Cell BE Monte Carlo Library API Reference Manual

Version 1.0 (beta)

Table of Contents

PREFACE ..... **iv**
About this publication iv
Intended audience iv
Conventions and terminology v
Typographical conventions v
Prerequisite and related information v
Chapter 1. Cell BE Monte Carlo Library Introduction **2**
Concepts 3
DMA 3
PPE 3
PPU 3
Random Number 3
Physical-Random Number 3
Pseudo-Random Number 3
Quasi-Random Number 3
RNG 3
SPE 3
SPU 4
Random Number Generators 4
Hardware-generated 4
Kirkpatrick-Stoll (KS) 4
Mersenne Twister (MT) 4
Sobol 4
Transformations 4
Box-Muller (BM) 4
Moro's Inversion (MI) 5
Polar Method (PO) 5
Selecting a Random Number Generator 5
Limitations 6
Chapter 2. Installing and Configuring Cell BE Monte Carlo Library **8**
Chapter 3. Programming **10**
Chapter 4. Programming for Cell BE **11**
Chapter 5. Random Number Generators **13**
Random Number Initialization 13
mc_rand_hw_init 14
mc_rand_ks_init 15
mc_rand_mt_init 16
mc_rand_sb_init 17
mc_rand_sb_seed 20
Random Number Generation 21
mc_rand_XX_u4 22
mc_rand_XX_array_u4 23
mc_rand_XX_0_to_1_d2 24
mc_rand_XX_0_to_1_array_d2 25

mc_rand_XX_minus1_to_1_d2 27
mc_rand_XX_minus1_to_1_array_d2 28
mc_rand_XX_0_to_1_f4 30
mc_rand_XX_0_to_1_array_f4 31
mc_rand_XX_minus1_to_1_f4 33
mc_rand_XX_minus1_to_1_array_f4 34
Chapter 6. Transformations 36
mc_transform_bm_f4 37
mc_transform_bm_array_f4 38
mc_transform_bm_d2 39
mc_transform_bm_array_d2 40
mc_transform_mi_f4 41
mc_transform_mi_array_f4 42
mc_transform_mi_d2 43
mc_transform_mi_array_d2 44
mc_transform_po_f4 45
mc_transform_po_array_f4 46
mc_transform_reject_po_array_f4 47
mc_transform_po_d2 49
mc_transform_po_array_d2 50
mc_transform_reject_po_array_d2 52
Appendix A. Examples 55
Hardware-Generated Example 56
Kirkpatrick-Stoll Example 58
Mersenne Twister Example 60
Sobol Example 61
Box-Muller Example 66
Moro's Inversion Example 68
Polar Method Example 70
Appendix B. Getting Help or Technical Assistance 73
Using the Documentation 73
Getting Help and Information from the World Wide Web 73
Contacting IBM Support 74
Appendix C. Accessibility 75
Appendix D. Notices 77
Code License and Disclaimer Information 77
Trademarks 77

PREFACE

About this publication

This document is the application programming interface (API) specification for the Cell Broadband™ (BE) Monte Carlo beta library provided in the IBM™ Cell Broadband Engine Software Development Kit (SDK). This library contains APIs to produce random numbers and perform distribution transformations on groups of numbers.

The library contains 4 random number generation (RNG) algorithms (hardware-generated, Kirkpatrick-Stoll, Mersenne Twister, and Sobol), 3 distribution transformations (Box-Muller, Moro's Inversion, and Polar Method), and two Monte Carlo simulation samples (calculations of pi and the volume of an n-dimensional sphere).

This document provides a detailed description of the APIs in their library and their use. Using this information, programmers on the Cell BE platform should be able to utilize the library to perform Monte Carlo simulations.

Specifically, the book covers the following sections:

- Chapter 1, "Cell BE Monte Carlo Library Introduction," on page 2 describes the various random number and transformation algorithms.
- Chapter 2, "Installing and Configuring Cell BE Monte Carlo Library," on page 8 addresses package installation.
- Chapter 3, "Programming," on page 10 covers basic programming setup for using the library.
- Chapter 4, "Programming for Cell BE," on page 11 documents platform unique restrictions for the library.
- Chapter 5, "Random Number Generators," on page 13 details the individual RNG APIs.
- Chapter 6, "Transformations," on page 36 describes the distribution transformation APIs.

Intended audience

This book provides details needed by software engineers and programmers. Specifically, it details random number generators and distribution transforms available with the Cell BE SDK on hardware platforms running the Cell Broadband Engine.

Conventions and terminology

Typographical conventions

The following table explains the typographical conventions used in this document.

Table 1. Typographical Conventions

Typeface	Indicates	Example
Bold	Lowercase commands, executable names, compiler options and directives.	If you specify **-O3**, the compiler assumes **-qhot=level=0**. To prevent all HOT optimizations with **-O3**, you must specify **-qnohot**.
Italics	Parameters or variables whose actual names or values are to be supplied by the user. Italics are also used to introduce new terms.	Make sure that you update the *size* parameter if you return more than the *size* requested.
`monospace`	Programming keywords and library functions, compiler built-in functions, examples of program code, command strings, or user-defined names.	If one or two cases of a `switch` statement are typically executed much more frequently than other cases, break out those cases by handling them separately before the `switch` statement.

Prerequisite and related information

The IBM Cell BE SDK 3.0 includes the Cell BE Monte Carlo library. The SDK should be installed prior to installing the library.

SDK installation documentation can be found in the *Software Development Kit 3.0 Installation Guide* available at the Cell Broadband Engine Resource Center developerWorks™ website, http://www-128.ibm.com/developerworks/power/cell.

Additional documentation pertaining to the SDK development environment can be found at this website—the *Cell Broadband Engine Programming Tutorial* and the *Cell Broadband Engine Programming Handbook*.

Part I. Overview

Chapter 1. Cell BE Monte Carlo Library Introduction

Random numbers generation and distribution transformation occur widely in many scientific and engineering applications for simulating random processes and statistical methods. Common applications for these numbers include lotteries and encryption key generation.

The Cell BE Monte Carlo Library provides two types of interfaces commonly used in Monte Carlo simulations – random number generators (RNGs) and distribution transformations.

The RNG algorithms implemented include:

1. Hardware-based
2. Kirkpatrick-Stoll
3. Mersenne Twister
4. Sobol

Additionally, the following transforms are also provided:

1. Box-Muller
2. Moro's Inversion
3. Polar Method

This SPU-only library generally provides interfaces in C and C++ to perform the operations—random number generation or distribution transformation—on either a single vector or an array of vectors.

Random numbers can be created of the following types:

- 32 bit integer (`unsigned int`)
- 32 bit single-precision floating point (`float`) with a range of (0 to 1]—from zero up to, but not including one.
- 32 bit single-precision floating point (`float`) with a range of [-1to 1] —from, but not including minus one up to, but not including one.
- 64 bit double-precision floating point (`double`) with a range of (0 to 1] —from zero up to, but not including one.
- 64 bit double-precision floating point (`double`) with a range of [-1 to 1] —from but not including minus 1 up to, but not including one.

Distribution transformations are provided for types both single- and double-precision floating point values (`float` and `double`).

Concepts

The following sections explain the main concepts and terms used in the Cell BE Monte Carlo library.

DMA

Direct Memory Access. A technique for using a special-purpose controller to generate the source and destination addresses for a memory or I/O transfer.

PPE

PowerPC™ Processor Element. The general-purpose processor in the Cell BE processor.

PPU

PowerPC Processor Unit. The part of the PPE that executes instructions from its main memory.

Random Number

A number obtained by chance.

Physical-Random Number

A random number obtained by sampling some physical object, such as a die.

Pseudo-Random Number

A number obtained by some defined arithmetic process, but is effectively a random number for the purpose for which it is required.

Quasi-Random Number

A random number also defined by an arithmetic process which compromises statistical randomness to obtain uniform distribution across the domain of potential values during its arithmetic sequence.

RNG

Random Number Generator. A program or library which returns random numbers.

SPE

Synergistic Processor Element. Extends the PowerPC 64 architecture by acting as cooperative offload processors (synergistic processors), with the direct memory access (DMA) and synchronization mechanisms to communicate with them (memory flow control), and with enhancements for real-time management. There are 8 SPEs on each Cell BE processor.

SPU

Synergistic Processor Unit. The part of an SPE that executes instructions from its local store (LS).

Random Number Generators

Hardware-generated

The hardware random number generator (HW RNG) samples hardware on Cell BE platform to generate its value. This physical RNG represents the closest interface to being truly random. No seed value is required and the resulting sequence does not have a predictable pattern.

Kirkpatrick-Stoll (KS)

The KS-RNG represents a quick and efficient implementation of a pseudo-random number generator. This RNG maintains a small set of working data and strives for linear independence among the generated numbers.

Mersenne Twister (MT)

The MT method for generating pseudo-random numbers also represents another fast and flexible approach to random-number generation.

The MT has a proven period of 2^(19937-1) with negligible serial correlation. The algorithm generates numbers using a twisted feedback shift register.

Sobol

The Sobol RNG represents the only quasi-random number generator in the library. Unlike pseudo-random number generators which strive for statistical randomness in its number, this RNG works for even distribution of numbers across the domain.

This implementation of the Sobol algorithm uses an application-provided initialization table and a large working data area to generate numbers extremely quickly.

Transformations

Box-Muller (BM)

The Box-Muller transformation converts a uniform distribution (0, 1] to a normal distribution with an expectant value of 0.

For the array interfaces, this transform returns two vectors for every input vector. For the vector interface, only a single vector is returned.

The general formula for the transformation of two input random numbers n_1 and n_2 are as follows:

$$t1 = \sqrt{-2 \ln (1 - n_1)} \cos(2\pi n_2$$

$$t_2 = \sqrt{-2 \ln (1 - n_1)} \sin(2\pi n_2)$$

Moro's Inversion (MI)

Like Box-Muller, the MI transform converts a uniform distribution (0,1] to a normal distribution `float` and `double` data types. This algorithm is the simplest of the distribution transformations, returning one transformed value for each input value.

Polar Method (PO)

The Polar Method is a derivative method of the Box-Muller transformation. This method also converts a uniform distribution (0,1] to a normal distribution of `float` and `double` data types. However, this method uses an accept-reject algorithm that generally produces fewer points, unless additional RNGs are generated.

The generalized formulas for the Polar Method uses two input random numbers n_1 and n_2 as follows:

$$q = (2n_1 - 1)^2 + (2n_2 - 1)^2$$

If q > 1 or q =0, the numbers are rejected and another pair is used. If not, the following formulas generate the pair of transformed values t_1 and t_2.

$$p = \sqrt{-2 \ln (q) / q}$$

$$t_1 = (2n_1 - 1) \bullet p$$

$$t_2 = (2n_2 - 1) \bullet p$$

In general, this method transforms data in a quicker fashion due to the substitution of one division for one multiplication and one trigonometric function.

Selecting a Random Number Generator

Applications requiring random numbers generally select the specific algorithm based upon their individual requirements and their knowledge of various algorithms.

When users are unfamiliar with the specific RNG algorithms, the following summary of the algorithms should be considered:

Table 2 Random Number Generator Comparisons

Algorithm	Location	Size	Speed	Randomness

libmisc rand()	PPU & SPU	Smallest	Moderate	Pseudo
Hardware	SPU	Small	Slowest	Physical
Kirkpatrick-Stoll	SPU	Moderate	Fast	Pseudo
Mersenne Twister	SPU	Moderate	Moderate	Pseudo
Sobol	SPU	Largest	Fastest	Quasi

Limitations

The hardware-generated random number generator has the following limitations on its values:

Table 3 Hardware-Generated Random Number Limitations

Function(s)	Number of Uniformly Distributed Values	Special Notes
`mc_rand_hw_u4,` `mc_rand_hw_array_u4`	2^{32}	
`mc_rand_hw_0_to_1_f4` `mc_rand_hw_0_to_1_array_f4`	2^{22}	The least significant bit (lsb) of the mantissa is always 0
`mc_rand_hw_minus1_to_1_f4` `mc_rand_hw_minus1_to_1_array_f4`	2^{21}	The 2 lsb's of the mantissa are always 0
`mc_rand_hw_0_to_1_d2` `mc_rand_hw_0_to_1_array_d2`	2^{51}	The lsb of the mantissa is always 0
`mc_rand_hw_minus1_to_1_d2` `mc_rand_hw_minus1_to_1_array_d2`	2^{50}	The 2 lsb's of the mantissa are always 0

Part II. Configuring Cell BE Monte Carlo Library

Chapter 2. Installing and Configuring Cell BE Monte Carlo Library

Installation and configuration of the Cell BE Monte Carlo library occurs after the SDK has been installed using the `cellsdk` script in the Cell BE SDK.

For details on installing the SDK, see the "Installing the SDK" section of the *Software Development Kit 3.0 Installation Guide* available at the Cell Broadband Engine Resource Center developerWorks website, http://www-128.ibm.com/developerworks/power/cell.

Once the SDK installation is complete, users wanting to develop with the library can install it directly with the following command:

```
yum install rpm_file_name
```

Customers developing their applications natively on Cell BE platforms should use an *rpm_file_name* of **libmc-rand-devel.3.0-1.ppc.rpm**. Customers developing on non-Cell BE platforms, should use *rpm_file_name* of **libmc-rand-cross-devel.3.0-1.ppc.rpm**.

Graphical installation can be accomplished by using the `cellsdk –gui install` command and then selecting the appropriate RPM.

Part III. Programming with Cell BE Monte Carlo Library

Chapter 3. Programming

To use the random number generators and transforms in the **libmc_rand** library, SPU programs should include the following statement:

```
#include <mc_rand.h>
```

The program's **Makefile** must also include the following statements to ensure linkage of the appropriate libraries:

```
INCLUDE = $(SDKPRINC)
LIBRARY += $(SDKPRLIB)

IMPORTS += -lmc_rand
```

Additionally, portions of the library have dependencies on the **simdmath** library. If the using program is not already including this library at link time, the following statement should be added to the **Makefile**:

```
IMPORTS += -lsimdmath
```

Programs running on the PPU and wishing to utilize the Sobol RNG algorithm on an SPU will also need the following include statement:

```
#include <mc_rand_sb.h>
```

No additional changes are needed to the **Makefile** for the PPU modules.

Chapter 4. Programming for Cell BE

The code provided in this design supports the same environments as the Cell BE SDK.

Although not explicitly prevented, all code except the Hardware RNG would function correctly on other Cell hardware such as the Sony™ PS3™. Detection of this environment is facilitated by an initialization routine for the Hardware RNG that returns a value indicating success or failure. The following table summarizes this limitation:

Table 4 Support Environments of the Hardware Random Number Generator

Secure CBE	Execution State	HW RNG	Comment
No	Isolated	-	Non-supported state
No	Non-isolated	Available	IBM Blade
Yes	Isolated	Available	Non-accessible state in current HW offerings
Yes	Non-isolated	Not Available	PS3

For more details, see the “Return Values” subsection of the mc_rand_hw_init API on page 14.

Part IV. Cell BE Monte Carlo Library API Reference

The following sections define the APIs found in the **libmc_rand** library.

Chapter 5. Random Number Generators

Two sets of APIs are generally provided with random number generators—initialization routines and random number generation routines. The following sections detail the interfaces provided in the Cell BE Monte Carlo Library.

Random Number Initialization

Each random number generator implementation has an initialization routine with its unique set of parameters. Before invoking any random number generation routines, the implementation-specific initialization routine should be called. Failure to do this will result in a poor variation of random numbers.

The following sections detail the random number initialization APIs.

mc_rand_hw_init

This interface initializes the hardware-generated random number generator.

Description

Verify and initialize the operating environment of the HW RNG. Indicate supported environment.

Syntax

int mc_rand_hw_init (void);

Parameters

None

Return Values

0	The environment supports the hardware-generated RNG.
< 0	The HW RNG is not supported in this environment.

Example

See Hardware-Generated Example on page 56.

Notes

The return value from the initialization routine must be checked. Execution of the RNG in an unsupported environment will result in random numbers of zero.

See Also

mc_rand_XX_u4 (page 22), mc_rand_XX_array_u4 (page 23), mc_rand_XX_0_to_1_d2 (page 24), mc_rand_XX_0_to_1_array_d2 (page 25), mc_rand_XX_minus1_to_1_d2 (page 27), mc_rand_XX_minus1_to_1_array_d2 (page 28), mc_rand_XX_0_to_1_f4 page 30), mc_rand_XX_0_to_1_array_f4 (page 31), mc_rand_XX_minus1_to_1_f4 (page 33), and mc_rand_XX_minus1_to_1_array_f4 (page 34) for related APIs.

Table 4 Support Environments of the Hardware Random Number Generator on page 11 for support environment details.

mc_rand_ks_init

This interface initializes the Kirkpatrick-Stoll random number generator.

Description

Initialize the operating environment of the KS RNG.

Syntax

void mc_rand_ks_init (unsigned int seed);

Parameters

`seed[IN]`	An initialization value for the RNG.

Return Values

None

Example

See Kirkpatrick-Stoll Example on page 58.

Notes

The initialization routine must be called prior to generating any random numbers. Failure to initialize the RNG will result in random number values of zeros.

See Also

mc_rand_XX_u4 (page 22), mc_rand_XX_array_u4 (page 23), mc_rand_XX_0_to_1_d2 (page 24), mc_rand_XX_0_to_1_array_d2 (page 25), mc_rand_XX_minus1_to_1_d2 (page 27), mc_rand_XX_minus1_to_1_array_d2 (page 28), mc_rand_XX_0_to_1_f4 page 30), mc_rand_XX_0_to_1_array_f4 (page 31), mc_rand_XX_minus1_to_1_f4 (page 33), and mc_rand_XX_minus1_to_1_array_f4 (page 34) for related APIs.

mc_rand_mt_init

This interface initializes the Mersenne Twister random number generator.

Description

Initialize the operating environment of the MT RNG using the seed provided.

Syntax

void mc_rand_mt_init (unsigned int seed);

Parameters

`seed[IN]`	An initialization value for the RNG.

Return Values

None

Example

See Mersenne Twister Example on page 60.

Notes

The initialization routine must be called prior to generating any random numbers. Failure to initialize the RNG will result in random number values of zeros.

See Also

mc_rand_XX_u4 (page 22), mc_rand_XX_array_u4 (page 23), mc_rand_XX_0_to_1_d2 (page 24), mc_rand_XX_0_to_1_array_d2 (page 25), mc_rand_XX_minus1_to_1_d2 (page 27), mc_rand_XX_minus1_to_1_array_d2 (page 28), mc_rand_XX_0_to_1_f4 page 30), mc_rand_XX_0_to_1_array_f4 (page 31), mc_rand_XX_minus1_to_1_f4 (page 33), and mc_rand_XX_minus1_to_1_array_f4 (page 34) for related APIs.

mc_rand_sb_init

This interface initializes the Sobol random number generator.

Description

Initialize the operating environment of the SB RNG using the seed provided.

Syntax

int mc_rand_sb_init (sobol_cntrlblk * p_control, unsigned int count_max_size, unsigned int dimension, vector unsigned char * p_memory, unsigned int size_of_memory);

Parameters

`p_control [IN]`	Specifies the control block that contains information about the direction table in main memory. The user must define this variable as data type `sobol_cntrlblck_t` as defined in the mc_rand_sb.h header file.
`count max size [IN]`	Defines the size of an array of vectors that can be filled with vectors of RNs. When using the **`mc_rand_sb_xx_array_yy( unsigned int count, vector <datatype> ** p_array)`** APIs, the *count* parameter must never exceed the value of *count_max_size*. When using the single vector versions (**`vector <datatype> mc_rand_sobol_xx (void)`**), set *count_max_size* = 1. However, larger *count_max_size* increases the performance of multiple calls to the single vector version of the RNG.
`dimension [IN]`	Defines the dimension of the random numbers. The maximum value of dimension depends on the initialization table. If a value above the maximum value is specified the initialization procedure will be aborted. The maximum value of dimension is defined by the direction table and is specified in the `sobol_cntrlblk_t` as variable *u32TableDimension*.
`p_memory [IN]`	Defines a pointer to the memory the RNG need to hold the lookup tables and to buffer RNs. The required amount of memory in bytes is equal to 640 times the dimension size. For needed amount of memory refer to the “Notes” section below.
`size of memory [IN]`	Defines the size of the memory to which *p_memory* points. The initialization procedure verifies if the amount of memory is sufficient and aborts if the memory is too

	small.

Return Values

`0`	Initialization successful. No error.
`2`	Error. Requested *dimension* parameter value is greater than the maximum dimension of the look-up table in the *p_control* structure.
`4`	Error. Requested *dimension* parameter value is less than 1.
`8`	Error. Look-up table supports random numbers with less than 1 bit only.
`16`	Error. Look-up table supports random numbers with more than 32 bits
`512`	Error. Passed memory area pointed to by *p_memory* is too small.

Example

See Sobol Example on page 61.

Notes

The `sobol_cntrlblk_t` is a key structure in the function of the Sobol RNG. The definition of this structure can be found in /opt/cell/sdk/prototype/usr/spu/include/mc_rand_sb.h for SPU programs and /opt/cell/sdk/prototype/usr/include/mc_rand_sb.h for PPU programs.

An example of how to initialize this structure can be found in /opt/cell/sdk/prototype/src/examples/monte-carlo/sphere/sobol_init_30_40.h.

Additionally, key defines are provided in the mc_rand_sb.h file as default values:

- `SOBOL_RUNS`
  Defines the maximal number of elements an array of random number vectors can have. Defaults to 112.
- `SOBOL_DIMENSION`
  Defines the dimension of the created RNs. Defaults to 40.
- `SOBOL_VECTOR_ARRAY_SIZE`
  Calculates the size of the vector array needed as the *p_memory* parameter.

It is strongly recommended that users to change the default for `SOBOL_RUNS` and `SOBOL_DIMENSION` by using the following code:

```
#undef SOBOL_RUNS
#define SOBOL_RUNS xxx

#undef SOBOL_DIMENSION
#define SOBOL_DIMENSION yyy
```

Where xxx and yyy are appropriate numbers.

Instead of using the `SOBOL_VECTOR_ARRAY_SIZE` literal, users can manually calculate the needed amount of memory by keeping in mind that the following formula:

<memory needed> = ((<number of runs> +76) * <dimensions> +8) *16.

See Also

mc_rand_XX_u4 (page 22), mc_rand_XX_array_u4 (page 23), mc_rand_XX_0_to_1_d2 (page 24), mc_rand_XX_0_to_1_array_d2 (page 25), mc_rand_XX_minus1_to_1_d2 (page 27), mc_rand_XX_minus1_to_1_array_d2 (page 28), mc_rand_XX_0_to_1_f4 page 30), mc_rand_XX_0_to_1_array_f4 (page 31), mc_rand_XX_minus1_to_1_f4 (page 33), and mc_rand_XX_minus1_to_1_array_f4 (page 34) for related APIs.

mc_rand_sb_seed

This interface seeds the Sobol random number generator.

Description

Seed the Sobol RNG with the specified value. This seed value represents the index into the Sobol sequence of random numbers.

Syntax

void mc_rand_sb_seed (unsigned int seed);

Parameters

`seed [IN]`	Index into the RNG sequence. This number should be evenly divisible by 4. If a non-multiple is provided, the value of `seed` will be truncated to the previous multiple of 4.

Return Values

None

Example

See Sobol Example on page 61.

Notes

The Sobol RNG defaults to a seed of 0 after invocation of `mc_rand_sb_init()`.

See Also

mc_rand_sb_init (page 17) for related API.

Random Number Generation

Random number generation interfaces provided with the Cell BE Monte Carlo library have consistent APIs across all implementation for a common data type. In general, these interfaces can be divided into routines to return single vectors or an array of vectors.

The following sections define the random number generation APIs.

mc_rand_XX_u4

This interface is generic across all RNG implementations. The value of "**XX**" will be "**hw**" for Hardware, "**ks**" for Kirkpatrick-Stoll, "**mt**" for Mersenne Twister, and "**sb**" for Sobol.

Description

Return one random number vector of type unsigned integers with each function call.

Syntax

vector unsigned int mc_rand_hw_u4 (void);

vector unsigned int mc_rand_ks_u4 (void);

vector unsigned int mc_rand_mt_u4 (void);

vector unsigned int mc_rand_sb_u4 (void);

Parameters

None

Return Values

`Random numbers`	Random number vector of 4 unsigned integers

Example

See Hardware-Generated Example on page 56, Kirkpatrick-Stoll Example on page 58, Mersenne Twister Example on page 60, and Sobol Example on page 61.

See Also

mc_rand_XX_array_u4(page 23) for related APIs.

mc_rand_hw_init (page 14), mc_rand_ks_init (page 15), mc_rand_mt_init (page 16), or mc_rand_sb_init (page 17) for appropriate initialization API.

mc_rand_XX_array_u4

This interface is generic across most RNG implementations with a slight variation for Sobol. The value of "**XX**" will be "**hw**" for Hardware, "**ks**" for Kirkpatrick-Stoll, "**mt**" for Mersenne Twister, and "**sb**" for Sobol.

Description

Return an array of random number vectors of type unsigned integers with each function call.

For the hardware-generated, Kirkpatrick-Stoll, and Mersenne Twister interfaces, the random numbers are returned into the array specified by the user. For Sobol, the pointer with the random numbers is returned by the random number generator.

Syntax

void mc_rand_hw_array_u4 (unsigned int count, vector unsigned int *array);

void mc_rand_ks_array_u4 (int count, vector unsigned int *array);

void mc_rand_mt_array_u4 (int count, vector unsigned int *array);

vector unsigned int *mc_rand_sb_array_u4 (int count);

Parameters

`count [IN]`	The number of random number vectors to return.
`array [IN/OUT]`	The pointer to the memory location where the random numbers should be generated. This parameter applies only to the HW, KS, and MT RNGs.

Return Values

`Random numbers`	An array of random number vectors with 4 unsigned integers. For the HW, KS, and MT RNGs, these numbers are created and stored in the memory location referenced by the *array* pointer. For the SB RNG, a pointer to these values is returned .

Example

See Hardware-Generated Example on page 56, Kirkpatrick-Stoll Example on page 58, Mersenne Twister Example on page 60, and Sobol Example on page 61.

See Also

mc_rand_XX_u4(page 22) for related APIs.

mc_rand_hw_init (page 14), mc_rand_ks_init (page 15), mc_rand_mt_init (page 16), or mc_rand_sb_init (page 17) for appropriate initialization API.

mc_rand_XX_0_to_1_d2

This interface is generic across all RNG implementations. The value of "**XX**" will be "**hw**" for Hardware, "**ks**" for Kirkpatrick-Stoll, "**mt**" for Mersenne Twister, and "**sb**" for Sobol.

Description

Return one random number vector of type `double` with each function call. These double precision floating-point random numbers range from 0 up to, but not including, 1.

Syntax

vector double mc_rand_hw_0_to_1_d2 (void);

vector double mc_rand_ks_0_to_1_d2 (void);

vector double mc_rand_mt_0_to_1_d2 (void);

vector double mc_rand_sb_0_to_1_d2 (void);

Parameters

None

Return Values

`Random numbers`	Random number vector of 2 double precision floating point numbers.

Example

See Hardware-Generated Example on page 56, Kirkpatrick-Stoll Example on page 58, Mersenne Twister Example on page 60, and Sobol Example on page 61.

See Also

mc_rand_XX_0_to_1_array_d2 (page 25) for a related API.

mc_rand_hw_init (page 14), mc_rand_ks_init (page 15), mc_rand_mt_init (page 16), or mc_rand_sb_init (page 17) for appropriate initialization API.

mc_rand_XX_0_to_1_array_d2

This interface is generic across most RNG implementations with a slight variation for Sobol. The value of "**xx**" will be "**hw**" for Hardware, "**ks**" for Kirkpatrick-Stoll, "**mt**" for Mersenne Twister, and "**sb**" for Sobol.

Description

Return an array of random number vectors of type `double` with each function call. These double precision floating-point random numbers range from 0 up to, but not including, 1.

For the hardware-generated, Kirkpatrick-Stoll, and Mersenne Twister interfaces, the random numbers are returned into the array specified by the user. For Sobol, the pointer with the random numbers is returned by the random number generator.

Syntax

void mc_rand_hw_0_to_1_array_d2 (unsigned int count, vector double *array);

void mc_rand_ks_0_to_1_array_d2 (unsigned int count, vector double *array);

void mc_rand_mt_0_to_1_array_d2 (unsigned int count, vector double *array);

vector double *mc_rand_sb_0_to_1_array_d2 (unsigned int count);

Parameters

`count [IN]`	The number of random number vectors to return.
`array [IN/OUT]`	The pointer to the memory location where the random numbers should be generated. This parameter applies only to the HW, KS, and MT RNGs.

Return Values

`Random numbers` An array of random number vectors each with 2 double precision floating point numbers. For the HW, KS, and MT RNGs, these numbers are created and stored in the memory location referenced by the *array* pointer. For the SB RNG, a pointer to these values is returned.

Example

See Hardware-Generated Example on page 56, Kirkpatrick-Stoll Example on page 58, Mersenne Twister Example on page 60, and Sobol Example on page 61.

See Also

mc_rand_XX_0_to_1_d2 (page 24) for a related API.

mc_rand_hw_init (page 14), mc_rand_ks_init (page 15), mc_rand_mt_init (page 16), or mc_rand_sb_init (page 17) for appropriate initialization API.

mc_rand_XX_minus1_to_1_d2

This interface is generic across all RNG implementations. The value of "**XX**" will be "**hw**" for Hardware, "**ks**" for Kirkpatrick-Stoll, "**mt**" for Mersenne Twister, and "**sb**" for Sobol.

Description

Return one random number vector of type `double` with each function call. These double precision floating-point random numbers range from, but not including -1 up to, but not including, 1.

Syntax

vector double mc_rand_hw_minus1_to_1_d2 (void);

vector double mc_rand_ks_minus1_to_1_d2 (void);

vector double mc_rand_mt_minus1_to_1_d2 (void);

vector double mc_rand_sb_minus1_to_1_d2 (void);

Parameters

None

Return Values

`Random numbers`	Random number vector of 2 double precision floating point numbers.

Example

See Hardware-Generated Example on page 56, Kirkpatrick-Stoll Example on page 58, Mersenne Twister Example on page 60, and Sobol Example on page 61.

See Also

mc_rand_XX_minus1_to_1_array_d2 (page 28) for a related API.

mc_rand_hw_init (page 14), mc_rand_ks_init (page 15), mc_rand_mt_init (page 16), or mc_rand_sb_init (page 17) for appropriate initialization API.

mc_rand_XX_minus1_to_1_array_d2

This interface is generic across most RNG implementations with a slight variation for Sobol. The value of "**XX**" will be "**hw**" for Hardware, "**ks**" for Kirkpatrick-Stoll, "**mt**" for Mersenne Twister, and "**sb**" for Sobol.

Description

Return an array of random number vectors of type `double` with each function call. These double precision floating-point random numbers range from, but not including -1 up to, but not including, 1.

For the hardware-generated, Kirkpatrick-Stoll, and Mersenne Twister interfaces, the random numbers are returned into the array specified by the user. For Sobol, the pointer with the random numbers is returned by the random number generator.

Syntax

void mc_rand_hw_minus1_to_1_array_d2 (unsigned int count, vector double *array);

void mc_rand_ks_minus1_to_1_array_d2 (unsigned int count, vector double *array);

void mc_rand_mt_minus1_to_1_array_d2 (unsigned int count, vector double *array);

vector double *mc_rand_sb_minus1_to_1_array_d2 (unsigned int count);

Parameters

`count [IN]`	The number of random number vectors to return.
`array [IN/OUT]`	The pointer to the memory location where the random numbers should be generated. This parameter applies only to the HW, KS, and MT RNGs.

Return Values

`Random numbers`	An array of random number vectors each with 2 double precision floating point numbers. For the HW, KS, and MT RNGs, these numbers are created and stored in the memory location referenced by the *array* pointer. For the SB RNG, a pointer to these values is returned.

Example

See Hardware-Generated Example on page 56, Kirkpatrick-Stoll Example on page 58, Mersenne Twister Example on page 60, and Sobol Example on page 61.

See Also

mc_rand_XX_minus1_to_1_d2 (page 27) for a related API.

mc_rand_hw_init (page 14), mc_rand_ks_init (page 15), mc_rand_mt_init (page 16), or mc_rand_sb_init (page 17) for appropriate initialization API.

mc_rand_XX_0_to_1_f4

This interface is generic across all RNG implementations. The value of "**XX**" will be "**hw**" for Hardware, "**ks**" for Kirkpatrick-Stoll, "**mt**" for Mersenne Twister, and "**sb**" for Sobol.

Description

Return one random number vector of type `float` with each function call. These single precision floating-point random numbers range from 0 up to, but not including, 1.

Syntax

vector float mc_rand_hw_0_to_1_f4 (void);

vector float mc_rand_ks_0_to_1_f4 (void);

vector float mc_rand_mt_0_to_1_f4 (void);

vector float mc_rand_sb_0_to_1_f4 (void);

Parameters

None

Return Values

`Random numbers`	Random number vector of 4 single precision floating point numbers.

Example

See Hardware-Generated Example on page 56, Kirkpatrick-Stoll Example on page 58, Mersenne Twister Example on page 60, and Sobol Example on page 61.

See Also

mc_rand_XX_0_to_1_array_f4 (page31) for a related API.

mc_rand_hw_init (page 14), mc_rand_ks_init (page 15), mc_rand_mt_init (page 16), or mc_rand_sb_init (page 17) for appropriate initialization API.

mc_rand_XX_0_to_1_array_f4

This interface is generic across most RNG implementations with a slight variation for Sobol. The value of "**XX**" will be "**hw**" for Hardware, "**ks**" for Kirkpatrick-Stoll, "**mt**" for Mersenne Twister, and "**sb**" for Sobol.

Description

Return an array of random number vectors of type `float` with each function call. These single precision floating-point random numbers range from 0 up to, but not including, 1.

For the hardware-generated, Kirkpatrick-Stoll, and Mersenne Twister interfaces, the random numbers are returned into the array specified by the user. For Sobol, the pointer with the random numbers is returned by the random number generator.

Syntax

void mc_rand_hw_0_to_1_array_f4 (unsigned int count, vector float *array);

void mc_rand_ks_0_to_1_array_f4 (unsigned int count, vector float *array);

void mc_rand_mt_0_to_1_array_f4 (unsigned int count, vector float *array);

vector float *mc_rand_sb_0_to_1_array_f4 (unsigned int count);

Parameters

`count [IN]`	The number of random number vectors to return.
`array [IN/OUT]`	The pointer to the memory location where the random numbers should be generated. This parameter applies only to the HW, KS, and MT RNGs.

Return Values

`Random numbers`	An array of random number vectors each with 4 single precision floating point numbers. For the HW, KS, and MT RNGs, these numbers are created and stored in the memory location referenced by the *array* pointer. For the SB RNG, a pointer to these values is returned.

Example

See Hardware-Generated Example on page 56, Kirkpatrick-Stoll Example on page 58, Mersenne Twister Example on page 60, and Sobol Example on page 61.

See Also

mc_rand_XX_0_to_1_f4 (page 30) for a related API.

mc_rand_hw_init (page 14), mc_rand_ks_init (page 15), mc_rand_mt_init (page 16), or mc_rand_sb_init (page 17) for appropriate initialization API.

mc_rand_XX_minus1_to_1_f4

This interface is generic across all RNG implementations. The value of "**XX**" will be "**hw**" for Hardware, "**ks**" for Kirkpatrick-Stoll, "**mt**" for Mersenne Twister, and "**sb**" for Sobol.

Description

Return one random number vector of type `float` with each function call. These single precision floating-point random numbers range from, but not including -1 up to, but not including, 1.

Syntax

vector float mc_rand_hw_minus1_to_1_f4 (void);

vector float mc_rand_ks_minus1_to_1_f4 (void);

vector float mc_rand_mt_minus1_to_1_f4 (void);

vector float mc_rand_sb_minus1_to_1_f4 (void);

Parameters

None

Return Values

`Random numbers`	Random number vector of 4 single precision floating point numbers.

Example

See Hardware-Generated Example on page 56, Kirkpatrick-Stoll Example on page 58, Mersenne Twister Example on page 60, and Sobol Example on page 61.

See Also

mc_rand_XX_minus1_to_1_array_f4 (page 34) for a related API.

mc_rand_hw_init (page 14), mc_rand_ks_init (page 15), mc_rand_mt_init (page 16), or mc_rand_sb_init (page 17) for appropriate initialization API.

mc_rand_XX_minus1_to_1_array_f4

This interface is generic across most RNG implementations with a slight variation for Sobol. The value of "**XX**" will be "**hw**" for Hardware, "**ks**" for Kirkpatrick-Stoll, "**mt**" for Mersenne Twister, and "**sb**" for Sobol.

Description

Return an array of random number vectors of type `float` with each function call. These single precision floating-point random numbers range from, but not including -1 up to, but not including, 1.

For the hardware-generated, Kirkpatrick-Stoll, and Mersenne Twister interfaces, the random numbers are returned into the array specified by the user. For Sobol, the pointer with the random numbers is returned by the random number generator.

Syntax

void mc_rand_hw_minus1_to_1_array_f4 (unsigned int count, vector float *array);

void mc_rand_ks_minus1_to_1_array_f4 (unsigned int count, vector float *array);

void mc_rand_mt_minus1_to_1_array_f4 (unsigned int count, vector float *array);

vector float *mc_rand_sb_minus1_to_1_array_f4 (unsigned int count);

Parameters

`count [IN]`	The number of random number vectors to return.
`array [IN/OUT]`	The pointer to the memory location where the random numbers should be generated. This parameter applies only to the HW, KS, and MT RNGs.

Return Values

`Random numbers`	An array of random number vectors each with 4 single precision floating point numbers. For the HW, KS, and MT RNGs, these numbers are created and stored in the memory location referenced by the *array* pointer. For the SB RNG, a pointer to these values is returned.

Example

See Hardware-Generated Example on page 56, Kirkpatrick-Stoll Example on page 58, Mersenne Twister Example on page 60, and Sobol Example on page 61.

See Also

mc_rand_XX_minus1_to_1_f4 (page 33) for a related API.

mc_rand_hw_init (page 14), mc_rand_ks_init (page 15), mc_rand_mt_init (page 16), or mc_rand_sb_init (page 17) for appropriate initialization API.

Chapter 6. Transformations

The Cell BE Monte Carlo library transformation routines have similar interfaces with distinct parameters. In general, these routines take one or more vectors of single or double precision floating-point numbers and transform them from a fixed distribution to a normal distribution.

The Box-Muller and Polar Methods use accept-reject algorithms to transform *m* inputs into *n* output numbers, where $m > n$. Moro's inversion, however, is a simple algorithm that transforms *m* inputs into *n* output numbers, where $m = n$.

All of the transformation routines may be invoked directly without an initialization routine.

mc_transform_bm_f4

This interface applies a Box-Muller transformation to a vector of single-precision floating point numbers from an uniform distribution to a normal distribution..

Description

Transform a vector of float-typed random numbers from a uniform distribution into a normal distribution and return the new vector.

Syntax

vector float mc_transform_bm_f4 (vector float src);

Parameters

`src [IN]`	An input vector of 4 single-precision floating point numbers, evenly distributed from 0 up to, but not including 1.

Return Values

`New vector`	A vector of 4 floating point numbers with a normal distribution.

Example

See Box-Muller Example on page 66.

See Also

mc_transform_bm_array_f4 (page 38) for a related API.

mc_transform_bm_array_f4

This interface applies a Box-Muller transformation to an array of single-precision floating point vectors from an uniform distribution to a normal distribution..

Description

Transform an array of vector float types from a uniform distribution into a normal distribution and return the new array.

Syntax

void mc_transform_bm_array_f4 (unsigned int count, vector float *s_rand_array, vector float *t_rand_array);

Parameters

`count [IN]`	The number of vectors contained in the input arrays *s_rand_array*.
`s_rand_array [IN]`	An input array of vectors. Each vector will contain four single-precision floating point numbers ranging from 0 up to, but not including, 1. The number of vectors pointed to by the array is contained in the *count* parameter.
`t_rand_array [IN/OUT]`	The memory location for the output array of vectors. Enough memory should be allocated by the invoking program to hold **`2*`***count* number of vectors.

Return Values

`New distribution in t_rand_array`	The new data distribution will be written to the memory location specified by the *t_rand_array* parameter. This array will contain **`2*`***count* number of single-precision floating point vectors with values having a normal distribution.

Example

See Box-Muller Example on page 66.

Notes

The same memory location may be used for the *s_rand_array* and the *t_rand_array* parameters.

See Also

mc_transform_bm_f4 (page 37) for a related API.

mc_transform_bm_d2

This interface applies a Box-Muller transformation to a vector of double-precision floating point numbers from an uniform distribution to a normal distribution.

Description

Transform a vector of double-typed random numbers from a uniform distribution into a normal distribution and return the new vector.

Syntax

vector double mc_transform_bm_d2 (vector double);

Parameters

`src [IN]`	An input vector of 2 double-precision floating point numbers, evenly distributed from 0 up to, but not including 1.

Return Values

`New vector`	A vector of 2 floating point numbers with a normal distribution.

Example

See Box-Muller Example on page 66.

See Also

mc_transform_bm_array_d2 (page 40) for a related API.

mc_transform_bm_array_d2

This interface applies Moro's Inversion to transform an array of double-precision floating point vectors from an uniform distribution to a normal distribution..

Description

Transform an array of vector double types from a uniform distribution into a normal distribution and return the new array.

Syntax

void mc_transform_bm_array_d2 (unsigned int count, vector double *s_rand_array, vector double *t_rand_array);

Parameters

`count [IN]`	The number of vectors contained in the input array *s_rand_array*.
`s_rand_array [IN]`	An input array of vectors. Each vector will contain 2 double-precision floating point numbers ranging from 0 up to, but not including, 1. The number of vectors pointed to by the array is contained in the *count* parameter.
`t_rand_array [IN/OUT]`	The memory location for the output array of vectors. Enough memory should be allocated by the invoking program to hold **`2*`***count* number of vectors.

Return Values

`New distribution in t_rand_array`	The new data distribution will be written to the memory location specified by the *t_rand_array* parameter. This array will contain **`2*`***count* number of double-precision floating point vectors with values having a normal distribution.

Example

See Box-Muller Example on page 66.

Notes

The same memory location may be used for the *s_rand_array* and the *t_rand_array* parameters.

See Also

mc_transform_bm_d2 (page 39) for a related API.

mc_transform_mi_f4

This interface applies Moro's Inversion to transform a vector of single-precision floating point numbers from an uniform distribution to a normal distribution..

Description

Transform a vector of float-typed random numbers from a uniform distribution into a normal distribution and return the new vector.

Syntax

vector float mc_transform_mi_f4 (vector float src);

Parameters

`src [IN]`	The input vector of 4 single-precision floating point numbers, evenly distributed from 0 up to, but not including 1.

Return Values

`New vector`	A vector of 4 floating point numbers with a normal distribution.

Example

See Moro's Inversion Example on page 68.

See Also

mc_transform_mi_array_f4 (page 42) for a related API.

mc_transform_mi_array_f4

This interface applies Moro's Inversion to transform an array of single-precision floating point vectors from an uniform distribution to a normal distribution..

Description

Transform an array of vector float types from a uniform distribution into a normal distribution and return the new array.

Syntax

void mc_transform_mi_array_f4 (unsigned int count, vector float *s_rand_array, vector float *t_rand_array);

Parameters

`count [IN]`	The number of vectors contained in the input array *s_rand_array*.
`s_rand_array [IN]`	The input array of vectors. Each vector will contain four single-precision floating point numbers ranging from 0 up to, but not including, 1. The number of vectors pointed to by the array is contained in the *count* parameter.
`t_rand_array [IN/OUT]`	The memory location for the output array of vectors. Enough memory should be allocated by the invoking program to hold *count* number of vectors.

Return Values

`New distribution in t_rand_array`	The new data distribution will be written to the memory location specified by the *t_rand_array* parameter. This array will contain *count* number of single-precision floating point vectors with values having a normal distribution.

Example

See Moro's Inversion Example on page 68.

Notes

The same memory location may be used for the *s_rand_array* and the *t_rand_array* parameters.

See Also

mc_transform_mi_f4 (page 41) for a related API.

mc_transform_mi_d2

This interface applies Moro's Inversion to transform a vector of double-precision floating point numbers from an uniform distribution to a normal distribution..

Description

Transform a vector of double-typed random numbers from a uniform distribution into a normal distribution and return the new vector.

Syntax

vector double mc_transform_mi_d2 (vector double src);

Parameters

`src [IN]`	The input vector of 2 double-precision floating point numbers, evenly distributed from 0 up to, but not including 1.

Return Values

`New vector`	A vector of 2 floating point numbers with a normal distribution.

Example

See Moro's Inversion Example on page 68.

See Also

mc_transform_mi_array_d2 (page 44) for a related API.

mc_transform_mi_array_d2

This interface applies Moro's Inversion to transform an array of double-precision floating point vectors from an uniform distribution to a normal distribution..

Description

Transform an array of vector double types from a uniform distribution into a normal distribution and return the new array.

Syntax

void mc_transform_mi_array_d2 (unsigned int count, vector double *s_rand_array, vector double *t_rand_array);

Parameters

`count [IN]`	The number of vectors contained in the input array *s_rand_array*.
`s_rand_array [IN]`	The input array of vectors. Each vector will contain 2 double-precision floating point numbers ranging from 0 up to, but not including, 1. The number of vectors pointed to by the array is contained in the *count* parameter.
`t_rand_array [IN/OUT]`	The memory location for the output array of vectors. Enough memory should be allocated by the invoking program to hold *count* number of vectors.

Return Values

`New distribution in t_rand_array`	The new data distribution will be written to the memory location specified by the *t_rand_array* parameter. This array will contain *count* number of double-precision floating point vectors with values having a normal distribution.

Example

See Moro's Inversion Example on page 68.

Notes

The same memory location may be used for the *s_rand_array* and the *t_rand_array* parameters.

See Also

mc_transform_mi_d2 (page 43) for a related API.

mc_transform_po_f4

This interface applies a Polar-Method transform to the results from a specified random number generator to create and return vector with a normal distribution of single-precision floating point numbers.

Description

Transform the vector results from an RNG into a normal distribution and return the new vector.

Syntax

vector float mc_transform_po_f4 (vector float (*p_rng) (void));

Parameters

`p_rng [IN]`	A function pointer to a RNG that returns a vector of type `float` from 0 up to, but not including 1.

Return Values

`New vector`	A vector of 4 single-precision floating point numbers with a normal distribution.

Example

See Polar Method Example on page 70.

See Also

mc_transform_po_array_f4 (page 46) and mc_transform_reject_po_array_f4 (page 47) for related APIs.

mc_transform_po_array_f4

This interface applies a Polar-Method transform to the source numbers, using specified random number generator as needed, to create and return an array of vectors with a normal distribution of single-precision floating point numbers.

Description

Transform an array of vector float types from a uniform distribution into a normal distribution and return the new array.

Syntax

void mc_transform_po_array_f4 (unsigned int count, vector float *s_rand_array, vector float *t_rand_array, vector float (*p_rng) (void));

Parameters

`count [IN]`	The number of vectors contained in the input arrays *s_rand_array*.
`s_rand_array [IN]`	An input array of vectors. Each vector will contain four single-precision floating point numbers ranging from 0 up to, but not including, 1. The number of vectors pointed to by the array is contained in the *count* parameter.
`t_rand_array [IN/OUT]`	The memory location for the output array of vectors. Enough memory should be allocated by the invoking program to hold *count* number of vectors.
`p_rng [IN]`	A function pointer to a RNG that returns a vector of type `float` from 0 up to, but not including 1.

Return Values

`New distribution in t_rand_array`	The new data distribution will be written to the memory location specified by the *t_rand_array* parameter. This array will contain *count* number of single-precision floating point vectors with values having a normal distribution.

Example

See Polar Method Example on page 70.

Notes

The same memory location may be used for the *s_rand_array* and the *t_rand_array* parameters.

See Also

mc_transform_po_f4 (page 45) and mc_transform_reject_po_array_f4 (page 47) for related APIs.

mc_transform_reject_po_array_f4

This interface applies a Polar-Method transform to the source numbers to create and return an array of vectors with equal or fewer elements than the input vector and a normal distribution of single-precision floating point numbers.

Description

Transform an array of vector float types from a uniform distribution into a normal distribution and return the new array.

Syntax

unsigned int mc_transform_reject_po_array_f4 (unsigned int count, vector float *s_rand_array, vector float *t_rand_array);

Parameters

`count [IN]`	The number of vectors contained in the input arrays *s_rand_array*.
`s_rand_array [IN]`	An input array of vectors. Each vector will contain four single-precision floating point numbers ranging from 0 up to, but not including, 1. The number of vectors pointed to by the array is contained in the *count* parameter.
`t_rand_array [IN/OUT]`	The memory location for the output array of vectors. Enough memory should be allocated by the invoking program to hold *count* number of vectors. The actual number of elements in the array will returned on the call.

Return Values

`New distribution in t_rand_array`	The new data distribution will be written to the memory location specified by the *t_rand_array* parameter. This array will contain *count* number of single-precision floating point vectors with values having a normal distribution.
`Count of elements in t_rand_array`	The number of elements written into the *t_rand_array*. This value will be less-than or equal-to the value of the *count* input parameter.

Example

See Polar Method Example on page 70.

Notes

The same memory location may be used for the *s_rand_array* and the *t_rand_array* parameters.

See Also

mc_transform_po_f4 (page 45) and mc_transform_po_array_f4 (page 46 for related APIs.

mc_transform_po_d2

This interface applies a Polar-Method transform to the results from a specified random number generator to create and return vector with a normal distribution of double-precision floating point numbers.

Description

Transform the vector results from an RNG into a normal distribution and return the new vector.

Syntax

vector double mc_transform_po_d2 (vector double (*p_rng) (void));

Parameters

`p_rng [IN]`	A function pointer to a RNG that returns a vector of type `double` from 0 up to, but not including 1.

Return Values

`New vector`	A vector of 2 double-precision floating point numbers with a normal distribution.

Example

See Polar Method Example on page 70.

See Also

mc_transform_po_array_d2 (page 50) and mc_transform_reject_po_array_d2 (page 52) for related APIs.

mc_transform_po_array_d2

This interface applies a Polar-Method transform to the source numbers, using specified random number generator as needed, to create and return an array of vectors with a normal distribution of double-precision floating point numbers.

Description

Transform an array of vector double types from a uniform distribution into a normal distribution and return the new array.

Syntax

void mc_transform_po_array_d2 (unsigned int count, vector double *s_rand_array, vector double *t_rand_array, vector double (*p_rng) (void));

Parameters

`count [IN]`	The number of vectors contained in the input arrays *s_rand_array*.
`s_rand_array [IN]`	An input array of vectors. Each vector will contain 2 double-precision floating point numbers ranging from 0 up to, but not including, 1. The number of vectors pointed to by the array is contained in the *count* parameter.
`t_rand_array [IN/OUT]`	The memory location for the output array of vectors. Enough memory should be allocated by the invoking program to hold *count* number of vectors.
`p_rng [IN]`	A function pointer to a RNG that returns a vector of type `double` from 0 up to, but not including 1.

Return Values

`New distribution in t_rand_array`	The new data distribution will be written to the memory location specified by the *t_rand_array* parameter. This array will contain *count* number of double-precision floating point vectors with values having a normal distribution.

Example

See Polar Method Example on page 70.

Notes

The same memory location may be used for the *s_rand_array* and the *t_rand_array* parameters.

See Also

mc_transform_po_d2 (page 49) mc_transform_reject_po_array_d2 (page 52) for related APIs.

mc_transform_reject_po_array_d2

This interface applies a Polar-Method transform to the source numbers to create and return an array of vectors with equal or fewer elements than the input vector and a normal distribution of double-precision floating point numbers.

Description

Transform an array of vector float types from a uniform distribution into a normal distribution and return the new array.

Syntax

unsigned int mc_transform_reject_po_array_d2 (unsigned int count, vector double *s_rand_array, vector double *t_rand_array);

Parameters

`count [IN]`	The number of vectors contained in the input arrays *s_rand_array*.
`s_rand_array [IN]`	An input array of vectors. Each vector will contain two double-precision floating point numbers ranging from 0 up to, but not including, 1. The number of vectors pointed to by the array is contained in the *count* parameter.
`t_rand_array [IN/OUT]`	The memory location for the output array of vectors. Enough memory should be allocated by the invoking program to hold *count* number of vectors. The actual number of elements in the array will returned on the call.

Return Values

`New distribution in t_rand_array`	The new data distribution will be written to the memory location specified by the *t_rand_array* parameter. This array will contain *count* number of double-precision floating point vectors with values having a normal distribution.
`Count of elements in t_rand_array`	The number of elements written into the *t_rand_array*. This value will be less-than or equal-to the value of the *count* input parameter.

Example

See Polar Method Example on page 70.

Notes

The same memory location may be used for the *s_rand_array* and the *t_rand_array* parameters.

See Also

mc_transform_po_d2 (page 49) and mc_transform_po_array_d2 (page 50) for related APIs.

Part V. Appendixes

Appendix A. Examples

The following sections show examples for each of the random number generators and the distribution transformations.

More detailed examples can be found in the samples provided in the SDK samples.

The Pi sample found in /opt/cellsdk/src/samples/monte-carlo/pi/ performs a simple Monte Carlo simulation to calculate the value of pi using the vector versions of the RNG APIs.

The Sphere sample in /opt/cellsdk/src/samples/monte-carlo/sphere/ performs a more complex simulation to calculate the volume of an n-dimensional sphere, using both vector and array RNG APIs along with various levels of optimization in the Monte Carlo simulation.

Hardware-Generated Example

The following C example initializes the hardware RNG and then generates integer and double-precision numbers for display. Vector data is extracted using C `union` statements.

```
#include <stdio.h>
#include <stdlib.h>
#include <mc_rand.h>

int main(void) {

  int i;

  // Unions for scalar data extraction
  union vecInt {
    vector unsigned int vec_int;
    unsigned int scalar_int[4];
  };
  union vecIntArray {
    vector unsigned int vec_int_array[10];
    unsigned int scalar_int_array[40];
  };
  union vecDbl {
    vector double vec_double;
    double scalar_double[2];
  };
  union vecDblArray {
    vector double vec_double_array[10];
    double scalar_double_array[20];
  };

  // Random numbers
  union vecInt oneInt;
  union vecIntArray tenInts;
  union vecDbl oneDouble;
  union vecDblArray tenDoubles;

  // Initialize HW RNG and check for support
  if (0 != mc_rand_hw_init()) {
    printf("The HW RNG is not supported\n");
    return -1;
  }

  // Generate single vector random numbers
  oneInt.vec_int = mc_rand_hw_u4();
  oneDouble.vec_double = mc_rand_hw_0_to_1_d2();

  // Generate array of vectors
  mc_rand_hw_array_u4(10,tenInts.vec_int_array);
  mc_rand_hw_minus1_to_1_array_d2(10,
    tenDoubles.vec_double_array);
```

```
  // Display single vectors
  printf("vec_double = %e %e\n",
    oneDouble.scalar_double[0],
    oneDouble.scalar_double[1]);
  printf("vec_int = %u %u %u %u\n",
    oneInt.scalar_int[0],
    oneInt.scalar_int[1],
         oneInt.scalar_int[2],
    oneInt.scalar_int[3]);

  // Display array of vectors
  int j=0, k=0;
  for (i=0;i<10;i++)
  {
    j = i*4;
    k = i*2;
    printf("vec_int_array[%d] = %u %u %u %u\n",i,
      tenInts.scalar_int_array[j],
      tenInts.scalar_int_array[j+1],
      tenInts.scalar_int_array[j+2],
      tenInts.scalar_int_array[j+3]);
    printf("double_array[%d] = %e %e\n",i,
      tenDoubles.scalar_double_array[k],
      tenDoubles.scalar_double_array[k+1]);
  }

  return 0;
}
```

Kirkpatrick-Stoll Example

Like the preceding hardware RNG, the following C example initializes the Kirkpatrick-Stoll RNG and generates integer and double-precision numbers for display. This time, however, the program extracts scalar data from the vectors by using pointer addressing.

```
#include <stdio.h>
#include <stdlib.h>
#include <mc_rand.h>

int main(void) {

  int i;

  // Pointers for scalar data extraction
  unsigned int * p_int;
  double * p_double;

  // Random numbers
  vector unsigned int vec_int;
  vector unsigned int vec_int_array[10];
  vector double vec_double;
  vector double vec_double_array[10];

  // Initialize KS RNG
  mc_rand_ks_init( 4711U );

  // Generate single vector random numbers
  vec_int = mc_rand_ks_u4();
  vec_double = mc_rand_ks_0_to_1_d2();

  // Generate array of vectors
  mc_rand_ks_array_u4(10,vec_int_array);
  mc_rand_ks_minus1_to_1_array_d2(10,vec_double_array);

  // Display single vectors
  p_double = (double *)&vec_double;
  p_int = (unsigned int *)&vec_int;
  printf("vec_double = %e %e\n",
    p_double[0],p_double[1]);
  printf("vec_int = %u %u %u %u\n",
    p_int[0],p_int[1],p_int[2],p_int[3]);

  // Display array of vectors
  p_double = (double *)vec_double_array;
  p_int = (unsigned int *)vec_int_array;
  for (i=0;i<10;i++)
  {
    printf("vec_int_array[%d] = %u %u %u %u\n",
      i,p_int[0],p_int[1],p_int[2],p_int[3]);
    printf("vec_double_array[%d] = %e %e\n",i,
```

```
      p_double[0],p_double[1]);
    ++p_double;
    ++p_int;
  }

  return 0;
}
```

Mersenne Twister Example

Like the preceding two examples, this C program utilizes a Mersenne Twister RNG to generate integer and double-precision numbers for display. To illustrate yet another scalar data extraction technique, the program calls SPU intrinsics to retrieve scalar values.

```
#include <stdio.h>
#include <stdlib.h>
#include <mc_rand.h>

int main(void) {

  int i;
  vector unsigned int vec_int;
  vector unsigned int vec_int_array[10];
  vector double vec_double;
  vector double vec_double_array[10];

  mc_rand_mt_init( 123456U );

  vec_int = mc_rand_mt_u4();
  vec_double = mc_rand_mt_0_to_1_d2();

  mc_rand_mt_array_u4(10,vec_int_array);
  mc_rand_mt_minus1_to_1_array_d2(10,vec_double_array);

  printf("vec_double = %e %e\n",
    spu_extract(vec_double,0),
    spu_extract(vec_double,1));
  printf("vec_int = %u %u %u %u\n",
    spu_extract(vec_int,0),
    spu_extract(vec_int,1),
    spu_extract(vec_int,2),
    spu_extract(vec_int,3));

  for (i=0;i<10;i++)
  {
    printf("vec_int_array[%d] = %u %u %u %u\n",i,
      spu_extract(vec_int_array[i],0),
      spu_extract(vec_int_array[i],1),
      spu_extract(vec_int_array[i],2),
      spu_extract(vec_int_array[i],3));
    printf("vec_double_array[%d] = %e %e\n",i,
      spu_extract(vec_double_array[i],0),
      spu_extract(vec_double_array[i],1));
  }

  return 0;
}
```

Sobol Example

The fourth and final example illustrates the same general ideas as previous examples-- generate integer and double-precision numbers using the Sobol RNG for display.

The code on the SPU resembles previous examples:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <spu_mfcio.h>
#include <mc_rand.h>

// Control block from PPU
sobol_cntrlblck_t  sobolCB
  __attribute__ ((aligned(128)));

// Memory needed for rand_sb_init.  This size is
// related to the parameters passed to rand_sb_init
// and must be calculated in conjunction with those
// parameter values.
// The formula is:
//   size = (parm 2 + 76) * parm 3 + 8
// The exact size in this example would be:
//   1004 = ((256+76)*3)+8

vector unsigned char rand_sb_mem[1004]
  __attribute__ ((aligned(128)));


int main(unsigned long long speid,
         unsigned long long argp,
         unsigned long long envp) {

  unsigned int tag = 1;
  unsigned int tag_mask = 1<<tag;

  int i,rc;
  vector float vec_flt;
  vector float vec_flt_array[10];


  // DMA control block into local store
  mfc_get(&sobolCB, (unsigned int)argp, 128, tag,
    0, 0);
  mfc_write_tag_mask(tag_mask);
  mfc_read_tag_status_all();

  // Initialize sobol generator
  rc = rand_sb_init(&sobolCB, 256,
    sobolCB.u32TableDimension, &rand_sb_mem[0],
```

```
    sizeof(rand_sb_mem));
  if (rc != 0)
  {
    printf("Error: Sobol Init failed with RC=%d/n",
      i);
    return(-1);
  }

  // Generate scalar values
  vec_flt = rand_sb_0_to_1_f4();

  // Generate array values
  memcpy(vec_flt_array,
    rand_sb_minus1_to_1_array_f4(10), 10*16 );

  // Display results
  printf("vec_float = %e %e %e %e\n",
    spu_extract(vec_flt,0),
    spu_extract(vec_flt,1),
    spu_extract(vec_flt,2),
    spu_extract(vec_flt,3));

  for (i=0;i<10;i++)
  {
    printf("vec_float_array[%d] = %e %e %e %e\n",i,
      spu_extract(vec_flt_array[i],0),
      spu_extract(vec_flt_array[i],1),
      spu_extract(vec_flt_array[i],2),
      spu_extract(vec_flt_array[i],3));
  }

  return 0;
}
```

Two differences occur between the Sobol RNG and the previous examples. First, the array of vectors RNG interface (`mc_rand_sb_array_f4`) returns a pointer to the data instead of putting the data in a specified location--producing and additional `memcpy` not in previous examples.

Second, this random number generator requires more data for initialization than previous examples. In this example, the data is DMA'd from the PPU. The details of this data are highlighted in the PPU code shown below:

```
#include <libspe2.h>
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/wait.h>
#include <string.h>
#include <mc_rand_sb.h>
```

```
// sobol direction vector table for 30 bits and 3
// dimensions


// Initialized for 5 bits x 3 dimensions
// dimensions a, b, c
// bits 0, 1, 2, 3

//  a0 b0 c0
//  a1 b1 c1
//  a2 b2 c2
//  a3 b3 c3
//  a4 b4 c4



// Maximum allowed space dimension
#undef SOBOL_MAX_DIMENSION
#define SOBOL_MAX_DIMENSION 3

// Bit count; assumes sizeof(int) >= 32-bit
#undef SOBOL_BIT_COUNT
#define SOBOL_BIT_COUNT 30

// This table is a 2D array
// bits x dimensions

vector unsigned int u32_vecDirections
[(SOBOL_BIT_COUNT+3)>>2][SOBOL_MAX_DIMENSION] = {
{  // bits 0-3
  {536870912, 268435456, 134217728, 67108864},  // a
  {536870912, 805306368, 671088640, 1006632960},// b
  {536870912, 268435456, 939524096, 738197504}},// c
{ // bits 4-7
  // NOTE: Zeroed values represent unused bits
  { 33554432, 0, 0, 0 },                          // a
  { 570425344, 0, 0, 0 },                         // b
  { 436207616, 0, 0, 0 }},                        //  c
{ { 0, 0, 0, 0 },
{ 0, 0, 0, 0 },
{ 0, 0, 0, 0 }},   // 2
{ { 0, 0, 0, 0 },
{ 0, 0, 0, 0 },
{ 0, 0, 0, 0 }},   // 3
{ { 0, 0, 0, 0 },
{ 0, 0, 0, 0 },
{ 0, 0, 0, 0 }},   // 4
{ { 0, 0, 0, 0 },
{ 0, 0, 0, 0 },
{ 0, 0, 0, 0 }},   // 5
{ { 0, 0, 0, 0 },
{ 0, 0, 0, 0 },
{ 0, 0, 0, 0 }},   // 6
{ { 0, 0, 0, 0 },
```

```
{ 0, 0, 0, 0 },
{ 0, 0, 0, 0 }}    // 7
};

inline static void rand_sobol set CB(sobol cntrlblck t
  *sobolCB, unsigned int u32Seed)
{
  sobolCB->pu32_vecDirection =
    &vecDirections[0][0];
  sobolCB->u32sizeofTable = sizeof(u32_vecDirections);
  sobolCB->u32TableDimension = SOBOL_MAX_DIMENSION;
  sobolCB->u32TableBitCount =
    (SOBOL_BIT_COUNT+3)&0xFFFFFFFC;
  sobolCB->u32MaxBitCount = SOBOL_BIT_COUNT;
  sobolCB->u32Seed = u32Seed;

  return;
}

sobol_cntrlblck_t  sobolCB
  __attribute__ ((aligned(128)));

// This is the pointer to the SPE code, to be used at
// thread creation time
extern spe_program_handle_t spu_rand;

void *ppu_pthread_function(void *arg) {
  spe_context_ptr_t context =
    *(spe_context_ptr_t *)arg;
  unsigned int entry = SPE_DEFAULT_ENTRY;
  spe_stop_info_t stop_info;
  int rc = spe_context_run(context,&entry,0,
     &sobolCB, NULL, &stop_info);
  if (rc < 0 ) perror("spe_context_run");
  pthread_exit(NULL);
}

int main( void )
{
  // Initialize Sobol control block
  rand_sobol_set_CB(&sobolCB, 0);

  // Create SPE thread
  pthread_t pthread;
  spe_context_ptr_t spe_context =
    spe_context_create( 0, NULL );
  spe_program_load( spe_context, &spu_rand );

  // Start SPE thread
  pthread_create(&pthread,NULL,&ppu_pthread_function,
    &spe_context);

  // Wait for thread completion
  pthread_join(pthread,NULL);
```

```
  spe_context_destroy( spe_context );

  printf("PPE: Done\n");

  return 0;
}
```

The above example shows how to instantiate a 5-bit by 3-dimension initialization table. Zero values in the *vecDirections* array represent unused initialization values for the example.

In practice, applications are likely to have their own initialization table for the Sobol algorithm. This table, like *vecDirections*, will need to be a two-dimensional array with the first index being up to 8 in size to represent the maximum bits and the second index representing the maximum number of RNG dimensions. Applications may elect to create their initialization data or use a basic set found in /opt/cell/sdk/src/samples/monte-carlo/sobol_init_30_40.h.

Although this example represents a simple instance of the Sobol RNG, even more complex applications will follow a similar structure of instantiating the Sobol initialization table on the PPU, creating and initializing a control block on the PPU, transferring this control to the SPU, and then initializing and invoking the SPU to generate numbers.

Additional examples of the Sobol and other random number generators can be found in /opt/cell/sdk/src/samples/monte-carlo.

Box-Muller Example

The following example generates data using the Mersenne Twister RNG, transforms it using the Box-Muller algorith, and displays the data. Array-based APIs are used for both number generation and transformation.

```
#include <stdio.h>
#include <stdlib.h>
#include <mc_rand.h>

#define NUM_RN_VECTORS 8

// Source RNs
vector float rn_source_f4[ NUM_RN_VECTORS ];
vector double rn_source_d2[ NUM_RN_VECTORS ];

// Transformed RNs
// NOTE: For Box-Mueller the transform data is twice
//       as large as the source data!!!
vector float rn_transform_f4[ NUM_RN_VECTORS*2 ];
vector double rn_transform_d2[ NUM_RN_VECTORS*2 ];

int main( void){

  int i;

  float *p_float_source, *p_float_transform;
  double *p_double_source, *p_double_transform;

  // Initialize RNG
  mc_rand_mt_init( 4711U );

  // Generate source data
  mc_rand_mt_0_to_1_array_f4( NUM_RN_VECTORS,
    rn_source_f4 );
  mc_rand_mt_0_to_1_array_d2( NUM_RN_VECTORS,
    rn_source_d2 );

  // Transform data
  mc_transform_bm_array_f4( NUM_RN_VECTORS,
    rn_source_f4, rn_transform_f4 );
  mc_transform_bm_array_d2( NUM_RN_VECTORS,
    rn_source_d2, rn_transform_d2 );

  // Float data
  for (i=0; i< NUM_RN_VECTORS; i++)
  {
     // Set pointer to current location
    p_float_source =
      (float *) &rn_source_f4[i];
    p_float_transform =
      (float *) &rn_transform_f4[i*2];
```

```
    // Output data
    printf("Float source: %f, %f, %f, %f\n",
      p_float_source[0], p_float_source[1],
      p_float_source[2], p_float_source[3]);
    printf("Float transform: %f, %f, %f, %f\n",
      p_float_transform[0], p_float_transform[1],
      p_float_transform[2], p_float_transform[3]);
    printf("Float transform: %f, %f, %f, %f\n",
      p_float_transform[4], p_float_transform[5],
      p_float_transform[6], p_float_transform[7]);
  }

  // Double data
  for (i=0; i< NUM_RN_VECTORS; i++)
  {
     // Set pointer to current location
    p_double_source =
      (double *) &rn_source_d2[i];
    p_double_transform =
      (double *) &rn_transform_d2[i*2];

    // Output data
    printf("Double source: %f, %f\n",
      p_double_source[0], p_double_source[1]);
    printf("Double transform: %f, %f\n",
      p_double_transform[0], p_double_transform[1]);
    printf("Double transform: %f, %f\n",
      p_double_transform[2], p_double_transform[3]);
  }

  return 0;
}
```

Programs using this transformation APIs need to ensure that the memory allocated for transformed data is twice as large as the source data. The above example accomplishes this by allocated `NUM_RN_VECTORS` for the source arrays and `NUM_RN_VECTORS*2` for the transformed arrays.

Moro's Inversion Example

Moro's Inversion algorithm transforms data most simply—one value in, one value out. The following code generates data using the Kirkpatrick-Stoll RNG, transforms it, and displays all data.

```
#include <stdio.h>
#include <stdlib.h>
#include <mc_rand.h>

#define NUM_RN_VECTORS 8

// Source RNs
vector float rn_source_f4[ NUM_RN_VECTORS ];
vector double rn_source_d2[ NUM_RN_VECTORS ];

// Transformed RNs
vector float rn_transform_f4[ NUM_RN_VECTORS ];
vector double rn_transform_d2[ NUM_RN_VECTORS ];

int main( void){

  int i;

  float *p_float_source, *p_float_transform;
  double *p_double_source, *p_double_transform;

  // Initialize RNG
  mc_rand_ks_init( 4711U );

  // Generate source data
  mc_rand_ks_0_to_1_array_f4( NUM_RN_VECTORS,
    rn_source_f4 );
  mc_rand_ks_0_to_1_array_d2( NUM_RN_VECTORS,
    rn_source_d2 );

  // Transform data
  mc_transform_mi_array_f4( NUM_RN_VECTORS,
    rn_source_f4, rn_transform_f4 );
  mc_transform_mi_array_d2( NUM_RN_VECTORS,
    rn_source_d2, rn_transform_d2 );

  // Float data
  for (i=0; i< NUM_RN_VECTORS; i++)
  {
     // Set pointer to current location
    p_float_source = (float *) &rn_source_f4[i];
    p_float_transform = (float *) &rn_transform_f4[i];

    // Output data
    printf("Float source: %f, %f, %f, %f\n",
      p_float_source[0], p_float_source[1],
```

```
      p_float_source[2], p_float_source[3]);
    printf("Float transform: %f, %f, %f, %f\n",
      p_float_transform[0], p_float_transform[1],
      p_float_transform[2], p_float_transform[3]);
  }

  // Double data
  for (i=0; i< NUM_RN_VECTORS; i++)
  {
     // Set pointer to current location
    p_double_source =
      (double *) &rn_source_d2[i];
    p_double_transform =
      (double *) &rn_transform_d2[i];

    // Output data
    printf("Double source: %f, %f\n",
      p_double_source[0], p_double_source[1]);
    printf("Double transform: %f, %f\n",
      p_double_transform[0], p_double_transform[1]);
  }

  return 0;
}
```

Polar Method Example

The Polar Method transformation APIS represents the most complex interfaces due to the accept-reject nature of the algorithm. Most interfaces require a function pointer to a random number generator to allow for generation of more values as needed.

The following example shows all three types of Polar Method interfaces. The first set of `float` data is generated using the array interface and transformed using the array interface `mc_transform_reject_po_array_f4` which rejects values without replacement.

The `double` data is generated using the array interface `mc_transform_po_array_d2` and then transformed with rejects being replaced using new values.

The second set of `float` data is generated and transformed a vector at a time, using a single API invocation of `mc_transform_po_f4`.

```
#include <stdio.h>
#include <stdlib.h>
#include <mc_rand.h>

#define NUM_RN_VECTORS 8

// Source RNs
vector float rn_source_f4[ NUM_RN_VECTORS ];
vector double rn_source_d2[ NUM_RN_VECTORS ];

// Transformed RNs
vector float rn_transform_1_f4[ NUM_RN_VECTORS];
vector float rn_transform_2_f4[ NUM_RN_VECTORS];
vector double rn_transform_d2[ NUM_RN_VECTORS ];

int main( void){

  int i, num_transformed;

  float *p_float_source, *p_float_transform;
  double *p_double_source, *p_double_transform;

  // Initialize RNG
  mc_rand_ks_init( 4711U );

  // Generate source data
  mc_rand_ks_0_to_1_array_f4( NUM_RN_VECTORS,
    rn_source_f4 );
  mc_rand_ks_0_to_1_array_d2( NUM_RN_VECTORS,
    rn_source_d2 );

  // Transform float data first time--passing an array,
```

```
// rejecting data (without an RNG), returning count
num_transformed =
  mc_transform_reject_po_array_f4( NUM_RN_VECTORS,
    rn_source_f4, rn_transform_1_f4 );

// Tranform double data using an RNG
mc_transform_po_array_d2( NUM_RN_VECTORS,
  rn_source_d2, rn_transform_d2,
  mc_rand_ks_minus1_to_1_d2 );

// Generate and transform floats a second time--
// one vector at at time, using an RNG
for (i=0; i < NUM_RN_VECTORS; i++ )
{
  rn_transform_2_f4[i] =
    mc_transform_po_f4( mc_rand_ks_minus1_to_1_f4 );
}

// Float data - No RNG
for (i=0; i< NUM_RN_VECTORS; i++)
{
   // Set pointer to current location
  p_float_source =
    (float *) &rn_source_f4[i];
  p_float_transform =
    (float *) &rn_transform_1_f4[i];

  // Output data
  printf("Float source: %f, %f, %f, %f\n",
    p_float_source[0], p_float_source[1],
    p_float_source[2], p_float_source[3]);
   if ( i < num_transformed )
  {
    printf(
      "Float transform (Reject): %f, %f, %f, %f\n",
      p_float_transform[0], p_float_transform[1],
      p_float_transform[2], p_float_transform[3]);
  }
}

// Double data
for (i=0; i< NUM_RN_VECTORS; i++)
{
   // Set pointer to current location
  p_double_source =
    (double *) &rn_source_d2[i];
  p_double_transform =
    (double *) &rn_transform_d2[i];

  // Output data
  printf("Double source: %f, %f\n",
    p_double_source[0], p_double_source[1]);
  printf("Double transform: %f, %f\n",
    p_double_transform[0], p_double_transform[1]);
```

```
  }

  // Float data - RNG
  for (i=0; i< NUM_RN_VECTORS; i++)
  {
     // Set pointer to current location
    p_float_transform =
      (float *) &rn_transform_2_f4[i];

    // Output data
    printf("Float transform (RNG): %f, %f, %f, %f\n",
      p_float_transform[0], p_float_transform[1],
      p_float_transform[2], p_float_transform[3]);
  }

  return 0;
}
```

Appendix B. Getting Help or Technical Assistance

If you need help, service, or technical assistance or just want more information about IBM products, you will find a wide variety of sources available from IBM to assist you. This appendix contains information about where to go for additional information about IBM and IBM products and whom to call for service, if it is necessary.

Using the Documentation

Information about your IBM hardware or software is available in the documentation that comes with the product. That documentation can include printed documents, online documents, readme files, and help files. See the troubleshooting information in your documentation for instructions for using diagnostic programs. The troubleshooting information or the diagnostic programs might tell you that you need additional or updated device drivers or other software. IBM maintains pages on the World Wide Web where you can get the latest technical information and download device drivers and updates. To access these pages, go to http://www.ibm.com/bladecenter/, click Support, and follow the instructions. Also, some documents are available through the IBM Publications Center at http://www.ibm.com/shop/publications/order/.

Getting Help and Information from the World Wide Web

You can locate documentation and other resources on the World Wide Web. Refer to the following web sites:

- IBM BladeCenter systems, optional devices, services, and support information at http://www.ibm.com/bladecenter/. For service information, select Support.
- developerWorks® Cell/B.E. Resource Center at http://www.ibm.com/developerworks/power/cell/. To access the Cell/B.E. forum on developerWorks, select Community.
- The Barcelona Supercomputing Center (BSC) Web site at http://www.bsc.es/projects/deepcomputing/linuxoncell.
- There is also support for the Full-System Simulator and XL C/C++ Compiler through their individual alphaWorks® forums. If in doubt, start with the Cell/B.E. architecture forum.
- The GNU Project debugger, GDB, is supported through many different forums on the Web, but primarily at the GDB Web site http://www.gnu.org/software/gdb/gdb.html.

Contacting IBM Support

To obtain telephone assistance, for a fee or on a support contract, contact IBM Support. In the U.S. and Canada, call 1-800-IBM-SERV (1-800-426-7378), or see http://www.ibm.com/planetwide/ for support telephone numbers.

Appendix C. Accessibility

Accessibility features help users who have a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

The following list includes the major accessibility features:

- Keyboard-only operation
- Interfaces that are commonly used by screen readers
- Keys that are tactilely discernible and do not activate just by touching them
- Industry-standard devices for ports and connectors
- The attachment of alternative input and output devices

IBM® and accessibility

See the IBM Accessibility Center at http://www.ibm.com/able/ for more information about the commitment that IBM has to accessibility.

Appendix D. Notices

Code License and Disclaimer Information

IBM grants you a nonexclusive copyright license to use all programming code examples from which you can generate similar function tailored to your own specific needs.

SUBJECT TO ANY STATUTORY WARRANTIES WHICH CANNOT BE EXCLUDED, IBM, ITS PROGRAM DEVELOPERS AND SUPPLIERS MAKE NO WARRANTIES OR CONDITIONS EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT, REGARDING THE PROGRAM OR TECHNICAL SUPPORT, IF ANY.

UNDER NO CIRCUMSTANCES IS IBM, ITS PROGRAM DEVELOPERS OR SUPPLIERS LIABLE FOR ANY OF THE FOLLOWING, EVEN IF INFORMED OF THEIR POSSIBILITY:

1. LOSS OF, OR DAMAGE TO, DATA;
2. DIRECT, SPECIAL, INCIDENTAL, OR INDIRECT DAMAGES, OR FOR ANY ECONOMIC CONSEQUENTIAL DAMAGES; OR
3. LOST PROFITS, BUSINESS, REVENUE, GOODWILL, OR ANTICIPATED SAVINGS.

SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF DIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, SO SOME OR ALL OF THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

Trademarks

developerWorks, DB2, IBM, the IBM logo, ibm.com, and PowerPC are registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

Cell Broadband Engine and Cell/B.E. are trademarks of Sony Computer Entertainment, Inc. in the United States, other countries, or both and is used under license therefrom.

Other company, product or service names may be trademarks or service marks of others.