are layered on top of multiple octaves of coherent *noise.*

The value of *minfreq* must be less than or equal to *maxfreq* or the function will not terminate. The value returned is in the range [-1..1).

Dependencies

noise1, noise2, noise3, noise4 on page 334

See Also

Texturing and Modeling, A Procedural Approach (Ebert, et al, 2nd ed).

15.4 turb1, turb2, turb3, turb4

C Specification

```
#include <turb1.h>
inline float _turb1(float x, float minfreq, float maxfreq)

#include <turb1_v.h>
inline vector float _turb1_v(vector float x, vector float minfreq, vector float maxfreq)

#include <turb2.h>
inline float _turb2(float x, float y, float freq, float ampl)

#include <turb2_v.h>
inline vector float turb2_v(vector float x, vector float y, vector float minfreq,
                            vector float maxfreq)

#include <turb3.h>
inline float _turb3(float x, float y, float z, float minfreq, float maxfreq)

#include <turb3_v.h>
inline vector float _turb3_v(vector float x, vector float y, vector float z,
                             vector float minfreq, vector float maxfreq)

#include <turb4.h>
inline float _turb4(float x, float y, float z, float w, float minfreq, float maxfreq)

#include <turb4_v.h>
inline vector float _turb4_v(vector float x, vector float y, vector float z, vector float w,
                             vector float float minfreq, vector float maxfreq)

#include <libnoise.h>
float turb1(float x, float minfreq, float maxfreq)

#include <libnoise.h>
vector float turb1_v(vector float x, vector float minfreq, vector float maxfreq)

#include <libnoise.h>
float turb2(float x, float y, float minfreq, float maxfreq)

#include <libnoise.h>
vector float turb2_v(vector float x, vector float y, vector float minfreq,
                     vector float maxfreq)

#include <libnoise.h>
float turb3(float x, float y, float z, float minfreq, float maxfreq)

#include <libnoise.h>
vector float turb3_v(vector float x, vector float y, vector float z, vector float minfreq,
                     vector float maxfreq)
```

```
#include <libnoise.h>
float turb4(float x, float y, float z, float w, float minfreq, float maxfreq)

#include <libnoise.h>
vector float turb4_v(vector float x, vector float y, vector float z,vector float w,
                     vector float minfreq, vector float maxfreq)
```

Descriptions

The *turb* subroutines implement turbulent *noise* (a.k.a. fractal Browninan motion) by iteratively scaling *noise* values.

The frequency begins at *minfreq*, and is scaled by 2.0 until *maxfreq* is reached. The value of *minfreq* must be less than or equal to *maxfreq*, or the function will not terminate. The value returned is in the range of [-1..1).

Dependencies

noise1, noise2, noise3, noise4 on page 334
fabs on page 203

See Also

Texturing and Modeling, A Procedural Approach (Ebert, et al, 2nd ed).
Ken Perlin’s NYU home page: http://mrl.nyu.edu/perlin/
Slides on the history of Perlin noisee: http://noisemachine.com/talk1/

16. Oscillator Libraries

The two oscillator libraries described in this chapter support the creation of a synthetic environment consisting of one or more configurable directional microphones, listening to a large number of oscillators moving along user-defined paths, relative to the static microphones.

The environment correctly computes time delays, volume changes, and doppler effects based on the positions of the oscillators.

Oscillators continue to vibrate and move according to their input parameters until the damped amplitude drops below the threshold of audibility, at which time the oscillator goes “inactive” and can be redefined.

Oscillator inactivation can be overriden by specifying that the oscillator be “locked”. Locked oscillators will continue to move and emit sound, no matter how inaudible the sounds are. This is useful for oscillators that are only audible when they are very close to the ear (microphone), like a mosquito.

These libraries create “frames” of sound, which are presumed to be synchronized with video frames in a graphics application. This is, one frame might correspond to 1/24 or 1/30 of a second.

The inputs to the oscillator libraries include:
- the position of each microphone
- the “optimal listening direction” of each microphone
- the attenuation of each microphone in directions away from optimal
- the waveform to be used for each oscillator
- the frequency of each oscillator
- the amplitude and damping factors for each oscillator
- the position of each oscillator at the begining and end of each frame

The output from the oscillator libraries include:
- an array of interleaved sound data (unsigned short), for each frame, which can be strung together to create a long sound file in raw format, suitable for playing through low-level sound drivers.

Name(s)

liboscillator.a

Header File(s)

<liboscillator.h>

16.1 Constants, Macros, and Structures

The libraries define and make used of four constants. They are:

`MAX_MICS`	The maximum number of microphones that the synthetic environment will handle. The library default is 6.
`MAX_WAVEFORMS`	The maximum number of distinct waveforms that the synthetic environment will handle. Each oscillator to one of the user-defined waveforms. The typical waveform that the user may define would be a sine wave or a triangle wave. The library default is 8.
`OSC_L2_WAVEFORM_SIZE`	The user specified waveforms are specified as the number of samples per cycle of the waveform. This number must be a power of 2, and the log (base 2) of this number is specified by this constant. The library default is 6 corresponding to 64 sample waveforms.
`SPEED_OF_SOUND`	The speed of sound (in feet per second). Users can modify this constant to accomodate speeds based upon differing conditions. The library default is 1116.0 feet/second.

The libraries define four query macros for the convenience of the user.

`OSC_ACTIVE(osc, idx)`	Returns true iff the oscillator specified by the *idx* parameter is defined and active. The *osc* parameter is a pointer to the osc_data structure.
`OSC_NONZERO(osc)`	Returns true iff the environment has a non-zero number of oscillators defined. The *osc* parameter is a pointer to the osc_data structure.
`OSC_LOCKED(osc, idx)`	Returns true iff the oscillator specified by the *idx* parameter is defined as "locked". The *osc* parameter is a pointer to the osc_data structure.

The libraries define and make use of three structures. They are:

`mic_osc_data`

Contains the details for each microphone-oscillator pair. The structures are created and updated by the PPE oscillator library and used by the SPE oscillator library. Five fields of the structure are:

signal	location in waveform at start of frame
dsignal	delta location over the frame
o_start	frame where oscillator starts
agg_amp	the overall amplitude
d_agg_amp	delta amplitude over the frame

Users of these fields (arrays) need not understand their details, just accept them from the PPE oscillator library and pass them to the SPE oscillator library.

`osc_pos`	Contains position of each oscillator (in world coordinates, not relative to any microphone) at the beginning and at the end of the frame in question.
`osc_data`	Contains global data for the entire oscillator environment. It include the total oscillators in play, an array of flags marking active oscillators, an array of flags marking locked oscillators, an array of pointers to the waveforms, and pointers to the *mic_osc_data* and *osc_pos* structures.

16.2 PPE Oscillator Subroutines

16.2.1 osc_add

C Specification

```
#include <liboscillator.h>
int osc_add(float frequency, float amplitude, float damping_value,
            float start_time_fraction, int waveform_index,
            float *start_position, float *end_position,
            int lock_flag)
```

Descriptions

The *osc_add* subroutine creates and adds an oscillator into the environment and returns a index identifying the oscillator.

The *frequency* parameter specifies the frequency of the oscillator. For example, if the *frequency* is 440.0, then the oscillator will process through 440 copies of its waveform every second.

The *amplitude* parameter specifies the signal amplitude of the output sound. This number must be in the range 0.0 to 1.0.

The *damping_value* parameter specifies a measure of how fast the oscillator reduces its volume over time. If the the *damping_value* is 1.0, the oscillator continues at full amplitude forever. If the *damping_value* is 0.5, the oscillator reduces its volume by 50% every frame.

The *start_time_fraction* parameter specifies exactly when (within the frame in question) the oscillator starts to make noise. This should be a number between 0.0 and 1.0.

The *waveform_index* parameter specifies the oscillator’s waveform. The waveform is identified by the index returned from the *osc_add_waveform* subroutine.

The *start_position* parameter specifies the 3-D world coordinate location of the oscillator at the start of the frame. World coordinates are specified in feet.

The *end_position* parameter specifies the 3-D world coordinate location of the oscillator at the end of the frame, in feet.

The *lock_flag* parameter specifies whether this oscillator is to be locked. A locked oscillator can not be deleted and is never inactivated no matter how quiet its signal becomes.

Dependencies

memcpy
fprintf
exp
log

See Also

osc_delete on page 348
osc_init on page 349

16.2.2 osc_delete

C Specification

```
#include <liboscillator.h>
void osc_delete(int osc_index)
```

Descriptions

The *osc_delete* subroutine informs the environment that the oscillator specified by the *osc_index* parameter is to be turned off, and immediately available for re-use. The *osc_index* corresponds to the value returned by the *osc_add* subroutine.

Note: Requests to delete locked oscillators are ignored.

Dependencies

See Also

osc_add on page 346
osc_init on page 349
osc_init_microphone on page 350
osc_update_for_new_frame on page 351
osc_add_waveform on page 352
osc_delete_waveform on page 353
osc_init_spu_machine on page 354
osc_produce_a_frame_of_sound on page 355

16.2.3 osc_init

C Specification

```
#include <liboscillator.h>
osc_data * osc_init(int frame_hz, int output_hz, int n_oscs, float maxdist, int nmic)
```

Descriptions

The *osc_init* subroutine sets up the initial structures and data for the synthetic environment and returns a pointer to the *osc_data* structure.

The *frame_hz* parameter specifies the number of frames to be produced for each second of output sound.

The *output_hz* parameter specifies the signal frequency of the output sound.

The *n_oscs* parameter specifies the maximum number of oscillators that the environment is being requested to handle simultaneously.

The *maxdist* parameter specifies the maximum predicted distance between any oscillator and any microphone. If the oscillator moves beyond this distance, the environment is permitted to truncate the location of the oscillator to this maximum distance.

The *nmics* parameter specifies the maximum number of microphones that the environment is being requested to handle simultaneously. *nmics* must be in the range 0 to MAX_MICS

Dependencies

malloc
sqrt

See Also

osc_add on page 346
osc_delete on page 348
osc_init_microphone on page 350
osc_update_for_new_frame on page 351
osc_add_waveform on page 352
osc_delete_waveform on page 353
osc_init_spu_machine on page 354
osc_produce_a_frame_of_sound on page 355

16.2.4 osc_init_microphone

C Specification

```
#include <liboscillator.h>
void osc_init_microphone(int mic_number, float *mic_position, float *mic_direction,
                                             float *mic_attenuation)
```

Descriptions

The *osc_init_microphone* subroutine sets data associated with a user defined microphone.

The *mic_number* parameter specifies the microphone to be defined. *mic_number* must be in the range 0 to *nmics*-1 (specified when the oscillator environment is initialized by the *osc_init* subroutine.

The *mic_position* parameter specifies the 3-D world space coordinate location of the microphone. 3-D world space coordinates are expressed in feet from an origin in the user's world coordinate system.

The *mic_direction* parameter is a 3-D world space vector that specifies the direction that the microphone is listening with highest efficiency.

The *mic_attenuation* parameter is an array of eleven attenuation values, corresponding to the efficiency of the microphone at listening to oscillators located 0, 18, 36, ..., and 180 degrees off the *mic_direction*. Normally, a directional microphone would be specified with a *mic_attenuation*[0] = 1.0 and the remainder of the array containing values that drop off from 1.0 towards 0.0.

Dependencies

sqrt

See Also

osc_add on page 346
osc_delete on page 348
osc_init on page 349
osc_update_for_new_frame on page 351
osc_add_waveform on page 352
osc_delete_waveform on page 353
osc_init_spu_machine on page 354
osc_produce_a_frame_of_sound on page 355

16.2.5 osc_update_for_new_frame

C Specification

```
#include <liboscillator.h>
void osc_update_for_new_frame(int osc_index, float *osc_position)
```

Descriptions

The *osc_update_for_new_frame* subroutine updates the oscillator specified by the *osc_index* with a new position. The new positiion, expressed as a 3-D world space coordinate, is specified by the *osc_position* parameter. The position corresponds to the location the oscillator is to move to during the next frame.

Dependencies
memcpy

See Also
osc_add on page 346
osc_delete on page 348
osc_init on page 349
osc_init_microphone on page 350
osc_add_waveform on page 352
osc_delete_waveform on page 353
osc_init_spu_machine on page 354
osc_produce_a_frame_of_sound on page 355

16.3 SPE Oscillator Subroutines

16.3.1 osc_add_waveform

C Specification

```
#include <liboscillator.h>
int osc_add_waveform(float *waveform_data)
```

Descriptions

The *osc_add* subroutine loads a waveform into the SPE oscillator environment and returns the index of the waveform.

The waveform, specified by the *waveform_data* parameter, is an array of (1<<OSC_L2_WAVEFORM_SIZE) floats, each in the range -1.0 to 1.0.

Dependencies

See Also

osc_add on page 346
osc_delete on page 348
osc_init on page 349
osc_init_microphone on page 350
osc_update_for_new_frame on page 351
osc_delete_waveform on page 353
osc_init_spu_machine on page 354
osc_produce_a_frame_of_sound on page 355

16.3.2 osc_delete_waveform

C Specification

```
#include <liboscillator.h>
void osc_delete_waveform(int waveform_index)
```

Descriptions

The *osc_delete_waveform* subroutine places the waveform specified by the *waveform_index* back on the available waveform list. The *waveform_index* must correspond to an index returned by the osc*_add_waveform* subroutine.

Dependencies

See Also

osc_add on page 346
osc_delete on page 348
osc_init on page 349
osc_init_microphone on page 350
osc_update_for_new_frame on page 351
osc_add_waveform on page 352
osc_init_spu_machine on page 354
osc_produce_a_frame_of_sound on page 355

16.3.3 osc_init_spu_machine

C Specification

```
#include <liboscillator.h>
int osc_init_spu_machine(int num_samples_per_frame, int max_num_waveforms, int nmics)
```

Descriptions

The *osc_init_spu_machine* subroutine initiailize the SPE portion of the oscillator environment.

The *num_samples_per_frame* parameter specifies the number of signal samples to be produced per frame of sound.

The *max_num_waveforms* parameter specifies the maximum number of simultaneous waveforms the oscillator machine should support.

The *nmics* parameter specifies the maximum number of simultaneous microphones the oscillator machine should support. *nmics* must be in the range 0 to MAX_MICS.

Upon successful initiaization, zero is returned. A non-zero value is return if *max_num_waveforms* or *nmics* is out of range.

Dependencies

See Also

osc_add on page 346
osc_delete on page 348
osc_init on page 349
osc_init_microphone on page 350
osc_update_for_new_frame on page 351
osc_add_waveform on page 352
osc_delete_waveform on page 353
osc_produce_a_frame_of_sound on page 355

16.3.4 osc_produce_a_frame_of_sound

C Specification

```
#include <liboscillator.h>
void osc_produce_a_frame_of_sound(void *data_pointer, int noscs_vector,
                                   unsigned short *generated_sound)
```

Descriptions

The *osc_produce_a_frame_of_sound* subroutine creates a frame of raw sound samples.

The *noscs_vector* parameter specifies how many oscillators are being communicated to this routine, in terms of quadwords. For example, if there are 59 oscillators, *noscs_vector* will contain the value 15 ((59 + 3)/4).

The *data_pointer* parameter specifies an array of data consisting of the concatenation of the following:

- *noscs_vector* quadwords of flags indicating which of the oscillators are active (one flag per word).
- *noscs_vector* quadwords of waveform indices, indicating which waveform is associated with each oscillator.
- *nmics* copies of the following five arrays:
  - *noscs_vector* quadwords of "signal" data
  - *noscs_vector* quadwords of "dsignal" data
  - *noscs_vector* quadwords of "o_start" data
  - *noscs_vector* quadwords of "agg_amp" data
  - *noscs_vector* quadwords of "d_agg_amp" data

So, for example, if there are 59 oscillators and 4 microphones, 59 is rounded up to 60 and *data_pointer* should point to (60/4)*(2+4*5) = 330 quadwords of data.

The *generated_sound* parameter specifies where the raw generated sound data is to be placed. The number of output sound data samples is the product of the number of microphones and the number of samples per frame. The output data ranges from 0 to 65535, with a signal of pure silence coded as a series of 32768's. The data is ordered such that all microphone samples for a given time tick, followed by all the microphone samples for the following time tick.

Dependencies

sum_across_float on page 414

See Also

osc_add on page 346
osc_delete on page 348
osc_init on page 349
osc_init_microphone on page 350
osc_update_for_new_frame on page 351
osc_add_waveform on page 352
osc_delete_waveform on page 353
osc_init_spu_machine on page 354

17. Simulation Library

The simulation library provides an assortment of services useful only in the simulation environment provided by the IBM Full System Simulator for the Cell Broadband Engine - either running in standalone mode or running the Linux Operating System.

Note: These function are only available on the simulator and should be avoided in any code that is destined to run on real hardware.

17.1 Library Subroutines

Library Name(s)

libsim.a

Header File(s)

<libsim.h>
<sim_printf.h>

17.1.1 sim_close

C Specification (SPE only)

```
#include <libsim.h>
int sim_close(int fd)
```

Descriptions

The *sim_close* subroutine closes the file descriptor specified by the *fd* parameter. This function is accomplished by issuing a syscall 2 to the simulator which in turn perfoms a “close” on the simulator host. For additional details, consult the simulator host documentation.

See Also

sim_close on page 358
sim_lseek on page 361
sim_open on page 362
sim_read on page 364
sim_write on page 367

17.1.2 sim_cycles

C Specification (SPE only)

```
#include <libsim.h>
int sim_cyles()
```

Descriptions

The *sim_cycles* subroutine fetches the current cycle count from the simulator by issuing a syscall 43 to the simulator. The cycle count is only valid when the simulator in run in “pipeline” mode.

See Also

sim_instructions on page 360

17.1.3 sim_instructions

C Specification (SPE only)

```
#include <libsim.h>
int sim_instructions()
```

Descriptions

The *sim_instructions* subroutine fetches the current instruction count from the simulator by issuing a syscall 42 to the simulator.

See Also

sim_cycles on page 359

17.1.4 sim_lseek

C Specification (SPE only)

```
#include <libsim.h>
int sim_lseek(int fd, int offset, int whence)
```

Descriptions

The *sim_lseek* subroutine performs a seek on the file specified by the file descriptor *fd* parameter. The lseek function positions the offset to the argument *offset* according to the directive *whence.* This function is accomplished by issuing a syscall 52 to the simulator which in turn perfoms a “lseek” on the simulator host. For additional details, consult the simulator host documentation.

See Also

sim_close on page 358
sim_open on page 362
sim_read on page 364
sim_write on page 367

17.1.5 sim_open

C Specification (SPE only)

```
#include <libsim.h>
int sim_open(const char *pathname, int flags)
```

Descriptions

The *sim_open* subroutine opens the file specified by *pathname* according to the *flags* parameter and returns the file-descriptor of the open file. This function is accomplished by issuing a syscall 1 to the simulator which in turn perfoms a “open” on the simulator host. For additional details, consult the simulator host documentation.

See Also

sim_close on page 358
sim_lseek on page 361
sim_read on page 364
sim_write on page 367

17.1.6 sim_printf

C Specification (SPE only)

```
#include <sim_printf.h>
int sim_printf(char *format, ...)
```

Descriptions

sim_printf is the ANSI compatible printf. This function is accomplished by issuing a syscall 21 to the simulator which in turn perfoms a “printf” on the simulator host.

See Also

17.1.7 sim_read

C Specification (SPE only)

```
#include <libsim.h>
int sim_read(int fd, void *buf, int count)
```

Descriptions

The *sim_read* subroutine performs of read of *count* bytes of data from the open file specified by the file descriptor *fd*. This function is accomplished by issuing a syscall 3 to the simulator which in turn perfoms a “read” on the simulator host. For additional details, consult the simulator host documentation.

See Also

sim_close on page 358
sim_lseek on page 361
sim_open on page 362
sim_write on page 367

17.1.8 sim_start_timer

C Specification (SPE only)

```
#include <libsim.h>
void sim_open(const cahr *pathname, int flags)
```

Descriptions

The *sim_start_timer* subroutine starts the simulator timed by issuing a syscall 13 to the simulator.

See Also

sim_stop_timer on page 366

17.1.9 sim_stop_timer

C Specification (SPE only)

```
#include <libsim.h>
void sim_stop_timer()
```

Descriptions

The *sim_stop_timer* subroutine stops the running timer and prints to the simulators stdout the amount of real time passed since the timer was started. This function is accomplished by issuing a syscall 14 to the simulator

See Also

sim_start_timer on page 365

17.1.10 sim_write

C Specification (SPE only)

```
#include <libsim.h>
int sim_write(const cahr *pathname, int flags)
```

Descriptions

The *sim_write* subroutine performs of write of *count* bytes of data from the open file specified by the file descriptor *fd*. This function is accomplished by issuing a syscall 4 to the simulator which in turn perfoms a “write” on the simulator host. For additional details, consult the simulator host documentation.

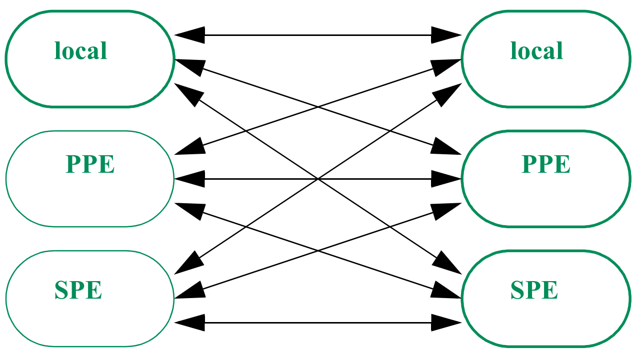
See Also

sim_close on page 358
sim_lseek on page 361
sim_open on page 362
sim_read on page 364

17.2 Connection Subroutines

The simulation environment also supports an IPC (inter-process communications) construct known as *Connections*. Connections are nothing more than uni-directional Unix sockets. A connection is identified by its *name* and *connectionType.* The name specifies a filename whose file descriptor is used for the socket and connection type specfies the direction of the socket. Valid *connectionTypes* include CONNECTION_PRODUCER and CONNECTION_CONSUMER.

Connections can be established between 2 processes, regardless of the type of processor being executed. Seperate files and header files are provided to support the connection subroutines for each of the processors..



Files

connection.c
(compiled to connection.o)

connection_ppu.h

connection_spu.h

17.2.1 closeConnection

C Specification

void closeConnection(char *name, int connectionType)

Description

The *closeConnection* subroutine closes a connection for the filename specfied by the *name* parameter of the type specified by the *connectionType* parameter. If the connection being closed is current, then the current connection for that type is set to **null**. Valid connectionsTypes include CONNECTION_PRODUCER and CONNECTION_CONSUMER.

See Also

openConnection on page 370
selectConnection on page 371

17.2.2 openConnection

C Specification

void openConnection(char *name, int connectionType)

Description

The *openConnection* subroutine opens a connection for the filename specfied by the *name* parameter of the type specified by the *connectionType* parameter and makes the connection current. Valid connectionTypes include CONNECTION_PRODUCER and CONNECTION_CONSUMER. A consumer connection must be opened (created) before a producer connection can be opened.

See Also

selectConnection on page 371
closeConnection on page 369

17.2.3 selectConnection

C Specification

void selectConnection(char *name, int connectionType)

Description

The *selectConnection* subroutine makes the connection specified by the *name* and *connectionType* parameters current. Each “execution” task has two current connections - one consumer and one producer connection.

See Also

openConnection on page 370
closeConnection on page 369

17.2.4 receiveData

C Specification (Local only)

int receiveData(char *ptr, int len)

void receiveDataBlock(char *ptr, int len)

C Specification (SPE and PPE only)

void receiveData(char *ptr, int len)

Description

The *receiveData* and *receiveDataBlock* subroutines receives a buffer of data of length *len* bytes on the current consumer connection and places the data into the array pointed to by *ptr*.

On non-simulated processors (i.e., local), two forms of receiveData are provided - blocking and non-blocking. The non-blocking form, *receiveData*, attempts to receive the specified number of bytes, returning the actual number of bytes received (up to the value *len*). The blocking form, *receiveDataBlock*, does not return until *len* bytes are received.

On simulated processors (i.e., PPE and SPE), receiveData is always blocked (but is named receiveData).

See Also

sendData on page 373

17.2.5 sendData

C Specification (Local only)

int sendData(char *ptr, int len)

void sendDataBlock(char *ptr, int len)

C Specification (SPE and PPE only)

void sendData(char *ptr, int len)

Description

The *sendData* and *sendDataBlock* subroutines sends a buffer of data pointed to by *ptr* of length *len* bytes on the current producer connection.

On non-simulated processors (i.e., local), two forms of sendData are provided - blocking and non-blocking. The non-blocking form, *sendData*, attempts to send the specified number of bytes, returning the actual number of bytes sent (up to the value *len*). The blocking form, *sendDataBlock*, does not return until *len* bytes are sent.

On simulated processors (i.e., PPE and SPE), sendData is always blocked (but is named sendData).

See Also

receiveData on page 372

18. 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 four sub-classes of synchronization primitives - atomic operations, mutexes, condition variables, and completion variables. The function closely match those found in current traditional operating systems.

This library is currently only supported on both the PPE and SPE.

Name(s)

libsync.a

Header File(s)

<libsync.h>

18.1 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.

18.1.1 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 377
atomic_inc on page 378
atomic_read on page 379
atomic_set on page 380
atomic_sub on page 381

18.1.2 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(int a, 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(int a, 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 the specified integer *a.*

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 376
atomic_inc on page 378
atomic_read on page 379
atomic_set on page 380
atomic_sub on page 381

18.1.3 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 376
atomic_dec on page 377
atomic_read on page 379
atomic_set on page 380
atomic_sub on page 381

18.1.4 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 376
atomic_dec on page 377
atomic_inc on page 378
atomic_set on page 380
atomic_sub on page 381

18.1.5 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 376
atomic_dec on page 377
atomic_inc on page 378
atomic_read on page 379
atomic_sub on page 381

18.1.6 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 the specified integer *a.*

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

Dependencies

See Also

atomic_add on page 376
atomic_dec on page 377
atomic_inc on page 378
atomic_read on page 379
atomic_set on page 380

18.2 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.

18.2.1 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 383
mutex_trylock on page 384
mutex_unlock on page 385

18.2.2 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 382
mutex_trylock on page 384
mutex_unlock on page 385

18.2.3 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 382
mutex_lock on page 383
mutex_unlock on page 385

18.2.4 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 382
mutex_lock on page 383
mutex_trylock on page 384

18.3 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.

18.3.1 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 387
cond_signal on page 388
cond_wait on page 389

18.3.2 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 386
cond_signal on page 388
cond_wait on page 389

18.3.3 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 386
cond_init on page 387
cond_wait on page 389

18.3.4 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 386
cond_init on page 387
cond_signal on page 388

18.4 Completion Variables

18.4.1 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 391
init_completion on page 392
wait_for_completion on page 393

18.4.2 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 390
init_completion on page 392
wait_for_completion on page 393

18.4.3 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 390
complete_all on page 391
wait_for_completion on page 393

18.4.4 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 390
complete_all on page 391
init_completion on page 392

19. 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>

19.1 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 19-1. NDC Packaging (128-Bit Floating-Point Vector)

msb lsb

x	y	z	w

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 libvertex.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 397

19.2 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

divide (floating point) on page 196

See Also

lerp_vec on page 407

19.3 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.

x	y	z	-

result.x = (v1.y * v2.z) - (v1.z * v2.y)
result.y = (v1.z * v2.x) - (v1.x * v2.z)
result.z = (v1.x * v2.y) - (v1.y * v2.x)
result.w = 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

19.4 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.

msb lsb

x	y	z	w

$$\text{dot\_product3(v1, v2)} = v1.x \times v2.x + v1.y \times v2.y + v1.z \times v2.z$$

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 quaterions.

$$\text{dot\_product4}(v1, v2) = v1.x \times v2.x + v1.y \times v2.y + v1.z \times v2.z + v1.w \times v1.w$$

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

19.5 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], constvector 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, ector 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],
                                    constvector 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],
                                    ector 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.

msb lsb

t	u	v	id

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. The eight hit records must ultimately be merged to determine the intersection for hte 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 398
dot_product on page 400
inverse on page 231
load_vec_float on page 409

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.

19.6 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*.

msb lsb

x	y	z	-

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 414
inv_sqrt on page 232

See Also

length_vec on page 406

19.7 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*.

msb lsb

x	y	z	-

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 414
sqrt on page 261

See Also

inv_length_vec on page 405

19.8 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 397

19.9 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:

msb			lsb
x	y	z	w

Dependencies

See Also

load_vec_int on page 410

19.10 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:

msb lsb

x	y	z	w

Dependencies

See Also

load_vec_float on page 409

19.11 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.

msb lsb

x	y	z	w

```
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

inv_sqrt on page 232

See Also

inv_length_vec on page 405

19.12 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

19.13 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

19.14 xform_norm3

C Specification

```
#include <xform_norm3.h>
inline vector float _xform_norm3(vector float in)

#include <xform_vec3_v.h>
inline void _xform_norm3(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(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 significant fields of a 128-

$$[\mathrm{out}] = [\mathrm{in}] \times [\mathrm{m}]$$

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_matricx4x4* 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 281
xform_vec on page 417

19.15 xform_vec

C Specification

```
#include <xform_vec3.h>
inline vector float _xform_vec3(vector float in)

#include <xform_vec4.h>
inline vector float _xform_vec4(vector float in)

#include <xform_vec3_v.h>
inline void _xform_vec3(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(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)

#include <libvector.h>
vector float xform_vec4(vector float in)

#include <libvector.h>
void xform_vec3(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(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*).

$$[out] = [in] \times [m]$$

The three components of the 3-D vector are stored in the 3 most significant fields of a 128-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_matricx4x4* 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.

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 281
xform_norm3 on page 415

20. Revision Log

The section documents the significant areas of change made to the libraries for each release of the SDK.

Table 1:

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.