CBEA JSRE Series
Cell Broadband Engine Architecture Joint Software
Reference Environment Series

SPE Runtime Management Library Version 2.2 DRAFT

SC33-8334-01

CBEA JSRE Series
Cell Broadband Engine Architecture Joint Software
Reference Environment Series

SPE Runtime Management Library Version 2.2 DRAFT

SC33-8334-01

Note
Before using this information and the product it supports, read the information in "Notices" on page 99.

Edition Notice

This edition applies to the SPE Runtime Management Library Version 2.2 and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces SC33-8334-00.

Preface

About this book

This document describes the SPE Runtime Management Library. This library constitutes the standardized low-level application programming interface for application access to the Cell Broadband Engine™ (Cell BE™) Synergistic Processing Elements (SPEs).

Who should read this book

The document is intended for system and application programmers who wish to develop Cell BE. applications that fully exploit the SPEs.

Prerequisites

This document and the use of the library assumes and requires that you are familiar with the Cell BE. architecture as described in *Cell Broadband Engine Architecture*.

New in this release

This section describes significant changes made to the SPE Runtime Management Library specification for each version of this document.

Version number and date	Changes
Version 2.0 November, 2006	New library version (previous version was 1.2)
Version 2.1 March, 2007	New functions: • spe_create_context_affinity • spe_cpu_info_get • spe_callback_handler_query
Version 2.2 September, 2007	New functions: • spe_mssync_start • spe_mssync_status

Other documentation

The following is a list of reference and supporting materials for the SPE Runtime Management Library specification:

- *Cell Broadband Engine Architectur*
- *Cell Broadband Engine Programming Handbook*
- *C/C++ Language Extensions for Cell Broadband Engine Architecture*

For a full list of documentation, see Appendix C, “Related documentation,” on page 95.

Contents

Chapter 1. Overview

The SPE Runtime Management Library (libspe) is the standardized low-level application programming interface (API) that enables application access to the Cell BE SPEs. This library provides an API that is neutral with respect to the underlying operating system and its methods to manage SPEs.

Implementations of libspe can provide additional functionality that enables access to operating system or implementation-dependent aspects of SPE runtime management.

Note: This functionality is not subject to standardization in this document and its use can lead to non-portable code and dependencies on certain implemented versions of the library.

In general, applications do not have control over the physical SPE system resources. The operating system manages these resources. Applications manage and use software constructs called **SPE contexts**. These SPE contexts are a logical representation of an SPE and are the base object on which libspe operates. The operating system schedules SPE contexts from all running applications onto the physical SPE resources in the system for execution according to the scheduling priorities and policies associated with the runable SPE contexts.

libspe also provides the means for communication and data transfer between PPE threads and SPEs.

The basic scheme for a simple application using an SPE is as follows:

1. Create an SPE context.
2. Load an SPE executable object into the SPE context local store.
3. Run the SPE context. This transfers control to the operating system, which requests the actual scheduling of the context onto a physical SPE in the system.
4. Destroy the SPE context.

Note: Step 3 represents a synchronous call to the operating system. The calling application blocks until the SPE stops executing and the operating system returns from the system call that invoked the SPE execution.

Using multiple SPEs concurrently

Many applications need to use multiple SPEs concurrently. In this case, the application must create at least as many threads as concurrent SPE contexts are required. Each of these threads may run a single SPE context at a time. If *N* concurrent SPE contexts are needed, it is common to have a main application thread plus *N* threads dedicated to SPE context execution.

The basic scheme for a simple application running *N* SPE contexts is as follows:

1. Create *N* SPE contexts.
2. Load the appropriate SPE executable object into each SPE context's local store.
3. Create *N* threads:
   a. In each of these threads run one of the SPE contexts.
   b. Stop thread.

4. Wait for all *N* threads to stop.
5. Destroy all *N* SPE contexts.

Other schemes are also possible and, depending on the application, potentially more suitable.

PPE functions

To provide this functionality, libspe consists of the following sets of PPE (PowerPC® Processing Element) functions to:

- Create and destroy SPE and gang contexts
- Load SPE objects into SPE local store memory for execution
- Start the execution of SPE programs and to obtain information about reasons why an SPE has stopped running
- Receive asynchronous events generated by an SPE
- Access the MFC (Memory Flow Control) problem state facilities, which includes:
  - MFC proxy command issue
  - MFC proxy tag-group completion facility
  - Mailbox facility
  - SPE signal notification facility
- Enable direct application access to an SPE's local store and problem state areas
- Register PPE-assisted library calls for an SPE program

Terminology

Library Name(s)
libspe2

Header File(s)
<libspe2.h>

For a full list of terms, see "Glossary" on page 103

Example

The following example shows how to load and run a simple SPE executable "hello".

Example 1: Run the simple SPE program "hello"

```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include "libspe2.h"

int main(void)
{
        spe_context_ptr_t ctx;
        int flags = 0;
        unsigned int entry = SPE_DEFAULT_ENTRY;
        void * argp = NULL;
        void * envp = NULL;
        spe_program_handle_t * program;
        spe_stop_info_t stop_info;
        int rc;

        program = spe_image_open("hello");
        if (!program) {
```

```
                perror("spe_open_image");
                return -1;
        }

        ctx = spe_context_create(flags, NULL);
        if (ctx == NULL) {
                perror("spe_context_create");
                return -2;
        }
        if (spe_program_load(ctx, program)) {
                perror("spe_program_load");
                return -3;
        }
        rc = spe_context_run(ctx, &entry, 0, argp, envp, &stop_info);
        if (rc < 0)
                perror("spe_context_run");

        spe_context_destroy(ctx);

        return 0;
}
```

The following simple multi-threaded example shows how an application can run the SPE program ″hello″ on multiple SPEs concurrently:

Example 2: Simple multi-threaded example

```
#include <stdlib.h>
#include <pthread.h>
#include "libspe2.h"

struct thread_args {
        struct spe_context * ctx;
        void * argp;
        void * envp;
};

void * spe_thread(void * arg)
{
        int flags = 0;
        unsigned int entry = SPE_DEFAULT_ENTRY;
        spe_program_handle_t * program;
        struct thread_args * arg_ptr;

        arg_ptr = (struct thread_args *) arg;

        program = spe_image_open("hello");
        spe_program_load(arg_ptr->ctx, program);
        spe_context_run(arg_ptr->ctx, &entry, flags, arg_ptr->argp,
                        arg_ptr->envp, NULL);
        pthread_exit(NULL);
}

int main() {
        int thread_id;
        pthread_t pts;
        spe_context_ptr_t ctx;
        struct thread_args t_args;
        int value = 1;

        ctx = spe_context_create(0, NULL);

        t_args.ctx = ctx;
        t_args.argp = &value;

        thread_id = pthread_create( &pts, NULL, &spe_thread, &t_args);
```

```
        pthread_join (pts, NULL);
        spe_context_destroy (ctx);

        return 0;
}
```

Chapter 2. SPE context creation

The SPE context is one of the base data structures for the libspe implementation. It holds all persistent information about a ″logical SPE″ used by the application. This data structure should only be accessed through libspe API calls, and should not be accessed directly.

Before being able to use an SPE, the SPE context data structure has to be created and initialized. This is done by calling the function **spe_context_create**.

When an application no longer needs a specific SPE context, it should call the function **spe_context_destroy** to release all associated resources and free the memory used by the SPE context data structure.

The SPE gang context is another of the base data structures for the libspe implementation. It holds all persistent information about a group of SPE contexts that should be treated as a gang, that is, be executed together with certain properties. This data structure should only be accessed through libspe API calls, and should not be accessed directly.

Before being able to use an SPE gang context, that is, before calling **spe_context_create** to add SPE contexts as members to the gang, the SPE gang context data structure must be created and initialized. This is done by calling the function **spe_gang_context_create**.

When an application no longer needs a specific SPE gang context, it should release all associated resources and free the memory used by the SPE context data structure. It does this by first calling **spe_context_destroy** to destroy all SPE contexts associated with the gang by on each of them and then calling the function **spe_gang_context_destroy**.

SPE-SPE affinity is always specified in affinity pairs. The function **spe_context_create_affinity** specifies SPE affinity. This function allows an SPE context to be created and placed next to another previously created SPE context. The SPUFS scheduler honors this relationship by scheduling the SPE contexts on physically adjacent SPUs. This routine can be used to create a chain of SPE contexts that consumes all of the SPU resources on a Cell BE. If you want to use additional SPU resources, you must create a separate gang for that purpose.

SPE context creation functions

The following describe the SPE context creation functions.

spe_context_create

NAME

spe_context_create - Create a new SPE context.

SYNOPSIS

#include <libspe2.h>

spe_context_ptr_t spe_context_create(unsigned int *flags*, **spe_gang_context_ptr_t** *gang***)**

Parameters

flags	A bit-wise OR of modifiers that are applied when the SPE context is created. See *Usage*.
gang	Associate the new SPE context with this gang context. If NULL is specified, the new SPE context is not associated with any gang.

RETURN VALUE

On success, a pointer to the newly created SPE context is returned.

EXIT STATUS

On error, **NULL** is returned and *errno* is set to indicate the error.

Possible errors include:

ENOMEM	The SPE context could not be allocated due to lack of system resources.
EINVAL	The value passed for flags was invalid.
EPERM	The process does not have permission to add threads to the designated SPE gang context, or to use the SPU_MAP_PS setting.
ESRCH	The gang context could not be found.
EFAULT	A runtime error of the underlying operating system service occurred.
ENODEV	An isolated SPE context has been requested but the system is not isolation-enabled.

OPTIONS

The following values are accepted for the *flags* parameter:

SPE_EVENTS_ENABLE	Enable event handling on this SPE context
SPE_CFG_SIGNOTIFY1_OR	Configure the SPU Signal Notification 1 Register to be in ″logical OR″ mode instead of the default ″Overwrite″ mode. See *Cell Broadband Engine Architecture, SPU Signal Notification Facility.*

SPE_CFG_SIGNOTIFY2_OR	Configure the SPU Signal Notification 2 Register to be in "logical OR" mode instead of the default "Overwrite" mode. See *Cell Broadband Engine Architecture, SPU Signal Notification Facility*.
SPE_MAP_PS	Request permission for memory-mapped access to the SPE's problem state area(s). See *Cell Broadband Engine Architecture, Problem State Memory-Mapped Registers*.
SPE_ISOLATED	This context executes on an SPU in the isolation mode. Programs loaded into contexts flagged with SPE_ISOLATED must be be correctly formatted for isolated execution.
SPE_ISOLATED_EMULATE	Run this context on an SPU in an emulated isolation mode. This mode provides emulation of an isolated SPU without truly being isolated as is intended for use by developers who need access to debug tools during the development of their isolated applications. Programs loaded into contexts flagged with SPE_ISOLATED_EMULATE must be correctly formatted for isolated emulation execution. **Note:** (Linux) Proper operation of a PPE assisted function call assumes the use of the ISOLATED version of the SPE library functions.

SEE ALSO

spe_context_destroy(3); spe_gang_context_create(3)

Cell Broadband Engine Architecture, SPU Signal Notification Facility

Cell Broadband Engine Architecture, Problem State Memory-Mapped Registers

spe_context_destroy

NAME

spe_context_destroy - Destroy the specified SPE context.

SYNOPSIS

#include <libspe2.h>
int spe_context_destroy (spe_context_ptr_t *spe***)**

Parameters

spe	Specifies the SPE context to be destroyed.

DESCRIPTION

Destroy the specified SPE context and free any associated resources.

RETURN VALUE

On success, **0** (zero) is returned.

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE context is invalid.
EAGAIN	The specified SPE context cannot be destroyed at this time because it is in use.
EFAULT	A runtime error of the underlying operating system service occurred.

SEE ALSO

spe_context_create(3)

spe_gang_context_create

NAME

spe_gang_context_create - Create a new SPE gang context.

SYNOPSIS

#include <libspe2.h>

spe_gang_context_ptr_t spe_gang_context_create (unsigned int *flags***)**

Parameters

flags	A bit-wise OR of modifiers that are applied when the SPE context is created. See *Usage*.

RETURN VALUE

On success, a pointer to the newly created gang context is returned.

EXIT STATUS

On error, **NULL** is returned and *errno* is set to indicate the error.

Possible errors include:

ENOMEM	The gang context could not be allocated due to lack of system resources.
EINVAL	The value passed for flags is not valid.
EFAULT	A runtime error of the underlying operating system service occurred.

USAGE

The following values are accepted for the flags parameter:

<none>	none

SEE ALSO

spe_context_destroy(3); spe_gang_sccontext_destroy(3)

spe_gang_context_destroy

NAME

spe_gang_context_destroy - Destroy the specified gang context.

SYNOPSIS

#include <libspe2.h>

int spe_gang_context_destroy (spe_gang_context_ptr_t *gang***)**

Parameters

gang — Specifies the gang context to be destroyed.

DESCRIPTION

Destroy the specified gang context and free any associated resources. Before you destroy a gang context, you must destroy all associated SPE contexts using **spe_context_destroy**.

RETURN VALUE

On success, **0** (zero) is returned.

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified gang context is not valid.
EAGAIN	The specified gang context cannot be destroyed at this time since it is in use.
EFAULT	A runtime error of the underlying operating system service occurred.

SEE ALSO

spe_context_destroy(3); spe_gang_context_create(3)

spe_context_create_affinity

NAME

spe_context_create_affinity - Create a new SPE context with an affinity constraint.

SYNOPSIS

#include <libspe2.h>

**spe_context_ptr_t spe_context_create_affinity(unsigned int flags,
spe_context_ptr_t affinity_neighbor, spe_gang_context_ptr_t gang)**

Parameters

flags	A bit-wise OR of modifiers that are applied when the SPE context is created.
affinity_neighbor	The *affinity_neighbor* parameter identifies a previously created SPE context in the named gang. A NULL value may be specified for the initial SPE context. Or the **spe_context_create()** routine may be used to create the initial SPE context. The *affinity_neighbor* must be in the same gang as the newly created SPE context.
gang	Associate the new SPE context with this gang context. NULL is not permitted here, because affinity constraints can only be specified for members of the same gang.

DESCRIPTION

This function allows an SPE context to be created and placed next to another previously created SPE context. SPE-SPE Affinity is always specified in affinity pairs. The SPE scheduler honors this relationship by scheduling the SPE contexts on physically adjacent SPUs. This function can be used to create a chain of SPE contexts that consumes all of the available SPE resources on a Cell BE, but not more. If you want to use additional SPE resources, you must create a separate gang or individual SPE contexts for that purpose. All SPE contexts in the gang must be created before you run any SPE contexts in the gang.

RETURN VALUE

On success, a pointer to the newly created SPE context is returned.

EXIT STATUS

On error, **NULL** is returned and *errno* is set to indicate the error.

Possible errors include:

ENOTSUP	Platform does not support affinity (PlayStation 3 does not support affinity).
EEXIST	Too many references to affinity neighbor. Too many SPE contexts with memory affinity specified.

ESRCH	No such SPE context (affinity_neighbor is not valid). No such gang context (gang context could not be found).
EINVAL	Argument is not valid (bad flag value).
EPERM	Lack of resources (too many isolated SPUs). The process does not have permission to add threads to the designated SPE gang context, or to use the SPU_MAP_PS setting.
ENOMEM	The SPE context could not be allocated due to lack of system resources.
EFAULT	A runtime error of the underlying operating system service occurred.
EBUSY	Cannot add more SPE affinity contexts because an SPE context in the gang is already running.
ENODEV	An isolated SPE context has been requested but the system is not isolation-enabled.

OPTIONS

The following values for *flags* are accepted:

Flags	Description
SPE_EVENTS_ENABLE	Event handling shall be enabled on this SPE context
SPE_CFG_SIGNOTIFY1_OR	Configure the SPU Signal Notification 1 Register to be in ″logical OR″ mode instead of the default ″Overwrite″ mode.
SPE_CFG_SIGNOTIFY2_OR	Configure the SPU Signal Notification 2 Register1 to be in ″logical OR″ mode instead of the default ″Overwrite″ mode.
SPE_MAP_PS	Request permission for memory-mapped access to the SPE's problem state area(s).
SPE_ISOLATED	This context executes on an SPU in the isolation mode. Programs loaded into contexts flagged with SPE_ISOLATED must be be correctly formatted for isolated execution.
SPE_ISOLATED_EMULATE	Run this context on an SPU in an emulated isolation mode. This mode provides emulation of an isolated SPU without truly being isolated as is intended for use by developers who need access to debug tools during the development of their isolated applications. Programs loaded into contexts flagged with SPE_ISOLATED_EMULATE must be correctly formatted for isolated emulation execution. **Note:** (Linux) Proper operation of a PPE assisted function call assumes the use of the ISOLATED version of the SPE library functions.

SPE_AFFINITY_MEMORY	Request that the new SPE context is placed on an SPE that is considered to be the closest to main memory. Only one SPE context in the gang may be created with memory affinity.

SEE ALSO

spe_cpu_info_get(3)

Chapter 3. CPU information

Applications often require basic information about the system they are running on, such as number of CPUs (PPEs), number of SPEs, and other information about the processing environment. The following section describes the function **spe_cpu_info_get** which you can use to obtain this information.

spe_cpu_info_get

NAME

spe_cpu_info_get - Query basic CPU properties and resources.

SYNOPSIS

#include <libspe2.h>

int spe_cpu_info_get(unsigned int info_requested, int cpu_node)

Parameters	
info_requested	Specifies the type of information requested.
cpu_node	Specifies the node for which the information is requested. The numbering of CPU nodes is consistent with the numbering used by the NUMA control. This information can be used in conjunction with explicit NUMA control by the application.

DESCRIPTION

Applications often require some basic information about the system they are running on, such as number of CPUs (PPEs) or number of SPEs.

In the context of this API, the term ″system″ means the ″hardware″ seen by the currently running operating system, and the term ″physical″ refers to resources in that system. For example, in case of a hypervisor-based system, the result returned can be different from the actual number of items present in the hardware.

RETURN VALUE

On success, this function returns **0** (zero) or a positive value that indicates the value requested.

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

EINVAL	Function argument error

USAGE

The following values for *info_requested* are accepted:

Flag	Description
SPE_COUNT_PHYSICAL_CPU_NODES	Request the number of physical CPU nodes of the system
SPE_COUNT_PHYSICAL_SPES	Request the total number of physical SPEs available either on the whole system or on a specified node.

SPE_COUNT_USABLE_SPES	Request the number of SPEs that can actually be used by the application at this point in time. This is the number of SPEs that can actually be scheduled to run for the application, provided it has high enough scheduling priority. In particular, if the operating system reserves SPEs or (privileged) applications have ″pinned″ SPEs, that is, made them non-schedulable, these are not counted as usable SPEs.

The following values for *cpu_node* are accepted:

Flags	Description
-1	Request an aggregated result for the whole system.
0..(n-1)	Request information for this specific CPU node. *n* is the number of physical CPU nodes in the system. On platforms with enabled NUMA-support, the numbering of CPU nodes is consistent with the numbering used by the NUMA control. This information can be used in conjunction with explicit NUMA control by the application. On platforms with a single processor, the number of the CPU node is 0. On platforms with multiple processors but without enabled NUMA-support, the numbering of CPU nodes is not specified. In this case, the operating system may also not be able to determine the association of SPEs with CPU nodes properly.

EXAMPLES

Assume the application is running on a system which as two Cell BE processors with eight physical SPEs available on each CPU. The operating system has reserved one SPE on node 0 for some kernel tasks and a concurrently running application has two SPEs ″pinned″ (″reserved exclusively″, ″non-schedulable″) on node 1.

```
no_cpus = spe_cpu_info_get(SPE_COUNT_PHYSICAL_CPU_NODES, -1);
==> 2
no_phys_spes = spe_cpu_info_get(SPE_COUNT_PHYSICAL_SPES, -1);
==> 16
no_phys_spes = spe_cpu_info_get(SPE_COUNT_PHYSICAL_SPES, 0);
==> 8
no_phys_spes = spe_cpu_info_get(SPE_COUNT_PHYSICAL_SPES, 1);
==> 8
no_usable_spes = spe_cpu_info_get(SPE_COUNT_USABLE_SPES, -1);
==> 13
no_usable_spes = spe_cpu_info_get(SPE_COUNT_USABLE_SPES, 0);
==> 7
no_usable_spes = spe_cpu_info_get(SPE_COUNT_USABLE_SPES, 1);
==> 6
```

Chapter 4. SPE program image handling

Before you can run an SPE context, you must load an SPE program into the SPE's local store. To do this, you use the function **spe_program_load**. The SPE program can either be an independent ELF image in a file or it can be embedded in the main thread executable in special sections. The first case requires that the SPE program image is loaded into memory by calling **spe_image_open**.

You can find information about SPE executables *Cell Broadband Engine Programming Handbook*, Version 1.0, chapter 14 ″Objects, Executables, and SPE Loading″.

SPE image functions

The following section describes the SPE program image functions.

spe_image_open

NAME

spe_image_open - Open an SPE ELF executable and map it into system memory.

SYNOPSIS

#include <libspe2.h>

spe_program_handle_t *spe_image_open (const char *filename)

Parameters

filename	Specifies the filename of an SPE ELF executable to be loaded and mapped into system memory.

DESCRIPTION

spe_open_image opens an SPE ELF executable indicated by *filename* and maps it into system memory. The result is a pointer to an SPE program handle which can then be used with **spe_program_load** to load this SPE main program into the local store of an SPE before running it with **spe_context_run**. The file containing the SPE executable must have execution access rights. SPE ELF objects loaded using this function are not shared with other applications and processes.

It can be more convenient to embed SPE ELF objects directly within the PPE executable using the linker and an ″embed_spu″ (or equivalent) tool (see toolchain documentation). In this case, SPE ELF objects are converted to PPE static or shared libraries with symbols, which point to the SPE ELF objects after these special libraries are loaded.

These libraries are then linked with the associated PPE code to provide a direct symbol reference to the SPE ELF object. The symbols in this scheme are equivalent to the address returned from the **spe_image_open** function and can be used as SPE program handles by **spe_program_load**. SPE ELF objects created using the embedding approach can be shared between processes.

RETURN VALUE

On success, a non-null handle to the mapped SPE ELF object is returned.

EXIT STATUS

On error, **NULL** is returned and *errno* is set to indicate the error.

Possible errors include:

EACCES	The calling process does not have the necessary permissions to access the specified file.
EFAULT	The filename parameter points to an address that was not contained in the calling process's address space.

other	A number of other errno values could be returned by the open(2), fstat(2), or mmap(2) system calls which may be utilized by the spe_image_open function.

SEE ALSO

spe_program_load(3); spe_context_run(3); spe_image_close(3)

spe_image_close

NAME

spe_image_close - Unmap and close an SPE ELF object.

SYNOPSIS

#include <libspe2.h>

int spe_image_close (spe_program_handle_t *program)

Parameters

program	A valid address of a mapped SPE program.

DESCRIPTION

Unmaps and closes an SPE ELF object that was previously opened and mapped using **spe_open_image**.

RETURN VALUE

On success, **0 (zero)** is returned.

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

EINVAL	The specified address of the SPE program is invalid.
other	A number of other *errno* values could be returned by the munmap(2) or close(2) system calls which may be utilized by the **spe_image_open** function.

SEE ALSO

spe_image_open(3)

spe_program_load

NAME

spe_program_load - Load an SPE main program.

SYNOPSIS

#include <libspe2.h>

int spe_program_load (spe_context_ptr_t spe, spe_program_handle_t *program)

Parameters

spe	A valid pointer to the SPE context for which an SPE program should be loaded.
program	A valid address of a mapped SPE program.

DESCRIPTION

Load an SPE main program that has been mapped to memory at the address pointed to by *program* into the local store of the SPE identified by the SPE context *spe*. This is mandatory before running the SPE context with **spe_context_run**.

RETURN VALUE

On success, **0** (zero) is returned.

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE context is not valid.
EINVAL	The specified address of the SPE program is not valid.
ENOEXEC	The program image is not correctly formatted.

SEE ALSO

spe_context_run(3); spe_image_close(3)

Chapter 5. SPE run control

After the application has created an SPE context and loaded an SPE program into its local store, it can call **spe_context_run** to run an SPE context. A thread which executes an SPE context is called an SPE thread.

The API function to run a context is a synchronous, blocking call from the perspective of the thread using it, that is, while an SPE program is executing, the associated SPE thread blocks and is usually put to ″sleep″ by the operating system. When the SPE program stops, either because it reaches its ″normal″ exit point, or a stop and signal instruction or an error condition, the **spe_context_run** function returns and the resulting return value specifies the exact condition under which the SPE program stopped.

Many applications need to use multiple SPEs concurrently. In this case, the application must create at least as many threads, by using standard methods of the operating system, as concurrent SPE contexts are required. Each of these threads may run a single SPE context at a time. If *N* concurrent SPE contexts are needed, it is, however, common to use *N*+1 threads; one main (application) thread that ″orchestrates″ the execution of *N* SPE threads.

In a multithreaded environment, it is often convenient to use an event mechanism for notification about certain events caused by the asynchronously running SPE threads. A specific event is used to indicate that an SPE context has stopped in the SPE thread. The function **spe_stop_info_read** allows the main thread to read the full information about why the SPE context stopped.

SPE run functions

The following section describes the SPE run functions.

spe_context_run

NAME

spe_context_run - Request execution of an SPE context.

SYNOPSIS

#include <libspe2.h>

int spe_context_run(spe_context_ptr_t spe, unsigned int *entry, unsigned int runflags, void *argp, void *envp, spe_stop_info_t *stopinfo)

Parameters

spe	A pointer to the SPE context that should be run.
entry	Input: The entry point, that is, the initial value of the SPU instruction pointer, at which the SPE program should start executing. If the value of entry is SPE_DEFAULT_ENTRY, the entry point for the SPU main program is obtained from the loaded SPE image. This is usually the local store address of the initialization function crt0 (see *Cell Broadband Engine Programming Handbook*, Objects, Executables, and SPE Loading). Output: The SPU instruction pointer at the moment the SPU stopped execution, that is, the local store address of the next instruction that would be have been executed. This parameter can be used, for example, to allow the SPE program to "pause" and request some action from the PPE thread, for example, performing an I/O operation. After this PPE-side action has been completed, you can continue the SPE program calling **spe_context_run** again without changing *entry*.
runflags	A bit mask that can be used to request certain specific behavior for the execution of the SPE context. If the value is 0, this indicates default behavior (see *Usage*).
argp	An (optional) pointer to application specific data, and is passed as the second parameter to the SPE program, (see *Note*).
envp	An (optional) pointer to environment specific data, and is passed as the third parameter to the SPE program, (see *Note*).
stopinfo	An (optional) pointer to a structure of type spe_stop_info_t (see *Usage*).

DESCRIPTION

Request execution of an SPE context. A SPE program must be loaded (using **spe_program_load**) before you can run the SPE context.

The thread calling **spe_context_run** blocks and waits until the SPE stops, due to normal termination of the SPE program, or an SPU stop and signal instruction, or an error condition. When **spe_context_run** returns, the calling thread must take appropriate actions depending on the application logic.

spe_context_run returns information about the termination of the SPE program in three ways. This allows applications to deal with termination conditions on various levels.

- First, the most common usage for many applications is covered by the return value of the function and the *errno* value being set appropriately.
- Second, the optional *stopinfo* structure provides detailed information about the termination condition in a structured way that allows applications more fine-grained error handling and enables implementation of special scenarios.
- Third, the *stopinfo* structure contains the field *spu_status* that contains the value of the CBEA SPU Status Register (SPU_Status) as specified in the *Cell Broadband Engine Architecture, Version 1*, section 8.5.2 upon termination of the SPE program. This can be very useful, especially in conjunction with the SPE_NO_CALLBACKS flag, for applications that run non-standard SPE programs and want to react to all possible conditions flexibly and not rely on predefined conventions.

RETURN VALUE

On success, **0** (zero) or a positive number is returned.

A return value of **0** (zero) indicates that the SPE program terminated normally by calling **exit()**. The actual exit value can be obtained from *stopinfo*.

A positive return value indicates that the SPE has stopped because the SPU issued a stop and signal instruction and the return value represents the 14-bit value set by that stop and signal instruction.

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE context is not valid.
EINVAL	The value passed for flags is not valid.
EIO	An SPE I/O error occurred, for example, a misaligned DMA. Details can be found in *stopinfo*.
EFAULT	Some other SPE runtime problem occurred. Details can be found in *stopinfo*.
EPERM	The SPE isolation system mechanism is corrupted. No isolated SPE program can be loaded and started.

OPTIONS

The following flags are accepted for the runflags parameter. Multiple flags can be combined using bit-wise OR.

SPE_RUN_USER_REGS
Specifies that the SPE setup registers r3, r4, and r5 are initialized with the 48 bytes pointed to by argp.

SPE_NO_CALLBACKS
Specifies that registered SPE library calls ("callbacks" from this library's view) should not run automatically if a callback is encountered. This also disables callbacks that are predefined in the library implementation. See *PPE-assisted library* calls for details.

spe_context_run returns as if the SPU would have issued a regular stop and signal instruction. The signal code is returned as part of *stopinfo*.

USAGE

stopinfo

When **spe_context_run** returns, it provides information about the exact conditions in which the SPE stopped program execution in the data structure pointed to by *stopinfo*. If *stopinfo* is NULL, this information is not returned by the call.

If *stopinfo* is a valid pointer, the structure is filled with all information available as to the reason why the SPE program stopped running. This information is important for some advanced programming patterns, or detailed error reporting, or both.

If *stopinfo* is **NULL**, no information beyond the return value (specified below) as to the reason and associated data why the SPE program stopped execution will be returned.

The data type spe_stop_info_t is defined as follows:

```
typedef struct spe_stop_info {
   unsigned int stop_reason;
   union {
     int spe_exit_code;
     int spe_signal_code;
     int spe_runtime_error;
     int spe_runtime_exception;
     int spe_runtime_fatal;
     int spe_callback_error;
     int spe_isolation_error;
     void *__reserved_ptr;
     unsigned long long __reserved_u64;
   } result;
   int spu_status;
} spe_stop_info_t;
```

The valid values for *stop_reason* are defined by the following constants:

SPE_EXIT	SPE program terminated calling exit(code) with code in the range 0..255. The code is saved in *spe_exit_code*.
SPE_STOP_AND_SIGNAL	SPE program stopped because SPU ran a stop and signal instruction. Further information in field *spe_signal_code*.
SPE_RUNTIME_ERROR	SPE program stopped because of a one of the reasons found in *spe_runtime_error*. **Note: (Linux)** The error SPE_SPU_INVALID_INSTR is reported as a Linux signal SIGILL if the SPE context was created without the flag SPE_EVENTS_ENABLE.

SPE_RUNTIME_EXCEPTION	SPE program stopped asynchronously because of a runtime exception (event) described in *spe_runtime_exception*. In this case, *spe_status* would be meaningless and is therefore set to -1. **Note: (Linux®)** This error situation can only be caught and reported by spe_context_run if the SPE context was created with the flag SPE_EVENTS_ENABLE indicating that event support is requested. Otherwise the Linux kernel generates a signal to indicate the runtime error.
SPE_RUNTIME_FATAL	The SPE program stopped for other reasons, usually fatal operating system errors such as insufficient resources. Further information in *spe_runtime_fatal*. In this case, *spe_status* would be meaningless and is therefore set to -1.
SPE_CALLBACK_ERROR	An SPE program tried to use unregistered library callback, or a library callback returned a non-zero exit value, which is provided in *spe_callback_error*.
SPE_ISOLATION_ERROR	The SPE isolation system mechanism has detected an error when attempting to load the isolated SPE program. The reason for the failure is captured in *spe_isolation_error*.

Depending on *stop_reason* more specific information is provided in the result field:

spe_exit_code	Exit code returned by the SPE program in the range 0..255. The convention for stop and signal usage by SPE programs is that 0x2000-0x20FF are exit events. 0x2100-0x21FF are callback events. 0x0 is an invalid instruction runtime error. Signal codes 0x0001-0x1FFF are user-defined signals. This convention determines the mapping to the respective fields in *stopinfo*.
spe_signal_code	Stop and signal code sent by the SPE program. The lower 14-bit of this field contain the signal number. The convention for stop and signal usage by SPE programs is that 0x2000-0x20FF are exit events. 0x2100-0x21FF are callback events. 0x0 is an invalid instruction runtime error. Signal codes 0x0001-0x1FFF are user-defined signals. This convention determines the mapping to the respective fields in *stopinfo*.
spe_runtime_error	SPE_SPU_HALT: SPU was stopped by halt. SPE_SPU_SINGLE_STEP: SPU is in single-step mode SPE_SPU_INVALID_INSTR: SPU has tried to execute an invalid instruction SPE_SPU_INVALID_CHANNEL: SPU has tried to access an invalid channel
spe_runtime_exception	SPE_DMA_ALIGNMENT: A DMA alignment error SPE_DMA_SEGMENTATION: A DMA segmentation error SPE_DMA_STORAGE: A DMA storage error SPE_INVALID_DMA: An invalid DMA error
spe_runtime_fatal	Contains the (implementation-dependent) *errno* as set by the underlying system call that failed.

spe_callback_error	Contains the return code from the failed library callback, or it is set to -1 in the case of unregistered library callback.
spe_isolation_error	Contains the implementation-dependent error code for the failed starting of an isolated SPE program.

The field *spu_status* contains the value of the architected "SPU Status Register (SPU_Status)" as defined in the *Cell Broadband Engine Architecture*, Version 1.0, section 8.5.2 at the point in time the SPU stopped execution. In some circumstances, for example, asynchronous errors such as DMA alignment errors, this value would be meaningless and therefore a value of -1 is returned to indicate that situation.

The content of *spu_status* is fully reflected in the *stop_reason* and subsequent field and is returned to allow low-level application their own, direct interpretation of *spu_status* directly following the CBE Architecture specification. Most applications do not need this field.

NOTES

Argument passing to SPE programs:

An application may pass arguments to an SPE program by using *argp*, *envp*, and the SPE_RUN_USER_REGS flag above. The SPE registers r3, r4, and r5 are initialized according to the following scheme:

If SPE_RUN_USER_REGS is not set, then the registers are initialized as follows:

- r3 spe - the address of the SPE context being run
- r4 argp - usually a pointer to argv of the main program
- r5 envp - usually the environment pointer of the main program

All 32-bit or 64-bit pointers are put into the correct preferred slots for the 128-bit SPE registers.

If SPE_RUN_USER_REGS is set, then the registers are initialized with a copy of an (uninterpreted) 48-byte user data field pointed to by *argp*. *envp* is ignored in this case.

SEE ALSO

spe_context_create(3); spe_program_load(3)

spe_stop_info_read

NAME

spe_stop_info_read - Read information about the conditions in which the SPE stopped.

SYNOPSIS

#include <libspe2.h>

int spe_stop_info_read (spe_context_ptr_t spe, spe_stop_info_t *stopinfo)

Parameters

spe	A pointer to the SPE context for which stop information is requested.
stopinfo	A pointer to a structure of type *spe_stop_info_t* (specified in **spe_context_run**). The structure is filled with all information available as to the reason why the SPE program stopped execution.

DESCRIPTION

Reads information about the conditions in which the SPE identified by *spe* stopped.

This function is intended for use when the **spe_context_run** call returns, that is the SPE stops, in the SPE thread.

This is a non-blocking call. If the information does not exist, for example, because the context has never been run, or has already been read, for example, by another thread, the function returns an error with *errno* set to EAGAIN.

This function requires that the SPE context *spe* has been created with event support, that is, the SPE_EVENTS_ENABLE flag has been set. Otherwise, it returns an error ENOTSUP.

RETURN VALUE

On success, **0** (zero) is returned.

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE context is invalid.
EAGAIN	No data available.
ENOTSUP	Event processing is not enabled for this SPE context.
EINVAL	The specified pointer to an spe_stop_info_t structure is invalid.

SEE ALSO

spe_context_run(3)

Chapter 6. SPE event handling

In a multithreaded environment, it is often convenient to use an event mechanism for asynchronous notification. A common use is that the main thread sets up an event handler to receive notification about certain events caused by the asynchronously running SPE threads, see **spe_event_handler_create** and **spe_event_handler_register**. It then uses an event loop to wait for events, using **spe_event_wait**, and performs appropriate actions in response.

The library supports events to indicate that an SPE has stopped execution, mailbox messages have been written or read by an SPE, or PPE-initiated DMA operations have completed. In order to obtain details associated with the event, the application has to perform a separate action, for example, call **spe_stop_info_read** to obtain the full information on the stop reason for an SPE context, call **spe_out_intr_mbox_read** to actually read the message from the SPE mailbox, or call **spe_mfcio_tag_status_read** to know which tag groups completed.

SPE event functions

The following section describes the SPE event functions.

spe_event_handler_create

NAME

spe_event_handler_create - Create a SPE event handler and return a pointer to it.

SYNOPSIS

#include <libspe2.h>

spe_event_handler_ptr_t spe_event_handler_create(void)

Parameters

void	*none*

RETURN VALUE

On success, a pointer to an SPE event handler is returned.

EXIT STATUS

On error, **NULL** is returned and *errno* is set to indicate the error.

Possible errors include:

ENOMEM	The SPE event handler could not be allocated due to lack of system resources.
EFAULT	A runtime error of the underlying OS service occurred.

SEE ALSO

spe_event_handler_destroy(3)

spe_event_handler_destroy

NAME

spe_event_handler_destroy - Destroy a SPE event handler.

SYNOPSIS

#include <libspe2.h>

int spe_event_handler_destroy (spe_event_handler_ptr_t evhandler);

Parameters

evhandler	A pointer to the SPE event handler to be destroyed.

RETURN VALUE

On success, **0** (zero) is returned.

EXIT STATUS

On error, **-1** is returned and *errno* ise set to indicate the error.

Possible errors include:

ESRCH	The specified SPE event handler is invalid.
EAGAIN	The specified SPE event handler cannot be destroyed at this time since it is in use, that is an spe_event_wait call is currently active waiting on this handler.
EFAULT	A runtime error of the underlying OS service occurred.

SEE ALSO

spe_event_handler_create(3); spe_event_wait(3)

spe_event_handler_deregister

NAME

spe_event_handler_deregister - Deregister the application's interest in SPE events.

SYNOPSIS

#include <libspe2.h>

int spe_event_handler_deregister(spe_event_handler_ptr_t evhandler, spe_event_unit_t *event);

Parameters

evhandler	A pointer to the SPE event handler.
event	A pointer to an SPE event structure.

DESCRIPTION

Deregisters the application's interest in SPE events of the specified nature as defined in the *event* structure.

It is no error to deregister interest in events that have not been registered before. Therefore, all events on a specific *evhandler* and *spe* can be always deregistered with a single function call using the SPE_EVENT_ALL_EVENTS mask.

This function requires that the SPE context *spe* in *event* has been created with event support, that is, the SPE_EVENTS_ENABLE flag has been set. Otherwise, it returns an error ENOTSUP.

RETURN VALUE

On success, **0** (zero) is returned.

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE event handler is invalid.
EINVAL	The specified pointer to an SPE event structure or the SPE context specified in the SPE event structure is invalid.
ENOTSUP	At least one of the requested events specified in events is not supported or invalid or the SPE context does not support events.
EFAULT	A runtime error of the underlying OS service occurred.

USAGE

The *spe_event_unit_t* data structure and its usage are specified in **spe_event_handler_register**. A single call to this interface can deregister multiple events at the same time. The field *spe* in *event* is a pointer to an SPE context for which the events have to be deregistered. The field *data* will be ignored by this call.

SEE ALSO

spe_event_handler_register(3); spe_event_wait(3); spe_out_intr_mbox_read(3); spe_in_mbox_write(3); spe_mfcio_tag_status_read(3); spe_stop_info_read(3)

spe_event_handler_register

NAME

spe_event_handler_register - Register the application's interest in SPE events.

SYNOPSIS

#include <libspe2.h>

int spe_event_handler_register(spe_event_handler_ptr_t evhandler, spe_event_unit_t *event);

Parameters

evhandler	A pointer to the SPE event handler.
event	A pointer to an SPE event structure.

DESCRIPTION

Registers the application's interest in SPE events as defined in the event structure.

This function requires that the SPE context *spe* in *event* has been created with event support, that is, the SPE_EVENTS_ENABLE flag has been set. Otherwise, it returns an error ENOTSUP.

RETURN VALUE

On success, **0** (zero) is returned.

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE event handler is invalid.
EINVAL	The specified pointer to an SPE event structure or the SPE context specified in the SPE event structure is invalid
ENOTSUP	At least one of the requested events specified in *events* is not supported or invalid or the SPE context does not support events.
EFAULT	A runtime error of the underlying OS service occurred.

USAGE

The data structure spe_event_unit_t is defined as follows:

```
typedef struct spe_event_unit {
   unsigned int events;
   spe_context_ptr_t spe;
   spe_event_data_t data;
}  spe_event_unit_t;
```

The field *events* specifies a bitmask to request certain SPE events to be delivered to the application. Multiple events can be requested at once by using bit-wise OR.

The following events are supported:

SPE_EVENT_OUT_INTR_MBOX	Data is available in the SPU outbound interrupting mailbox. This event is generated, when the SPU has written at least one entry to the SPU outbound interrupting mailbox (see **spe_out_intr_mbox_read**).
SPE_EVENT_IN_MBOX	Data can now be written to the SPU inbound mailbox. This event is generated when the inbound mailbox is not full and signals that at least one message can be successfully written **spe_in_mbox_write**).
SPE_EVENT_TAG_GROUP	An SPU event tag group signaled completion (see **spe_mfcio_tag_status_read**).
SPE_EVENT_SPE_STOPPED	Program execution on the SPE has stopped. (see **spe_stop_info_read**).
SPE_EVENT_ALL_EVENTS	Interest in all defined SPE events, this corresponds to a bit-wise OR of all flags above.

The field *spe* is a pointer to an SPE context for which the events have to be registered.

The structure *spe_event_unit* contains a field *data* of type spe_event_data that is intended to hold user data. The value of this field will be returned to the application by **spe_event_wait** unmodified, whenever an event as specified here occurs.

```
typedef union spe_event_data {
     void *ptr;
     unsigned int u32;
     unsigned long long u64;
 }   spe_event_data_t;
```

SEE ALSO

spe_event_handler_deregister(3); spe_event_wait(3); spe_out_intr_mbox_read(3); spe_in_mbox_write(3); spe_mfcio_tag_status_read(3); spe_stop_info_read(3)

spe_event_wait

NAME

spe_event_wait - Wait for SPE events.

SYNOPSIS

#include <libspe2.h>

int spe_event_wait(spe_event_handler_ptr_t evhandler, spe_event_unit_t *events, int max_events, int timeout);

evhandler	A valid pointer to the SPE event handler.
events	The pointer to the memory area where the events will be stored. The *'events'* member will contain the event bit field indicating the actual event received, and the *'spe'* member will contain pointer to the SPE context that generated the event. For the specification of *spe_event_unit_t*, see **spe_event_handler_register**.
max_events	Maximum number of 'events' to receive. The call will return if at least one event has been received or if it times out.
timeout	Timeout in milliseconds. -1 means 'infinite'. 0 means that the call should not wait but return immediately with as many events as are currently available up to a maximum of *max_events*.

RETURN VALUE

On success, the number of SPE events received. If **0** (zero) is returned, no SPE event was received because the request timed out.

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE event handler is invalid.
EINVAL	Error in parameters.
EFAULT	A runtime error of the underlying OS service occurred.

SEE ALSO

spe_event_handler_register(3); spe_event_handler_deregister(3); spe_out_intr_mbox_read(3); spe_out_intr_mbox_status(3); spe_in_mbox_write(3); spe_mfcio_tag_status_read(3); spe_stop_info_read(3)

Chapter 7. SPE MFC problem state facilities

SPE MFC proxy command functions

This set of functions provides PPE-initiated DMA functionality (see *Cell Broadband Engine Architecture*, MFC Proxy Commands) through the usage of the SPE MFC Proxy Command Issue facility. Main threads can use these functions to move data to and from an SPE local store area.

Note: The naming of the commands is based on a SPE centric view, for example, "put" means a transfer from the SPE local store to an effective address valid in the main thread.

spe_mfcio_put

NAME

spe_mfcio_put - Place a put DMA command on the proxy command queue of the SPE context.

SYNOPSIS

#include <libspe2.h>

int spe_mfcio_put (spe_context_ptr_t spe, unsigned int lsa, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)

Parameters

spe	Specifies the SPE context of the proxy command queue in which the put command is to be placed.
lsa	Specifies the starting local store source address.
ea	Specifies the starting effective address destination address.
size	Specifies the size, in bytes, to be transferred.
tag	Specifies the tag ID used to identify the DMA command. The range for valid tag IDs is 0:31. Based on the needs of the underlying operating system implementations of this API can restrict the range. **Note: (Linux)** In the Linux implementation of libspe2, the range for the valid tags is 0:15. See *Cell Broadband Engine Architecture*, MFC Command Tag Register.
tid	Specifies the transfer class identifier of the DMA command.
rid	Specifies the replacement class identifier of the DMA command.

DESCRIPTION

Places a put DMA command on the proxy command queue of the SPE context specified by *spe*

The **spe_mfcio_put** command transfers *size* bytes of data starting at the local store address specified by *lsa* to the effective address specified by *ea*. The DMA is identified by the tag ID specified by *tag* and performed according transfer class and replacement class specified by *tid* and *rid* respectively.

The caller of this function must ensure that the address alignment and transfer size is in accordance with the limitation and restrictions of the Cell Broadband Engine Architecture.

RETURN VALUE

On success, **0** (zero) is returned.

EXIT STATUS

On error, **-1** is returned and *errno* are set to indicate the error.

Possible errors include:

ESRCH	The specified SPE context is invalid.

spe_mfcio_putb

NAME

spe_mfcio_putb - Place a put DMA command with a barrier on the proxy command queue of the SPE context.

SYNOPSIS

#include <libspe2.h>

int spe_mfcio_putb (spe_context_ptr_t spe, unsigned int lsa, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)

Parameters

spe	Specifies the SPE context of the proxy command queue in which the put command is to be placed.
lsa	Specifies the starting local store source address.
ea	Specifies the starting effective address destination address.
size	Specifies the size, in bytes, to be transferred.
tag	Specifies the tag ID used to identify the DMA command. The range for valid tag IDs is 0:31. Based on the needs of the underlying operating system implementations of this API can restrict the range. **Note: (Linux)** In the Linux implementation of libspe2, the range for the valid tags is 0:15. See *Cell Broadband Engine Architecture*, MFC Command Tag Register.
tid	Specifies the transfer class identifier of the DMA command.
rid	Specifies the replacement class identifier of the DMA command.

DESCRIPTION

Place a put DMA command with a barrier on the proxy command queue of the SPE context specified by *spe*.

The **spe_mfcio_putb** function is identical to **spe_mfcio_put** except that it places a **putb** (put with barrier) DMA command on the proxy command queue. The barrier form ensures that this command and all sequential commands with the same tag identifier as this command are locally ordered with respect to all previously issued commands with the same tag group and command queue.

The caller of this function must ensure that the address alignment and transfer size is in accordance with the limitation and restrictions of the Cell Broadband Engine Architecture.

RETURN VALUE

On success, **0** (zero) is returned.

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE context is invalid.

spe_mfcio_putf

NAME

spe_mfcio_putf - Place a put DMA command with a fence on the proxy command queue of the SPE context.

SYNOPSIS

#include <libspe2.h>

int spe_mfcio_putf (spe_context_ptr_t spe, unsigned int lsa, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)

Parameters

spe	Specifies the SPE context of the proxy command queue which the put command is to be placed.
lsa	Specifies the starting local store source address.
ea	Specifies the starting effective address destination address.
size	Specifies the size, in bytes, to be transferred.
tag	Specifies the tag ID used to identify the DMA command. The range for valid tag IDs is 0:31. Based on the needs of the underlying operating system implementations of this API can restrict the range. **Note: (Linux)** In the Linux implementation of libspe2, the range for the valid tags is 0:15. See *Cell Broadband Engine Architecture*, MFC Command Tag Register.
tid	Specifies the transfer class identifier of the DMA command.
rid	Specifies the replacement class identifier of the DMA command.

DESCRIPTION

Places a put DMA command with a fence on the proxy command queue of the SPE context specified by *spe*.

The **spe_mfcio_putf** function is identical to **spe_mfcio_put** except that it places a **putf** (put with fence) DMA command on the proxy command queue. The fence form ensures that this command is locally ordered with respect to all previously issued commands with the same tag group and command queue.

The caller of this function must ensure that the address alignment and transfer size is in accordance with the limitation and restrictions of the Cell Broadband Engine Architecture.

RETURN VALUE

On success, **0** (zero) is returned.

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE context is invalid.

spe_mfcio_get

NAME

spe_mfcio_get - Place a get DMA command on the proxy command queue of the SPE context.

SYNOPSIS

include <libspe2.h>

int spe_mfcio_get (spe_context_ptr_t spe, unsigned int lsa, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)

Parameters

spe	Specifies the SPE context of the proxy command queue into which the get command is to be placed.
lsa	Specifies the starting local store destination address.
ea	Specifies the starting effective address source address.
size	Specifies the size, in bytes, to be transferred.
tag	Specifies the tag ID used to identify the DMA command. The range for valid tag IDs is 0:31. Based on the needs of the underlying operating system implementations of this API can restrict the range. **Note: (Linux)** In the Linux implementation of libspe2, the range for the valid tags is 0:15. See *Cell Broadband Engine Architecture*, MFC Command Tag Register.
tid	Specifies the transfer class identifier of the DMA command.
rid	Specifies the replacement class identifier of the DMA command.

DESCRIPTION

Places a get DMA command on the proxy command queue of the SPE context specified by *spe*

The spe_mfcio_get command transfers *size* bytes of data starting at the effective address specified by *ea* to the local store address specified by *lsa*. The DMA is identified by the tag id specified by *tag* and performed according transfer class and replacement class specified by *tid* and *rid* respectively.

The caller of this function must ensure that the address alignment and transfer size is in accordance with the limitation and restrictions of the Cell Broadband Engine Architecture.

RETURN VALUE

On success, **0** (zero) is returned.

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE context is invalid.

spe_mfcio_getb

NAME

spe_mfcio_getb - Place a get with barrier DMA command on the proxy command queue of the SPE contexts.

SYNOPSIS

include <libspe2.h>

int spe_mfcio_getb (spe_context_ptr_t spe, unsigned int lsa, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)

Parameters

spe	Specifies the SPE context of the proxy command queue into which the get command is to be placed.
lsa	Specifies the starting local store destination address.
ea	Specifies the starting effective address source address.
size	Specifies the size, in bytes, to be transferred.
tag	Specifies the tag ID used to identify the DMA command. The range for valid tag IDs is 0:31. Based on the needs of the underlying operating system implementations of this API can restrict the range. **Note: (Linux)** In the Linux implementation of libspe2, the range for the valid tags is 0:15. See *Cell Broadband Engine Architecture*, MFC Command Tag Register.
tid	Specifies the transfer class identifier of the DMA command.
rid	Specifies the replacement class identifier of the DMA command.

DESCRIPTION

Places a get with barrier DMA command on the proxy command queue of the SPE context specified by *spe*.

The **spe_mfcio_getb** command transfers *size* bytes of data starting at the effective address specified by *ea* to the local store address specified by *lsa*. The DMA is identified by the tag id specified by *tag* and performed according transfer class and replacement class specified by *tid* and *rid* respectively.

The **spe_mfcio_getb** function is identical to **spe_mfcio_get** except that it places a **getb** (get with barrier) DMA command on the proxy command queue. The barrier form ensures that this command and all sequential commands with the same tag identifier are locally ordered with respect to all previously issued commands with the same tag group and command queue.

The caller of this function must ensure that the address alignment and transfer size is in accordance with the limitation and restrictions of the Cell Broadband Engine Architecture.

RETURN VALUE

On success, **0** (zero) is returned.

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE context is invalid.

spe_mfcio_getf

NAME

spe_mfcio_getf - Place a get with fence DMA command on the proxy command queue of the SPE context .

SYNOPSIS

include <libspe2.h>

int spe_mfcio_getf (spe_context_ptr_t spe, unsigned int lsa, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)

Parameters

spe	Specifies the SPE context of the proxy command queue into which the get command is to be placed.
lsa	Specifies the starting local store destination address.
ea	Specifies the starting effective address source address.
size	Specifies the size, in bytes, to be transferred.
tag	Specifies the tag ID used to identify the DMA command. The range for valid tag IDs is 0:31. Based on the needs of the underlying operating system implementations of this API can restrict the range. **Note: (Linux)** In the Linux implementation of libspe2, the range for the valid tags is 0:15. See *Cell Broadband Engine Architecture*, MFC Command Tag Register.
tid	Specifies the transfer class identifier of the DMA command.
rid	Specifies the replacement class identifier of the DMA command.

DESCRIPTION

Places a get with fence DMA command on the proxy command queue of the SPE context specified by *spe*

The **spe_mfcio_getf** command transfers *size* bytes of data starting at the effective address specified by *ea* to the local store address specified by *lsa*. The DMA is identified by the tag id specified by *tag* and performed according transfer class and replacement class specified by *tid* and *rid* respectively.

The **spe_mfcio_getf** function is identical to **spe_mfcio_get** except that it places a **getf** (get with fence) DMA command on the proxy command queue. The barrier form ensures that this command and all sequence commands with the same tag identifier as this command are locally ordered with respect to all previously issued commands with the same tag group and command queue.

The caller of this function must ensure that the address alignments and transfer size is in accordance with the limitation and restrictions of the Cell Broadband Engine Architecture.

RETURN VALUE

On success, **0** (zero) is returned.

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE context is invalid.

SPE MFC multi-source synchronization functions

The MFC multi-source synchronization functions provide facilities for achieving cumulative ordering across the local storage and main storage address domains for a specified SPE.

To achieve cumulative ordering, first call **spe_mssync_start** to cause the MFC to start tracking all requested, but not completed, transfers targeted at the specified SPE. When all the transfers that are being tracked are completed, **spe_mssync_status** returns a value of `0`.

See the *Cell Broadband Engine Architecture* for more information about multi-source synchronization facility.

spe_mssync_start

NAME

spe_mssync_start - Start multi-source synchronization.

SYNOPSIS

include <libspe2.h>

int spe_mssync_start (spe_context_ptr_t spe)

Parameters

spe	Specifies the SPE for which MFC transfers are to be synchronized.

DESCRIPTION

Start tracking all pending transfers targeted at the specified SPE to facilitate cumulative ordering of transfers across the local storage and main storage address domains. Cumulative order is ensured when a subsequent call to **spe_mssync_status** returns a value of `0`.

RETURN VALUE

On success, **0** (zero) is returned.

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE context is invalid.

SEE ALSO

spe_mssync_status(3)

spe_mssync_status

NAME

spe_mssync_status - Fetch the status of the multi-source synchronization.

SYNOPSIS

include <libspe2.h>

int spe_mssync_status (spe_context_ptr_t spe)

Parameters

spe	Specifies the SPE for which the MFC transfers are to be synchronized.

DESCRIPTION

Fetch the status of the previously requested multi-source synchronization. A synchronization request is initiated by calling **spe_mssync_start**.

RETURN VALUE

On success, **0** (zero) is returned.

A value of 0 indicates that all transfers targeting the SPE and received before the last **spe_mssync_start()** are complete.

A value of 1 indicates that all transfers targeting the SPE and received before the last **spe_mssync_start()** are not complete.

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE context is invalid.

SEE ALSO

spe_mssync_start(3)

SPE MFC proxy tag-group completion functions

The following describes the SPE MFC proxy tag-group completion functions.

spe_mfcio_tag_status_read

NAME

spe_mfcio_tag_status_read - Check the completion of DMA requests.

SYNOPSIS

#include <libspe2.h>

int spe_mfcio_tag_status_read(spe_context_ptr_t spe, unsigned int mask, unsigned int behavior, unsigned int *tag_status)

Parameters

spe	Specifies the SPE context for which DMA completion status is to be checked.
mask	Specifies which DMA requests should be taken into account.
behavior	Specifies the behavior of the operation.
tag_status	Result: the current tag status for tags specified by *mask* is returned.

DESCRIPTION

The **spe_mfcio_tag_status_read** function is used to check the completion of DMA requests. The optional *mas*k parameter is used to restrict the check to specific tag groups. A *mask* of value '0' indicates that all current DMA requests should be taken into account. The *behavior* field specifies completion of any of the members of the specified tag groups, or completion of all members of the specified tag groups.

The non-blocking reading of the tag status by specifying SPE_TAG_IMMEDIATE is especially advantageous when combining with SPE event handling. Note that after receiving a tag group completion event, the tag status has to be read *before* another DMA is started on the same SPE.

RETURN VALUE

On success, **0** (zero) is returned.

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE context is invalid.
ENOTSUP	The usage of a non-zero *mask* parameter is not supported by this implementation of the library or underlying OS.
EINVAL	The specified behavior or the specified pointer to a space for the result is invalid.

OPTIONS

The value of the *behavior* parameter can be one of:

Value	Description
SPE_TAG_ALL	The function blocks execution until all DMA commands in the tag groups enabled by the mask parameter have no outstanding DMAs in the proxy command queue of the SPE context specified by spe. The masked tag status is returned.
SPE_TAG_ANY	The function blocks execution until any DMA commands in the tag groups enabled by the *mask* parameter have no outstanding DMAs in the proxy command queue of the SPE context specified by spe. The masked tag status is returned.
SPE_TAG_IMMEDIATE	The function returns the tag status for the tag groups specified by the *mask* parameter for the proxy command queue of the SPE context specified by the spe.

The value of the *mask* parameter can be one of:

Value	Description
0 (zero)	Indicates that all current DMA requests should be taken into account. This takes into account only those DMAs started using libspe library calls, because the library and operating system have no way to know about DMA initiated by applications using direct problem state access.
A non-zero value	A non-zero value must be specified according to the *Cell Broadband Engine Architecture*, Version 1.0, section 8.4.3.

SEE ALSO

spe_mfcio_get(3); spe_mfcio_getb(3); spe_mfcio_getf(3); spe_mfcio_put(3); spe_mfcio_putb(3); spe_mfcio_putf(3)

SPE mailbox functions

This set of functions allows a main thread to communicate with an SPE through its mailbox facility.

The naming of the mailboxes is based on a SPE centric view, for example, ″out_mbox″ is the outbound mailbox for the SPE, and the corresponding library function **spe_out_mbox_read** is used to read the mailbox message from the main thread.

spe_out_mbox_read

NAME

spe_out_mbox_read - read up to *count* available messages from the SPE outbound mailbox.

SYNOPSIS

#include <libspe2.h>

int spe_out_mbox_read (spe_context_ptr_t spe, unsigned int *mbox_data, int count)

Parameters

spe	Specifies the SPE context of the SPU outbound mailbox to be read.
mbox_data	A pointer to an array of *count* unsigned integers of size to store the 32-bit mailbox messages read by the call.
count	The maximum number of mailbox entries to be read by this call.

DESCRIPTION

This function reads up to *count* available messages from the SPE outbound mailbox for the SPE context *spe*. This is a non-blocking function call. If less than *count* mailbox entries are available, only those will be read.

spe_out_mbox_status can be called to ensure that data is available prior to reading the outbound mailbox.

RETURN VALUE

>0 the number of 32-bit mailbox messages read

0 (zero) no data read

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE context is invalid.
EIO	The I/O error occurred.
EINVAL	The specified pointer to the mailbox message or the specified maximum number of mailbox entries is invalid.

SEE ALSO

spe_out_mbox_status(3)

spe_out_mbox_status

NAME

spe_out_mbox_status - Fetch the status of the SPU outbound mailbox.

SYNOPSIS

#include <libspe2.h>

int spe_out_mbox_status (spe_context_ptr_t spe)

Parameters

spe	Specifies the SPE context of the SPU outbound mailbox to be read.

DESCRIPTION

The **spe_out_mbox_status** function fetches the status of the SPU outbound mailbox for the SPE context specified by the *spe* parameter.

RETURN VALUE

>0 the number of 32-bit mailbox messages available for read

0 (zero) no data available (the mailbox is empty)

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE context is invalid.
EIO	The I/O error occurred.

SEE ALSO

spe_out_mbox_read(3)

spe_in_mbox_write

NAME

spe_in_mbox_write - Write up to *count* messages to the SPE inbound mailbox.

SYNOPSIS

#include <libspe2.h>

int spe_in_mbox_write (spe_context_ptr_t spe, unsigned int *mbox_data, int count, unsigned int behavior)

Parameters

spe	Specifies the SPE context of the SPU inbound mailbox to be written.
mbox_data	A pointer to an array of *count* unsigned integers containing the 32-bit mailbox messages to be written by the call.
count	The maximum number of mailbox entries to be written by this call.
behavior	Specifies whether the call should block until mailbox messages are written.

DESCRIPTION

Write up to *count* messages to the SPE inbound mailbox for the SPE context *spe*. This call may be blocking or non-blocking, depending on *behavior*.

The blocking version of this call is useful to send a sequence of mailbox messages to an SPE program, which do not require further synchronization. The non-blocking version is advantageous when SPE events are used for synchronization in multi-threaded applications.

spe_in_mbox_status can be called to ensure that data can be written before writing the SPU inbound mailbox.

RETURN VALUE

>0 the number of 32-bit mailbox messages written

0 (zero) no mailbox message could be written

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE context is invalid.
EIO	The I/O error occurred.
EINVAL	The specified pointer to the mailbox message, the specified maximum number of mailbox entries, or the specified behavior is invalid.

OPTIONS

Possible values for *behavior* are:

Value	Description
SPE_MBOX_ALL_BLOCKING	The call blocks until all *count* mailbox messages have been written.
SPE_MBOX_ANY_BLOCKING	The call blocks until at least one mailbox message has been written.
SPE_MBOX_ANY_NONBLOCKING	The call writes as many mailbox messages as possible up to a maximum of *count* without blocking.

SEE ALSO

spe_in_mbox_status(3)

spe_in_mbox_status

NAME

spe_in_mbox_status - Fetch the status of the SPU inbound mailbox for the SPE context.

SYNOPSIS

#include <libspe2.h>

int spe_in_mbox_status (spe_context_ptr_t spe)

Parameters

spe	Specifies the SPE context of the SPU outbound mailbox to be read.

DESCRIPTION

The **spe_in_mbox_status** function fetches the status of the SPU inbound mailbox for the SPE context specified by the *spe* parameter.

RETURN VALUE

>0 the number of 32-bit mailbox messages that can be written

0 (zero) no data can be written (mailbox full)

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE context is invalid.
EIO	The I/O error occurred.

SEE ALSO

spe_in_mbox_write(3)

spe_out_intr_mbox_read

NAME

spe_out_intr_mbox_read - Read up to *count* messages from the SPE outbound interrupting mailbox.

SYNOPSIS

#include <libspe2.h>

int spe_out_intr_mbox_read (spe_context_ptr_t spe, unsigned int *mbox_data, int count, unsigned int behavior)

Parameters

spe	Specifies the SPE context of the SPU inbound mailbox to be written.
mbox_data	A pointer to an array of *count* unsigned integers holding the 32-bit mailbox messages to be written by the call.
count	The maximum number of mailbox entries to be read by this call.
behavior	Specifies whether the call should block until completion.

DESCRIPTION

This function reads up to *count* messages from the SPE outbound interrupting mailbox for the SPE context *spe*. This call may be blocking or non-blocking, depending on *behavior*.

The blocking version of this call is particularly useful to receive a sequence of mailbox messages from an SPE program without further need for synchronization. The non-blocking version may be advantageous when using SPE events for synchronization in a multi-threaded application.

spe_out_intr_mbox_status can be called to ensure that data can be written prior to writing the SPU outbound interrupting mailbox.

RETURN VALUE

>0 the number of 32-bit mailbox messages read

0 (zero) no mailbox message could be read

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE context is invalid.
EIO	The I/O error occurred.

EINVAL	The specified pointer to the mailbox message, the specified maximum number of mailbox entries or the specified behavior is invalid.

OPTIONS

Possible values for *behavior* are:

Value	Description
SPE_MBOX_ALL_BLOCKING	The call blocks until all *count* mailbox messages have been read.
SPE_MBOX_ANY_BLOCKING	The call blocks until at least one mailbox message has been read.
SPE_MBOX_ANY_NONBLOCKING	The call reads as many mailbox messages as possible up to a maximum of *count* without blocking.

SEE ALSO

spe_out_intr_mbox_status(3)

spe_out_intr_mbox_status

NAME

spe_out_intr_mbox_status - Fetch the status of the SPU outbound interrupt mailbox.

SYNOPSIS

#include <libspe2.h>

int spe_out_intr_mbox_status (spe_context_ptr_t spe)

Parameters

spe	Specifies the SPE context for which the SPU outbound mailbox has to be read.

DESCRIPTION

The **spe_out_intr_mbox_status** function fetches the status of the SPU outbound interrupt mailbox for the SPE context specified by the *spe* parameter.

RETURN VALUE

>0 the number of 32-bit mailbox messages available for read

0 (zero) no data available (mailbox is empty)

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE context is invalid.
EIO	The I/O error occurred.

SEE ALSO

spe_out_intr_mbox_read(3)

SPE SPU signal notification functions

The following describe the SPE SPU signal notification functions.

spe_signal_write

NAME

spe_signal_write - Write *data* to the signal notification register specified by *signal_reg* for the SPE context specified by the *spe* parameter

SYNOPSIS

#include <libspe2.h>

int spe_signal_write (spe_context_ptr_t spe, unsigned int signal_reg, unsigned int data)

Parameters

spe	Specifies the SPE context of the signal register to be written to.
signal_reg	Specifies the signal notification register to be written.
data	The 32-bit data to be written to the specified signal notification register.

RETURN VALUE

On success, **0** (zero) is returned.

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE context is invalid.
EIO	An I/O error occurred.
EINVAL	The specified signal notification register is invalid.

OPTIONS

Valid signal notification registers for *signal_reg* are:

Flag	Description
SPE_SIG_NOTIFY_REG_1	SPE signal notification register 1
SPE_SIG_NOTIFY_REG_2	SPE signal notification register 2

Chapter 8. Direct SPE access for applications

This section describes how applications can access directly an SPE's local store memory and the various problem state registers.

The function **spe_ls_area_get** maps the local store of an SPE to the thread's address space. You can then access it like regular system memory. This is not recommended for regular use because DMA operations to and from local store are generally more efficient.

A more common use of the local store mapping is to communicate the effective address of one SPE's local store to another SPE, which allows SPEs to use DMA operations to directly transfer data to and from another local store. This mode of data transfer is very efficient, because the DMA transfers go directly from SPE to SPE, and not through system memory.

The function **spe_ps_area_get** maps an area of an SPE's problem state registers to the thread's address space. The problem state pointer can be used to directly access problem state features without using library system calls.

Problem state features include multi-source synchronization, proxy DMAs, mailboxes, and signal notifiers. These pointers, along with local store pointers (see **spe_ls_area_get**), can also be used to perform and control SPE to SPE communications through mailboxes, DMAs and signal notification.

When you use direct problem state access, you must ensure that applications serialize multiple problem state operations appropriately. Also, when you use both library and direct problem state operations, these must be properly serialized with respect to each other. Otherwise, unexpected behavior, application errors, or both can occur.

Note: (Linux) If you stop a running SPU by writing to SPE_RunCntrl, this does not ensure that the Linux kernel (scheduler) is informed allowing it to reclaim the SPE resources.

Direct access functions

The following section describes the direct access functions.

spe_ls_area_get

NAME

spe_ls_area_get - Map the local store of the SPE context.

SYNOPSIS

#include <libspe2.h>

void * spe_ls_area_get (spe_context_ptr_t spe)

Parameters

spe	Specifies the SPE context

DESCRIPTION

Maps the local store of the SPE context specified by *spe* to the thread's address space and returns a pointer to the start of the memory mapped local store area. The size of the local store area can be obtained by using the function **spe_ls_size_get**.

RETURN VALUE

On success, a pointer to the start of the memory mapped local store is returned.

EXIT STATUS

On error, **NULL** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE context is invalid.
ENOSYS	Access to the local store of an SPE thread is not supported by the operating system.

spe_ls_size_get

NAME

spe_ls_size_get - Obtain the size of the SPE local store in number of bytes.

SYNOPSIS

#include <libspe2.h>

int spe_ls_size_get (spe_context_ptr_t spe)

Parameters

spe	Specifies the SPE context

DESCRIPTION

The Cell Broadband Engine Architecture does not specify a fixed size for the SPE local store. Applications that are intended to be portable across different implementations of the CBEA should obtain the actual value through this call.

RETURN VALUE

On success, the SPE local store size (in bytes) is returned.

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified address of the SPE program is invalid.

SEE ALSO

spe_image_open(3)

spe_ps_area_get

NAME

spe_ps_area_get - Map the problem state area specified by *ps_area* of the SPE context specified by *spe* to the thread's address space.

SYNOPSIS

#include <libspe2.h>

void * spe_ps_area_get (spe_context_ptr_t spe, enum ps_area area)

Parameters	
spe	The identifier of a specific SPE context.
ps_area	The problem state area pointer to map.

DESCRIPTION

In order to obtain a problem state area pointer the specified SPE context must have been created with the SPE_MAP_PS flag set.

RETURN VALUE

On success, a pointer to the requested problem state area is returned.

EXIT STATUS

On error, **NULL** is returned and *errno* is set to indicate the error.

Possible errors include:

ESRCH	The specified SPE context is invalid.
EACCES	Permission for direct access to the specified problem state area is denied or the SPE context was not created with memory-mapped problem state access.
EINVAL	The specified problem state area is invalid.>
ENOSYS	Access to the specified problem area for the specified SPE context is not supported by the operating system.

OPTIONS

The following are possible problem state values for the parameter *ps_area*:

Problem state value	Description
SPE_MSSYNC_AREA	Return a pointer to the specified SPE's MFC multisource synchronization register problem state area as defined by the following structure:

```
typedef struct spe_mssync_area
{
 unsigned int MFC_MSSync;
} spe_mssync_area_t;
```

SPE_MFC_COMMAND_AREA

Return a pointer to the specified SPE's MFC command parameter and command queue control area as defined by the following structure:

```
typedef struct spe_mfc_command_area {
 unsigned char reserved_0_3[4];
 unsigned int MFC_LSA;
 unsigned int MFC_EAH;
 unsigned int MFC_EAL;
 unsigned int MFC_Size_Tag;
 union {
  unsigned int MFC_ClassID_CMD;
  unsigned int MFC_CMDStatus;
 };
 unsigned char reserved_18_103[236];
 unsigned int MFC_QStatus;
 unsigned char reserved_108_203[252];
 unsigned int Prxy_QueryType;
 unsigned char reserved_208_21B[20];
 unsigned int Prxy_QueryMask;
 unsigned char reserved_220_22B[12];
 unsigned int Prxy_TagStatus;
} spe_mfc_command_area_t;
```

Note: The MFC_EAH and MFC_EAL registers can be written simultaneously using a 64-bit store. Likewise, MFC_Size_Tag and MFC_ClassID_CMD registers can be written simultaneously using a 64-bit store.

SPE_CONTROL_AREA

Return a pointer to the specified SPE's SPU control area as defined by the following structure:

```
typedef struct spe_spu_control_area {
 unsigned char reserved_0_3[4];
 unsigned int SPU_Out_Mbox;
 unsigned char reserved_8_B[4];
 unsigned int SPU_In_Mbox;
 unsigned char reserved_10_13[4];
 unsigned int SPU_Mbox_Stat;
 unsigned char reserved_18_1B[4];
 unsigned int SPU_RunCntl;
 unsigned char reserved_20_23[4];
 unsigned int SPU_Status;
 unsigned char reserved_28_33[12];
 unsigned int SPU_NPC;
} spe_spu_control_area_t;
```

SPE_SIG_NOTIFY_1_AREA

Return a pointer to the specified SPE's signal notification area 1 as defined by the following structure:

```
typedef struct spe_sig_notify_1_area {
 unsigned char reserved_0_B[12];
 unsigned int SPU_Sig_Notify_1;
} spe_sig_notify_1_area_t;
```

SPE_SIG_NOTIFY_2_AREA

Return a pointer to the specified SPE's signal notification area 2 as defined by the following structure:

```
typedef struct spe_sig_notify_2_area {
 unsigned char reserved_0_B[12];
 unsigned int SPU_Sig_Notify_2;
} spe_sig_notify_2_area_t;
```

SEE ALSO

spe_ls_area_get(3); spe_context_create(3)

The data structures specified above are defined in the header files of the library implementation.

Chapter 9. PPE-assisted library facilities

The SPEs in the Cell BE are designed to bear the computational workload of an application. They are not well-suited for the general purpose code that is often needed outside the "compute kernels" of an application.

The SPE Runtime Management Library provides the infrastructure that enables the SPE program to issue a callback to the PPE-side of the SPE thread. From an SPE program's point of view, this mechanism allows certain functions to be offloaded to the PPE.

To provide this functionality the SPE program uses the stop and signal instruction (see note) with a signal type 0x21XX to stop the SPE and notify the PPE-side of the SPE thread that the callback with number XX should be run. The SPE can also pass 4 bytes as an argument to the library function. This argument must immediately follow the stop and signal instruction in the SPE local store.

Note: See *C/C++ Language Extensions for Cell Broadband Engine Architecture*, SPU Control Intrinsics,spu_stop: stop and signal – (void) spu_stop(type)

Execution of the SPU program is stopped. The address of the stop instruction is placed into the least significant bits of the SPU NPC register. The signal type is written to the SPU status register, and the PPU is interrupted.

In libspe the execution of callbacks is handled inside the **spe_context_run** function. It recognizes the SPE callback as a special stop reason, stop and signal with a signal type in the range of 0x2100 to 0x21ff, and matches the lower 8 bit of the signal type with a list of registered library callback function handlers, which are then called. After the function returns, **spe_context_run** restarts SPE program execution at the last SPU instruction counter plus 4, that is, it skips the argument in the SPE local store.

The prototype of a valid library callback function handler must be:

```
int function_name (void *ls_base, unsigned int ls_address)
```

Parameters

ls_base
A pointer to the beginning of the memory-mapped SPE local store.

ls_address
the offset of the callback argument relative to *ls_base* in bytes.

Return values

On success, the function returns 0 (zero).

A non-zero return value is interpreted as failure. In this case, the SPE stops, *spe_context_run* returns with an SPE_CALLBACK_ERROR, and this return value is reported as part of *stopinfo*.

Example

A simple example of a callback that just prints its argument:

```
/*
 * simple library callback handler
 */

int simple_handler (void *ls_base, unsigned int ls_address)
{
 int arg = *((int *)((char *)ls_base + ls_address));

 printf ("callback argument was %d \n", arg];
 return 0;
};
```

Before you can use a library callback function, you must use the libspe function **spe_callback_handler_register** to register it. If an SPE program tries to use a callback that has not been properly registered, the SPE stops and spe_context_run returns with an SPE_CALLBACK _ERROR.

Implementations of libspe can reserve certain callback numbers for ″built-in″ functions:

Note: (Linux) The Cell BE Linux Reference Implementation Application Binary Interface Specification reserves certain standardized library classes and call opcodes. These correspond the following reserved callnums in libspe:

0	ISO/IEC C Standard Header
1	POSIX.1 (IEEE Standard 1003.1)
2	POSIX.4
3	Operating system-dependent system calls

If invalid opcodes and/or invalid pointers are passed to callbacks corresponding to these reserved callnums as their arguments, the callbacks return non-zero values to indicate failure.

PPE-assisted library functions

The following section describes the PPE-assisted library functions.

spe_callback_handler_register

NAME

spe_callback_handler_register - Register a user-defined function specified by the function pointer *handler* as the library callback function identified by *callnum*.

SYNOPSIS

#include <libspe2.h>

int spe_callback_handler_register (void *handler, unsigned int callnum, unsigned int mode)

Parameters

handler	A function pointer to the user-defined callback handler.
callnum	The identifier under which to register this callback function. The valid range is 0..255.
mode	Specifies if a new callback should be registered or if an existing callback is being updated by this call. If a new registration is requested using a preexisting identifier for *callnum*, the request fails. If an update is requested for an unregistered value of *callnum*, the request fails.

DESCRIPTION

The **spe_callback_handler_register** function registers a new user-defined function specified by the function pointer as the library callback handler identified by callnum or updates an existing registration. In either case, *handler* must be a valid function pointer. and cannot be NULL. The function can also be used to update a built-in function registration. If an application wants to make a temporary change to an existing callback handler registration, it should first query the existing handler, then update with its own, temporary handler, and once the temporary handler is no longer required perform another update restoring the original handler registration.

RETURN VALUE

On success, the function returns **0** (zero).

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

EINVAL	Function argument error, for example, callnum out of range.
ESRCH	Error updating registration - no existing registration found.
EACCES	Error registering new callback - a callback is already registered for this callnum.

OPTIONS

Possible values of *mode* are as follows:

Values	Description
SPE_CALLBACK_NEW	Register a new callback handler
SPE_CALLBACK_UPDATE	Update registration of an existing callback handler

SEE ALSO

spe_context_run(3)

For Linux, see also default_c99_handler.h and default_posix1_handler.h

spe_callback_handler_deregister

NAME

spe_callback_handler_deregister - Deregister the user-defined function identified by *callnum*.

SYNOPSIS

#include <libspe2.h>

int spe_callback_handler_deregister (unsigned int callnum)

Parameters

callnum	The identifier of the function callback to be deregistered. The valid range is 0..255.

DESCRIPTION

Note: (Linux) The reserved callbacks 0..3 cannot be deregistered. They can, however, be overwritten with new, user-defined callbacks. See **spe_callback_handler_register()**.

RETURN VALUE

On success, the function returns **0** (zero).

EXIT STATUS

On error, **-1** is returned and *errno* is set to indicate the error.

Possible errors include:

EINVAL	Function argument error, for example, callnum out of range.
ESRCH	No callback registered for callnum.
EACCES	Attempt to deregister a reserved callback.

SEE ALSO

spe_context_run(3); spe_callback_handler_register(3)

spe_callback_handler_query

NAME

spe_callback_handler_query - Returns the function pointer associated with a callback number.

SYNOPSIS

#include <libspe2.h>

void *spe_callback_handler_query(unsigned int callnum)

Parameters

callnum	The function is identified by this *callnum*. The valid range is 0..255.

DESCRIPTION

The function **spe_callback_handler_query** returns the function pointer associated with a callback number.

RETURN VALUE

On success, the function returns the function pointer to the user-defined or built-in callback handler registered for *callnum*.

EXIT STATUS

On error, **0** (zero) is returned and *errno* is set to indicate the error.

Possible errors include:

EINVAL	Function argument error, for example, *callnum* not in valid range.
ESRCH	No callback registered for *callnum* or no registration for the provided function pointer can be found.

Appendix A. Data structures

This section summarizes the specified data structures upon which the libspe API relies. These data structures are defined in the <libspe2.h> header file. Any libspe application should include this header file.

SPE context

```
 /*
  * spe_context_ptr_t
  *  This pointer serves as the identifier for a specific
  * SPE context throughout the API (where needed)
  */
typedef struct spe_context * spe_context_ptr_t;
```

SPE gang context

```
/*
 * spe_gang_context_ptr_t
 *  This pointer serves as the identifier for a specific
 * SPE gang context throughout the API (where needed)
 */
typedef struct spe_gang_context * spe_gang_context_ptr_t;
```

SPE program handle

```
/*
 * SPE program handle
 * Structure spe_program_handle per CESOF specification
 * libspe2 applications usually only keep a pointer
 * to the program handle and do not use the structure
 * directly.
 */
typedef struct spe_program_handle {
 /*
  * handle_size allows for future extensions of the spe_program_handle
  * struct by new fields, without breaking compatibility with existing users.
  * Users of the new field would check whether the size is large enough.
  */
 unsigned int handle_size;
 void    *elf_image;
 void    *toe_shadow;
} spe_program_handle_t;
```

SPE runtime error information

```
/*
 * SPE stop information
 * This structure is used to return all information available
 * on the reason why an SPE program stopped execution.
 * This information is important for some advanced programming
 * patterns and/or detailed error reporting.
 */

/* spe_stop_info_t
 */
typedef struct spe_stop_info {
 unsigned int stop_reason;
 union {
  int spe_exit_code;
  int spe_signal_code;
  int spe_runtime_error;
  int spe_runtime_exception;
```

```
  int spe_runtime_fatal;
  int spe_callback_error;
  void *__reserved_ptr;
     unsigned long long __reserved_u64;
 } result;
 int spu_status;
} spe_stop_info_t;
```

SPE problem state areas

```
/* spe problem state areas
 */

typedef struct spe_mssync_area {
 unsigned int MFC_MSSync;
} spe_mssync_area_t;

typedef struct spe_mfc_command_area {
 unsigned char reserved_0_3[4];
 unsigned int MFC_LSA;
 unsigned int MFC_EAH;
 unsigned int MFC_EAL;
 unsigned int MFC_Size_Tag;
 union {
  unsigned int MFC_ClassID_CMD;
  unsigned int MFC_CMDStatus;
 };
 unsigned char reserved_18_103[236];
 unsigned int MFC_QStatus;
 unsigned char reserved_108_203[252];
 unsigned int Prxy_QueryType;
 unsigned char reserved_208_21B[20];
 unsigned int Prxy_QueryMask;
 unsigned char reserved_220_22B[12];
 unsigned int Prxy_TagStatus;
} spe_mfc_command_area_t;

typedef struct spe_spu_control_area {
 unsigned char reserved_0_3[4];
 unsigned int SPU_Out_Mbox;
 unsigned char reserved_8_B[4];
 unsigned int SPU_In_Mbox;
 unsigned char reserved_10_13[4];
 unsigned int SPU_Mbox_Stat;
 unsigned char reserved_18_1B[4];
 unsigned int SPU_RunCntl;
 unsigned char reserved_20_23[4];
 unsigned int SPU_Status;
 unsigned char reserved_28_33[12];
 unsigned int SPU_NPC;
} spe_spu_control_area_t;

typedef struct spe_sig_notify_1_area {
  unsigned char reserved_0_B[12];
  unsigned int SPU_Sig_Notify_1;
 } spe_sig_notify_1_area_t;

typedef struct spe_sig_notify_2_area {
  unsigned char reserved_0_B[12];
  unsigned int SPU_Sig_Notify_2;
 } spe_sig_notify_2_area_t;
```

SPE event structure

```
/*
 * SPE event structure
 * This structure is used for SPE event handling
 */
```

```
/*
 * spe_event_data_t
 * User data to be associated with an event
 */
typedef union spe_event_data {
  void *ptr;
  unsigned int u32;
  unsigned long long u64;
} spe_event_data_t;

/* spe_event_t
 */
typedef struct spe_event_unit {
  unsigned int events;
  spe_context_ptr_t spe;
  spe_event_data_t data;
} spe_event_unit_t;
```

Appendix B. Symbolic constants

This section summarizes the specified symbolic constants the libspe API relies on. These symbols are defined in the <libspe2.h> header file. Any libspe application should include this header file.

SPE context creation

SPE_EVENTS_ENABLE	Event handling is enabled on this SPE context.
SPE_CFG_SIGNOTIFY1_OR	Configure the SPU Signal Notification 1 Register to be in ″logical OR″ mode instead of the default ″Overwrite″ mode.
SPE_CFG_SIGNOTIFY2_OR	Configure the SPU Signal Notification 2 Register to be in ″logical OR″ mode instead of the default ″Overwrite″ mode.
SPE_MAP_PS	Request permission for memory-mapped access to the SPE's problem state area(s).
SPE_ISOLATED	This context runs on an SPU in the isolation mode. Programs loaded into contexts flagged with SPE_ISOLATED must be be correctly formatted for isolated execution.
SPE_ISOLATED_EMULATE	Run this context on an SPU in an emulated isolation mode. This mode provides emulation of an isolated SPU without truly being isolated as is intended for use by developers who need access to debug tools during the development of their isolated applications. Programs loaded into contexts flagged with SPE_ISOLATED_EMULATE must be correctly formatted for isolated emulation execution. **Note: (Linux)** Proper operation of a PPE assisted function call assumes the use of the ISOLATED version of the SPE library functions.

spe_gang_context_create

<none>	<none defined>

SPE run control

spe_context_run

SPE_RUN_USER_REGS	Specifies that the SPE setup registers r3, r4, and r5 are initialized with the 48 bytes pointed to by argp.

SPE_NO_CALLBACKS	Specifies that registered SPE library calls ("callbacks" from this library's view) should *not* be automatically executed. If a callback is encountered, **spe_context_run** returns as if the SPU would have issues a regular stop and signal instruction. Details can then be found in *stopinfo*.

spe_context_run; spe_stop_info_read

SPE_EXIT	SPE program terminated calling exit(code) with code in the range 0..255. The code will be saved in *spe_exit_code*.
SPE_STOP_AND_SIGNAL	SPE program stopped because SPU executed a stop and signal instruction. Further information in *spe_signal*.
SPE_RUNTIME_ERROR	SPE program stopped because of one of the reasons found in *spe_runtime_error*. **Note: (Linux)** The error SPE_SPU_INVALID_INSTR is reported as a Linux signal SIGILL if the SPE context was created without the flag SPE_EVENTS_ENABLE.
SPE_RUNTIME_EXCEPTION	SPE program stopped asynchronously because of a runtime exception (event) described in *spe_runtime_exception*. In this case, *spe_status* would be meaningless and is therefore set to -1. **Note: (Linux)** This error situation can only be caught and reported by spe_context_run if the SPE context was created with the flag SPE_EVENTS_ENABLE indicating that event support is requested. Otherwise the Linux kernel generates a signal to indicate the runtime error.
SPE_RUNTIME_FATAL	SPE program stopped for other reasons, usually fatal operating system errors such as insufficient resources. Further information in *spe_runtime_fatal* In this case, *spe_status* would be meaningless and is therefore set to -1.
SPE_CALLBACK_ERROR	An SPE program tried to use unregistered library callback, or a library callback returned a non-zero exit value, which is provided in *spe_callback_error*.
SPE_ISOLATION_ERROR	The SPE isolation system mechanism has detected an error when attempting to load the isolated SPE program.
SPE_DMA_ALIGNMENT	A DMA alignment error occurred.
SPE_DMA_SEGMENTATION	A DMA segmentation error occurred.
SPE_DMA_STORAGE	A DMA storage error occurred.
SPE_INVALID_DMA	An invalid DMA error.

SPE_SPU_HALT	SPU was stopped by halt
SPE_SPU_SINGLE_STEP	SPU is in single-step mode
SPE_SPU_INVALID_INSTR	SPU has tried to run an invalid instruction
SPE_SPU_INVALID_CHANNEL	SPU has tried to access an invalid channel

SPE events

SPE_EVENT_OUT_INTR_MBOX	Data available to be read from the SPU outbound interrupting mailbox. This event will be generated, if the SPU has written at least one entry to the SPU outbound interrupting mailbox (see **spe_out_intr_mbox_read**).
SPE_EVENT_IN_MBOX	Data can now be written to the SPU inbound mailbox. This event will be generated, if the SPU inbound mailbox had been full and the SPU read at least on entry, so that now it can be written to the SPU inbound mailbox again (see **spe_in_mbox write**).
SPE_EVENT_TAG_GROUP	An SPU event tag group signaled completion (see **spe_tag_group_read**).
SPE_EVENT_SPE_STOPPED	Program execution on the SPE has stopped (see **spe_stop_info_read**).
SPE_EVENT_ALL_EVENTS	Interest in all defined SPE events. This corresponds to a bit-wise OR of all flags above.

SPE tag group completion facility

SPE_TAG_ALL	The function suspends execution until all DMA commands in the tag groups enabled by the *mask* parameter have no outstanding DMAs in the proxy command queue of the SPE context specified by *spe*. The masked tag status is returned.
SPE_TAG_ANY	The function suspends execution until any DMA commands in the tag groups enabled by the *mask* parameter have no outstanding DMAs in the proxy command queue of the SPE context specified by *spe*. The masked tag status is returned.
SPE_TAG_IMMEDIATE	The function returns the tag status for the tag groups specified by the *mask* parameter for the proxy command queue of the SPE context specified by the *spe*.

SPE mailbox facility

SPE_MBOX_ALL_BLOCKING	The call blocks until all *count* mailbox messages have been read.
SPE_MBOX_ANY_BLOCKING	The call blocks until at least one mailbox message has been read.

SPE_MBOX_ANY_NONBLOCKING	The call reads as many mailbox messages as possible up to a maximum of *count* without blocking.

SPE problem state areas

SPE_MSSYNC_AREA	MFC multisource synchronization register problem state area.
SPE_MFC_COMMAND_AREA	MFC command parameter and command queue control area.
SPE_CONTROL_AREA	SPE control area.
SPE_SIG_NOTIFY_1_AREA	SPE signal notification area 1.
SPE_SIG_NOTIFY_2_AREA	SPE signal notification area 2.

spe_cpu_info_get

SPE_COUNT_PHYSICAL_CPU_NODES	Requests the number of physical CPU nodes of the system.
SPE_COUNT_PHYSICAL_SPES	Requests the total number of physical SPEs available either on the whole system or on a specified node.
SPE_COUNT_USABLE_SPES	Requests the number of SPEs that can actually be used by the application at this point in time.
SPE_CPU_IS_CELLBE	Identifies the CPU as a Cell BE CPU.
SPE_CPU_IS_CELLEDP	Identifies the CPU as a CelleDP CPU with enhanced SPU double precision capabilities.

spe_callback_handler_register

SPE_CALLBACK_NEW	Register a new callback handler.
SPE_CALLBACK_UPDATE	Update registration of an existing callback handler.

Appendix C. Related documentation

This topic helps you find related information.

Document location

Links to documentation for the SDK are provided on the developerWorks® Web site located at:

`http://www-128.ibm.com/developerworks/power/cell/`

Click on the **Docs** tab.

The following documents are available, organized by category:

Architecture

- *Cell Broadband Engine Architecture*
- *Cell Broadband Engine Registers*
- *SPU Instruction Set Architecture*

Standards

- *C/C++ Language Extensions for Cell Broadband Engine Architecture*
- *SPU Assembly Language Specification*
- *SPU Application Binary Interface Specification*
- *SIMD Math Library Specification for Cell Broadband Engine Architecture*
- *Cell Broadband Engine Linux Reference Implementation Application Binary Interface Specification*

Programming

- *Cell Broadband Engine Programming Handbook*
- *Programming Tutorial*
- *SDK for Multicore Acceleration Version 3.0 Programmer's Guide*

Library

- *SPE Runtime Management library*
- *SPE Runtime Management library Version 1.2 to Version 2.0 Migration Guide*
- *Accelerated Library Framework for Cell Programmer's Guide and API Reference*
- *Accelerated Library Framework for Hybrid-x86 Programmer's Guide and API Reference*
- *Data Communication and Synchronization for Cell Programmer's Guide and API Reference*
- *Data Communication and Synchronization for Hybrid-x86 Programmer's Guide and API Reference*
- *SIMD Math Library Specification*
- *Monte Carlo Library API Reference Manual (Prototype)*

Installation

- *SDK for Multicore Acceleration Version 3.0 Installation Guide*

IBM® XL C/C++ Compiler and IBM XL Fortran Compiler

Details about documentation for the compilers is available on the developerWorks Web site.

IBM Full-System Simulator and debugging documentation

Detail about documentation for the simulator and debugging tools is available on the developerWorks Web site.

PowerPC Base

- *PowerPC Architecture™ Book, Version 2.02*
  - *Book I: PowerPC User Instruction Set Architecture*
  - *Book II: PowerPC Virtual Environment Architecture*
  - *Book III: PowerPC Operating Environment Architecture*
- *PowerPC Microprocessor Family: Vector/SIMD Multimedia Extension Technology Programming Environments Manual Version 2.07c*

Appendix D. Accessibility features

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.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing 2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Software Interoperability Coordinator, Department 49XA
3605 Highway 52 N
Rochester, MN 55901
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating

platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Edition notices

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

alphaWorks
BladeCenter
developerWorks
IBM
POWER
Power PC
PowerPC
PowerPC Architecture

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

Intel®, Intel Inside® (logos), MMX, and Pentium® are trademarks of Intel Corporation in the United States, other countries, or both.

Microsoft®, Windows®, Windows NT®, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java™ and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Red Hat, the Red Hat “Shadow Man” logo, and all Red Hat-based trademarks and logos are trademarks or registered trademarks of Red Hat, Inc., in the United States and other countries.

UNIX® is a registered trademark of The Open Group in the United States and other countries.

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

Glossary

This glossary contains terms and abbreviations used in libspe and Cell/B.E. systems.

ELF

Executable and Linking Format. The standard object format for many UNIX operating systems, including Linux. Compilers generate ELF files. Linkers link to files with ELF files in libraries. Systems run ELF files.

Gang context

The SPE gang context is one of the base data structures for the libspe implementation. It holds all persistent information about a group of SPE contexts that should be treated as a gang, that is, be executed together with certain properties. This data structure should not be accessed directly; instead the application uses a pointer to an SPE gang context as an identifier for the SPE gang it is dealing with through libspe API calls.

LS

Local Store. The 256-KB local store associated with each SPE. It holds both instructions and data.

Main thread

The application's main thread. In many cases, CBEA programs are multi-threaded using multiple SPEs running concurrently. A typical scenario is that the application consists of a main thread that creates as many SPE threads as needed and ″orchestrates″ them.

MFC

Memory Flow Controller. Part of an SPE which provides two main functions: it moves data via DMA between the SPE's local store (LS) and main storage, and it synchronizes the SPU with the rest of the processing units in the system.

PPE

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

SPE

Synergistic Processor Element. It includes a SPU, a MFC, and a LS.

SPE context

The SPE context is one of the base data structures for the libspe implementation. It holds all persistent information about a ″logical SPE″ used by the application. This data structure should not be accessed directly; instead the application uses a pointer to an SPE context as an identifier for the ″logical SPE″ it is dealing with through libspe API calls.

SPE event

In a multi-threaded environment, it is often convenient to use an event mechanism for asynchronous notification. A common usage is that the main thread sets up an event handler to receive notification about certain events caused by the asynchronously running SPE threads. The current library supports events to indicate that an SPE has stopped execution, mailbox messages being written or read by an SPE, and PPE-initiated DMA operations have completed.

SPE thread

A thread scheduled and run on a SPE. A program has one or more SPE threads. Each such thread has its own SPU local store (LS),.128 x 128-bit register file, program counter, and MFC Command Queues, and it can communicate with other execution units (or with effective-address memory through the MFC channel interface). The API call **spe_context_run** is a synchronous, blocking call from the perspective of the thread using it, that is, while an SPE program is executed, the associated SPE thread blocks and is usually put to ″sleep″ by the operating system.

SPU

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

Index

D

P

S

Readers' Comments — We'd Like to Hear from You

CBEA JSRE Series
Cell Broadband Engine Architecture Joint Software Reference Environment Series
SPE Runtime Management Library
Version 2.2
DRAFT

Publication No. SC33-8334-01

We appreciate your comments about this publication. Please comment on specific errors or omissions, accuracy, organization, subject matter, or completeness of this book. The comments you send should pertain to only the information in this manual or product and the way in which the information is presented.

For technical questions and information about products and prices, please contact your IBM branch office, your IBM business partner, or your authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you. IBM or any other organizations will only use the personal information that you supply to contact you about the issues that you state on this form.

Comments:

Thank you for your support.

Submit your comments using one of these channels:
- Send your comments to the address on the reverse side of this form.
- Send a fax to the following number: +49-7031-16-3456
- Send your comments via e-mail to: eservdoc@de.ibm.com
- Send a note from the web page:

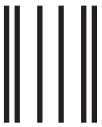
If you would like a response from IBM, please fill in the following information:

Name	Address
Company or Organization	
Phone No.	E-mail address

Readers' Comments — We'd Like to Hear from You
SC33-8334-01

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Deutschland Entwicklung GmbH
Department 3248
Schoenaicherstrasse 220
D -71032 Boeblingen
Germany

Fold and Tape | **Please do not staple** | Fold and Tape

Cut or Fold
Along Line

Printed in USA

SC33-8334-01