IBM®

SPE Runtime Management Library

Version 2.0

CBEA JSRE Series

Cell Broadband Engine Architecture
Joint Software Reference
Environment Series

November 11, 2006

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

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

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

November 11, 2006

Table of Contents

About This Document

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's Synergistic Processing Elements (SPEs).

This document and the usage of the library requires that the application programmer is familiar with the Cell Broadband Engine (CBE) architecture as described in “Cell Broadband Engine Architecture, Version 1.0”.

Audience

The document is intended for system and application programmers who wish to develop Cell Broadband Engine (CBE) applications that fully exploit the SPEs.

Version History

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 11, 2006	Initial public release of the document.

Related Documentation

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

Document Title	Version
Cell Broadband Engine Architecture	Version 1.0, August 8, 2005
Cell Broadband Engine Programming Handbook	Version 1.0, April 19, 2006

Overview

The SPE Runtime Management Library (libspe) constitutes the standardized low-level application programming interface (API) for application access to the Cell Broadband Engine's Synergistic Processing Elements (SPEs). This library provides an API that is neutral with respect to the underlying operating system and its methods to manage SPEs.

Implementations of this library may provide additional functionality that allows for access to operating system or implementation dependent aspects of SPE runtime management. These capabilities are not subject to standardization in this document and their usage may lead to non-portable code and dependencies on certain implemented versions of the library.

This document and the usage of the library require that the application programmer is familiar with the CBE architecture as described in "Cell Broadband Engine Architecture, Version 1.0".

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

Furthermore, the SPE Runtime Management Library 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 SPE context – this transfers control to the operating system requesting the actual scheduling of the context to a physical SPE in the system
4. Destroy SPE context

Note that step 3. above represents a synchronous call to the operating system. The calling application will block until the SPE stops execution and the operating system returns from the system call invoking the SPE execution.

Many applications need to use multiple SPEs concurrently. In this case, it is necessary for the application to 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. Terminate thread
4. Wait for all N threads to terminate

5. Destroy all N SPE contexts

Of course, other schemes are also possible and, depending on the application, potentially more adequate.

In order to provide this functionality, the SPE Runtime Management Library consists of various sets of PPE functions:

1. A set of PPE functions to create and destroy SPE and gang contexts.
2. A set of PPE functions to load SPE objects into SPE local store memory for execution.
3. A set of PPE functions to start the execution of SPE programs and to obtain information on reasons why an SPE has stopped running.
4. A set of PPE functions to receive asynchronous events generated by an SPE.
5. A set of PPE functions used to access the MFC (Memory Flow Control) problem state facilities, which includes
    a. MFC proxy command issue
    b. MFC proxy tag-group completion facility
    c. Mailbox facility
    d. SPE signal notification facility
6. A set of PPE functions that enable direct application access to an SPE’s local store and problem state areas.
7. Means to register PPE-assisted library calls for an SPE program.

***Terminology**:*

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.

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.

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.

SPE thread: A regular thread executing on the PPE that actually runs an SPE context is called an SPE thread. 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 will usually be put to “sleep” by the operating system.

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.

Library Name(s):

libspe2

Header File(s)

<libspe2.h>

Examples

The following example shows how to load and run a simple SPE executable “hello”:

Example 1: Run the simple SPE program “hello”

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

int main()
{
   spe_context_ptr_t spe;
   unsigned int createflags = 0;
   unsigned int runflags = 0;
   unsigned int entry = SPE_DEFAULT_ENTRY;
   void * argp = NULL;
   void * envp = NULL;
   spe_program_handle_t * program;

   program = spe_image_open("hello");

   spe = spe_context_create(createflags, NULL);
   spe_program_load(spe, program);
   spe_context_run(spe, &entry, runflags, argp, envp, NULL);
   spe_image_close(program);
   spe_context_destroy(spe);
}
```

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>

#define N 4

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

void my_spe_thread(struct thread_args * arg) {
   unsigned int runflags = 0;
   unsigned int entry = SPE_DEFAULT_ENTRY;

  // run SPE context
  spe_context_run(arg->spe, &entry, runflags, arg->argp, arg->envp, NULL);
  // done - now exit thread
  pthread_exit(NULL);
}
```

```
int main() {

  pthread_t pts[N];
  spe_context_ptr_t spe[N];
  struct thread_args t_args[N];
  int value[N];
  int i;

  // open SPE program
  spe_program_handle_t * program;
  program = spe_image_open("hello");

  for ( i=0; i<N; i++ ) {
    // create SPE context
    spe[i] = spe_context_create(0, NULL);
    // load SPE program
    spe_program_load(spe[i], program);
    // create pthread
    t_args[i].spe = spe[i];
    t_args[i].argp = &value[i];
    t_args[i].envp = NULL;
    pthread_create( &pts[i], NULL, &my_spe_thread, t_args[i]);
  }

  // wait for all threads to finish
  for ( i=0; i<N; i++ ) {
    pthread_join (pts[i], NULL);
  }

  // close SPE program
  spe_image_close(program);

  // destroy SPE contexts
  for ( i=0; i<N; i++ ) {
    spe_context_destroy (spe[i]);
  }

  return 0;
}
```

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 not be accessed directly, but the application uses a pointer to an SPE context as an identifier for the "logical SPE" it is dealing with through libspe API calls. 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**. Once an application no longer needs a specific SPE context, it should release all associated resources and free the memory used by the SPE context data structure by calling the function **spe_context_destroy**.

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 not be accessed directly, but 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. Before being able to use an SPE gang context, that is, before being able to add SPE contexts as members to the gang by calling **spe_context_create**, the SPE gang context data structure has to be created and initialized. This is done by calling the function **spe_gang_context_create**. Once 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 by first destroying all SPE contexts associated with the gang by calling **spe_context_destroy** on each of them and then calling the function **spe_gang_context_destroy**.

spe_context_create

C Specification

#include <libspe2.h>

spe_context_ptr_t spe_context_create(unsigned int flags, spe_gang_context_ptr_t gang)

Description

Create a new SPE context.

Parameters

flags	A bit-wise OR of modifiers that are applied when the SPE context is created. The following values are accepted:
gang	Associate the new SPE context with this gang context. If NULL is specified, the new SPE context will not be associated with any gang

SPE_EVENTS_ENABLE	Event handling shall be enabled on this SPE context
SPE_CFG_SIGNOTIFY1_OR	Configure the SPU Signal Notification 1 Register[1] 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) [2].
SPE_ISOLATE	This context will execute on an SPU in the isolation mode. The specified SPE program must be correctly formated for isolated execution.

[1] See “Cell Broadband Engine Architecture, Version 1.0, section 8.7

[2] See “Cell Broadband Engine Architecture, Version 1.0, chapter 8

Return Value

On success, a pointer to the newly created SPE context is returned. On error, NULL is returned and errno will be 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 OS service occurred.

See Also

spe_gang_context_create; spe_context_destroy;

spe_context_destroy

C Specification

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

Description

Destroy the specified SPE context and free any associated resources.

Parameters

spe	Specifies the SPE context to be destroyed.

Return Value

On success, 0 is returned. On failure, -1 is returned and errno is set appropriately.

Possible errors include:

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

See Also

spe_context_create;

spe_gang_context_create

C Specification

```
#include <libspe2.h>
spe_gang_context_ptr_t spe_gang_context_create (unsigned int flags)
```

Description

Create a new SPE gang context.

Parameters

flags	A bit-wise OR of modifiers that are applied when the SPE context is created. The following values are accepted: <none> <none>

Return Value

On success, a pointer to the newly created gang context is returned. On error, NULL is returned and errno will be 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 was invalid.
EFAULT	A runtime error of the underlying OS service occurred.

See Also

spe_context_create; spe_gang_context_destroy;

spe_gang_context_destroy

C Specification

```
#include <libspe2.h>
int spe_gang_context_destroy (spe_gang_context_ptr_t gang)
```

Description

Destroy the specified gang context and free any associated resources.

Before destroying a gang context, you must destroy all associated SPE contexts using **spe_context_destroy**.

Parameters

gang	Specifies the gang context to be destroyed.

Return Value

On success, 0 is returned. On failure, -1 is returned and errno is set appropriately.

Possible errors include:

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

See Also

spe_gang_context_create; spe_context_destroy;

SPE Program Image Handling

Before being able to run an SPE context, an SPE program has to be loaded into the SPE’s local store. This is done by 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**. Details on SPE executables can be found in “Cell Broadband Engine Programming Handbook, Version 1.0”, chapter 14 “Objects, Executables, and SPE Loading”

spe_image_open

C Specification

#include <libspe2.h>

spe_program_handle_t * spe_image_open (const char *filename)

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 application needs "execute" access rights to the file with the SPE executable. SPE ELF objects loaded using this function are not shared with other applications/processes.

It is sometime 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 will 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.

Parameters

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

Return Value

On success, spe_open_image returns the address at which the specified SPE ELF object has been mapped. On failure, NULL is returned and errno is set

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; spe_context_run; spe_image_close;

spe_image_close

C Specification

```
#include <libspe2.h>
int spe_image_close (spe_program_handle_t *program)
```

Description

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

Parameters

program	A valid address of a mapped SPE program.

Return Value

On success, 0 is returned. On failure, -1 is returned and errno is set appropriately.

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;

spe_program_load

C Specification

```
#include <libspe2.h>
int spe_program_load (spe_context_ptr_t spe, spe_program_handle_t *program)
```

Description

spe_program_load loads 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**.

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.

Return Value

On success, 0 is returned. On failure, -1 is returned and errno is set appropriately.

Possible errors include:

ESRCH	The specified SPE context is invalid.
EINVAL	The specified address of the SPE program is invalid.

See Also

spe_image_open; spe_context_run;

SPE Run Control

After creating an SPE context and loading an SPE program into its local store, the application can run an SPE context by calling **spe_context_run**. A thread that executes an SPE context is called an SPE thread. The API function 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 will usually be put to "sleep" by the operating system. When the SPE program stops – either because of reaching its "normal" exit point,a stop and signal instruction or an error condition – the **spe_context_run** function returns and its return values specify the exact condition why the SPE program stopped.

Many applications need to use multiple SPEs concurrently. In this case, it is necessary for the application to 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 multi-threaded environment, it is often convenient to use an event mechanism[3] for asynchronous notification about certain events caused by the asynchronously running SPE threads. A specific event indicates that an SPE context was stopped in the SPE thread. The function **spe_stop_info_read** allows the main thread to read the full information about the stop reason.

[3] See section "SPE Event Handling" in this document

spe_context_run

C Specification

#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)
```

Description

The function **spe_context_run** requests execution of an SPE context on a physical SPE resource of the system. It is necessary that a SPE program has been loaded (using **spe_program_load**) before running the SPE context.

The thread calling **spe_context_run** will block and wait until the SPE stops, either because of normal termination of the SPE program, an SPU stop and signal instruction, or some 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 of the termination condition in a structured way that allows applications more fine-grained error handling and 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" in 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 flexible on all possible conditions and not rely on pre-defined conventions.

Parameters

spe	A valid 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 default entry point for the SPU main program obtained from the loaded SPE image will be used. This is usually the local store address of the initialization function crt0[4]. 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.

[4] See "Cell Broadband Engine Programming Handbook, Version 1.0", chapter 14 "Objects, Executables, and SPE Loading"

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, the SPE program can be continued by simply calling **spe_context_run** again without changing *entry*.

runflags

A bitmask 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.

The following flags are valid. 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* be automatically executed. If a callback is encountered. This also disables callbacks that are predefined in the library implementation. See also section "PPE-assisted library calls" for more details. **spe_context_run** will return as if the SPU would have issues a regular stop and signal instruction. The signal code will be returned as part of *stopinfo*.

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 (specified below).

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

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. This is sufficient for many applications.

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 will not be returned by the call.

The data type is spe_stop_info_t which 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;
        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 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 field *spe_signal_code*.
SPE_RUNTIME_ERROR	SPE program stopped because of a one of the reasons found in *spe_runtime_error*.
SPE_RUNTIME_EXCEPTION	SPE program stopped asynchronously because of an runtime exception (event) described in *spe_runtime_exception*. In this case, *spe_status* would be meaningless and is therefore set to -1. Linux Note: 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 indicated that event support is requested. Otherwise the Linux kernel will generate 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	A library callback returned a non-zero exit value, which is provided in *spe_callback_error*. *spe_status* contains the information about the failed library callback (*spe_status* & 0x3fff0000 is the stop code which led to the library callback.)

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[5]	
spe_signal_code	Stop and signal code sent by the SPE program. The lower 14-bit of this field contain the signal number.	
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_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.	

The field *spu_status* contains the value of the architected "SPU Status Register (SPU_Status)" as defined in the Cell Broadband Engine Architecture V1.0 in 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 will not need this field.

[5] 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*.

Return Value

On success, 0 or a positive number is returned.

A return value of 0 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.

On failure, -1 is returned and errno is set appropriately.

Possible errors include:

ESRCH	The specified SPE context is invalid.
EINVAL	Invalid parameters.
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*.

See Also

spe_context_create, spe_program_load,

Note

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 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.

spe_stop_info_read

C Specification

#include <libspe2.h>

int spe_stop_info_read (spe_context_ptr_t spe, spe_stop_info_t *stopinfo)

Description

Read information about the exact conditions in which the SPE identified by *spe* stopped program execution, corresponding to the last SPE_EVENT_STOPPED event.

This function is intended for usage in a multi-threaded environment. An SPE thread would run the SPE context using **spe_context_run**. A main thread would be able to receive stop events, whenever 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 will return 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 will return an error ENOTSUP.

Parameters

spe	A valid 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 will be filled with all information available as to the reason why the SPE program stopped execution.

Return Value

On success, 0 is returned. On failure, -1 is returned and errno is set appropriately.

Possible errors include:

ESRCH	The specified SPE context is invalid.
EAGAIN	No data available.
ENOTSUP	Event processing is not enabled for this SPE context.

See Also

spe_context_run;

SPE Event Handling

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, 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 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. 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_handler_create

C Specification

```
#include <libspe2.h>
spe_event_handler_ptr_t spe_event_handler_create(void)
```

Description

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

Parameters

void	*none*

Return Value

On success, a valid pointer to an SPE event handler is returned. On failure, NULL is returned and errno is set appropriately.

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;

spe_event_handler_destroy

C Specification

#include <libspe2.h>

int spe_event_handler_destroy (spe_event_handler_ptr_t evhandler);

Description

Destroy a SPE event handler and free all resources associated with it.

Parameters

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

Return Value

On success, 0 is returned. On failure, -1 is returned and errno is set appropriately.

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; spe_event_wait;

spe_event_handler_register

C Specification

```
#include <libspe2.h>
int spe_event_handler_register(spe_event_handler_ptr_t evhandler, spe_event_unit_t *event);
```

Description

Register the application's interest in SPE events of the specified nature 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 will return an error ENOTSUP.

Parameters

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

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 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_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;
```

Return Value

On success, 0 is returned. On failure, -1 is returned and errno is set appropriately.

Possible errors include:

ESRCH	The specified SPE event handler is invalid.
EINVAL	The specified pointer to an 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.

See Also

spe_event_handler_deregister; spe_event_wait; spe_out_intr_mbox_read; spe_in_mbox_write; spe_mfcio_tag_status_read; spe_stop_info_read;

spe_event_handler_deregister

C Specification

```
#include <libspe2.h>

int spe_event_handler_deregister(spe_event_handler_ptr_t evhandler,
    spe_event_unit_t *event);
```

Description

Deregister 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 will return an error ENOTSUP.

Parameters

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

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.

Return Value

On success, 0 is returned. On failure, -1 is returned and errno is set appropriately.

Possible errors include:

ESRCH	The specified SPE event handler is invalid.
EINVAL	The specified pointer to an 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.

See Also

spe_event_handler_register; spe_event_wait; spe_out_intr_mbox_read; spe_in_mbox_write; spe_mfcio_tag_status_read; spe_stop_info_read;

spe_event_wait

C Specification

```
#include <libspe2.h>

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

Description

Wait for SPE events.

Parameters

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 as 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 is returned, no SPE event was received because the request timed out. On failure, -1 is returned and errno is set appropriately.

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; spe_event_handler_deregister; spe_out_intr_mbox_read; spe_in_mbox_write; spe_mfcio_tag_status_read; spe_stop_info_read;

SPE MFC Problem State Facilities

SPE MFC Proxy Command Issue

This set of functions provides PPE-initiated DMA[6] functionality through the usage of the SPE MFC Proxy Command Issue facility. Main threads may use these functions to move data to and from an SPE local store area. Note that 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.

[6] See “Cell Broadband Engine Architecture, Version 1.0”, chapter 7 and section 8.2 and 8.3

spe_mfcio_put, spe_mfcio_putb, spe_mfcio_putf

C Specification

```
#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)

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)

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)
```

Description

The **spe_mfc_put** function places a get DMA command on the proxy command queue of the SPE context specified by *spe*. The 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 **spe_mfc_putb** function is identical to **spe_mfc_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 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 **spe_mfc_putf** function is identical to **spe_mfc_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 these functions must ensure that the address alignments and transfer size is in accordance with the limitation and restrictions of the Cell Broadband Engine Architecture.

Parameters

spe	Specifies the SPE context whose proxy command queue the put command is to be placed into.
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[7]
tid	Specifies the transfer class identifier of the DMA command.
rid	Specifies the replacement class identifier of the DMA command.

Return Value

On success, 0 is returned. On failure, -1 is returned and errno is set appropriately.

Possible errors include:

ESRCH	The specified SPE context is invalid.

See Also

[7] See "Cell Broadband Engine Architecture, Version 1.0", section 8.1.3

spe_mfcio_get, spe_mfcio_getb, spe_mfcio_getf

C Specification

```
#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)
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)
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)
```

Description

The **spe_mfc_get** function places a get DMA command on the proxy command queue of the SPE context specified by *spe*. The 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 **spe_mfc_getb** function is identical to **spe_mfc_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 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 **spe_mfc_getf** function is identical to **spe_mfc_get** except that it places a getf (get with fence) DMA command on the proxy command queue. The fence form ensure that this command is locally ordered with respect to all previously issued commands with the same tag group and command queue.

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

Parameters

spe	Specifies the SPE context into which proxy command queue the get command is to be placed into.
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[8]
tid	Specifies the transfer class identifier of the DMA command.
rid	Specifies the replacement class identifier of the DMA command.

Return Value

On success, 0 is returned. On failure, -1 is returned and errno is set appropriately.

Possible errors include:

ESRCH	The specified SPE context is invalid.

See Also

[8] See “Cell Broadband Engine Architecture, Version 1.0”, section 8.1.3

SPE MFC Proxy Tag-Group Completion Facility

spe_mfcio_tag_status_read

C Specification

```
#include <libspe2.h>

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

Description

The **spe_mfc_tag_status_read** function is used to check the completion of DMA requests associated with the tag groups specified by the optional *mask* parameter. A *mask* of value '0' indicates that all current DMA requests should be taken into account. The *behavior* field specifies whether all or any of the specified tag groups have to be completed, or whether it just checks current completion status.

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.

Parameters

spe	Specifies the SPE context for which DMA completion status is to be checked.
mask	The *mask* parameter can be set to 0 indicating that all current DMA requests should be taken into account. This will take into account only those DMAs started using libspe library calls, since the library and operating system have no way to know about DMA initiated by applications using direct problem state access. A non-zero value has to be specified according to the "Cell Broadband Engine Architecture, Version 1.0", section 8.4.3. Each of the bits 0:31 of this *mask* corresponds to a tag group. These tag groups may include those used for DMA started using application direct problem state access.
behavior	Specifies the behavior of the operation. The value can be one of:

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.

tag_status	Result: the current tag status for tags specified by *mask* is returned.

Return Value

On success, 0 is returned. On failure, -1 is returned and errno will be set appropriately.

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.

See Also

spe_mfcio_get, spe_mfcio_getb, spe_mfcio_getf, spe_mfcio_put, spu_mfcio_putb, spu_mfcio_putf

SPE Mailbox Facility

This set of functions allows a main thread to communicate with an SPE through its mailbox facility. Note that 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 is **spe_out_mbox_read** to read the mailbox message from the main thread.

spe_out_mbox_read

C Specification

```
#include <libspe2.h>
int spe_out_mbox_read (spe_context_ptr_t spe, unsigned int *mbox_data, int count)
```

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.

Parameters

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

Return Value

>0	the number of 32-bit mailbox messages read
0	no data read
-1	error condition and errno is set appropriately

Possible errors include:

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

See Also

spe_out_mbox_status;

spe_out_mbox_status

C Specification

```
#include <libspe2.h>
int spe_out_mbox_status (spe_context_ptr_t spe)
```

Description

The **spe_out_mbox_status** function fetches the status of the SPU outbound mailbox for the SPE context specified by the *spe* parameter. A 0 value is return if the mailbox is empty. A non-zero value specifies the number of 32-bit unread mailbox entries.

Parameters

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

Return Value

>0	the number of 32-bit mailbox messages available for read
0	no data available
-1	error condition and errno is set appropriately

Possible errors include:

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

See Also

spe_out_mbox_read;

spe_in_mbox_write

C Specification

```
#include <libspe2.h>

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

Description

This function writes 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 particularly useful to send a sequence of mailbox messages to 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_in_mbox_status can be called to ensure that data can be written prior to writing the SPU inbound mailbox.

Parameters

spe	Specifies the SPE context for which the SPU inbound mailbox has to be written.
mbox_data	A pointer to an array of unsigned integers of size *count* holding 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 be blocking or non-blocking. Possible values are:

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

Return Value

>0 the number of 32-bit mailbox messages written

0	no mailbox message could be written
-1	error condition and errno is set appropriately

Possible errors include:

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

See Also

spe_in_mbox_status;

spe_in_mbox_status

C Specification

```
#include <libspe2.h>
int spe_in_mbox_status (spe_context_ptr_t spe)
```

Description

The **spe_in_mbox_status** function fetches the status of the SPU inbound mailbox for the SPE context specified by the *spe* parameter. A 0 value is return if the mailbox is full. A non-zero value specifies the number of available (32-bit) mailbox entries.

Parameters

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

Return Value

>0	the number of 32-bit mailbox messages that can be written
0	no data can be written (mailbox full)
-1	error condition and errno is set appropriately

Possible errors include:

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

See Also

spe_in_mbox_write;

spe_out_intr_mbox_read

C Specification

#include <libspe2.h>

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

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 send a sequence of mailbox messages to 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.

Parameters

spe	Specifies the SPE context for which the SPU inbound mailbox has to be written.
mbox_data	A pointer to an array of unsigned integers of size *count* 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 be blocking or non-blocking. Possible values are:

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

Return Value

>0	the number of 32-bit mailbox messages read
0	no mailbox message read

-1 error condition and errno is set appropriately

Possible errors include:

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

See Also

spe_out_intr_mbox_status;

spe_out_intr_mbox_status

C Specification

```
#include <libspe2.h>
int spe_out_intr_mbox_status (spe_context_ptr_t spe)
```

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. A 0 value is return if the mailbox is empty. A non-zero value specifies the number of 32-bit unread mailbox entries.

Parameters

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

Return Value

>0	the number of 32-bit mailbox messages available for read
0	no data available
-1	error condition and errno is set appropriately

Possible errors include:

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

See Also

spe_out_intr_mbox_read;

SPE SPU Signal Notification Facility

spe_signal_write

C Specification

```
#include <libspe2.h>
int spe_signal_write (spe_context_ptr_t spe, unsigned int signal_reg, unsigned int data)
```

Description

The **spe_signal_write** function writes *data* to the signal notification register specified by *signal_reg* for the SPE context specified by the *spe* parameter.

Parameters

spe	Specifies the SPE context whose signal register is to be written to.
signal_reg	Specifies the signal notification register to be written. Valid signal notification registers are:
	SPE_SIG_NOTIFY_REG_1 — SPE signal notification register 1
	SPE_SIG_NOTIFY_REG_2 — SPE signal notification register 2
data	The 32-bit data to be written to the specified signal notification register.

Return Value

On success, 0 is returned. On failure, -1 is returned and errno is set appropriately.

Possible errors include:

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

See Also

Direct SPE Access for Applications

Applications may access directly an SPE's local store memory and the various problem state registers as described in detail below.

The function **spe_ls_area_get** maps the local store of an SPE to the thread's address space. It can then be accessed like regular system memory. This is, however, only recommended for special purpose usage, since in general DMA to and from local store is far more efficient. A more common usage of the local store mapping is to communicate the effective address of one SPE's local store to a program running on another SPE, which allows that SPE to use DMA to directly transfer data to and from another local store. This is very efficient, since the DMA transfer will directly go from SPE to SPE, and not through system memory.

The function **spe_ps_area_get** maps a selected area of an SPE's problem state registers to the thread's address space. Thus the problem state pointer can be used to access directly problem state features without having to make library system calls. Problem state features include multi-source synchronization, proxy DMAs, mailboxes, and signal notifiers. In addition, these pointers, along with local store pointers (see **spe_ls_area_get**), can be used to perform and control SPE to SPE communications via mailboxes, DMA's and signal notification.

When using direct problem state access, applications have to take special care to serialize multiple problem state operations appropriately. Also, when using both library and direct problem state operations, these must be properly serialized with respect to each other. Otherwise, unexpected behavior and/or application errors may arise.

Linux Note: Stopping a running SPU by writing to SPE_RunCntrl will not ensure that the Linux kernel (scheduler) will be informed so that it can reclaim the SPE.

spe_ls_area_get

C Specification

```
#include <libspe2.h>
void * spe_ls_area_get (spe_context_ptr_t spe)
```

Description

The **spe_ls_area_get** functions 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**.

Parameters

spe	Specifies the SPE context

Return Value

On success, a valid pointer to the start of the memory mapped local store is returned. On failure, NULL is returned and errno is set appropriately.

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.

See Also

spe_ps_area_get; spe_ls_size_get;

spe_ls_size_get

C Specification

```
#include <libspe2.h>
int spe_ls_size_get (spe_context_ptr_t spe)
```

Description

The **spe_ls_size_get** function returns the size of the SPE local store in number of bytes.

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 therefore not rely on a specific local store size, for example, 256 KB, but obtain the actual value through this call.

Parameters

spe	Specifies the SPE context

Return Value

On success, a positive number representing the SPE local store size in number of bytes is returned. On failure, -1 is returned and errno is set appropriately.

Possible errors include:

ESRCH	The specified SPE context is invalid.

See Also

spe_ls_area_get; spe_mfcio_get/getb/getf/put/putb/putf

spe_ps_area_get

C Specification

```
#include <libspe2.h>

void * spe_ps_area_get (spe_context_ptr_t spe, enum ps_area area)
```

Description

The **spe_ps_area_get** function maps the problem state area specified by *ps_area* of the SPE context specified by *spe* to the thread's address space and returns a pointer to the beginning of that problem state.

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

Parameters

spe	The identifier of a specific SPE context.
ps_area	The problem state area pointer to be granted access and returned. Possible problem state areas include:

SPE_MSSYNC_AREA	Return a pointer to the specified SPE's MFC multisource synchronization register problem state area as defined by the following structure:
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_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;
```

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;
```

Return Value

On success, a valid pointer to the requested problem state area is returned. On failure, NULL is returned and errno is set appropriately.

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.

See Also

spe_ls_area_get; spe_context_create;

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

PPE-assisted Library Calls

The SPEs on a CBEA processor are designed to bear the computational workload of an application. They are not very well-suited for the general purpose code 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 for the offloading of certain functions to the PPE.

The SPE program uses the stop and signal instruction[9] 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 executed. The SPE may pass 4 bytes as an argument to the library function. This argument has to follow immediately the stop and signal instruction in the SPE local store.

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 will then be 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 Value:

On success, the function returns 0.

A non-zero return value is interpreted as failure. This return value will be reported as part of *stopinfo*.

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));
```

[9] See “SPE C/C++ Language Extensions, Version 2.1”, section 2.11. “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.

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

Before being able to use a library callback function, it has to be registered using the libspe function **spe_callback_handler_register**.

Implementations of libspe may reserve certain callback numbers for “built-in” functions. For example, the Linux implementation of this library reserves the numbers 0 and 1 for C99 and Posix handlers respectively.

spe_callback_handler_register

C Specification

```
#include <libspe2.h>
void spe_callback_handler_register (void *handler, unsigned int callnum)
```

Description

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

Linux Note: The *callnum*s 0 and 1 are reserved for C99 and Posix calls respectively.

Parameters

handler	A function pointer to the user-defined callback handler.
callnum	The function will be identified by this *callnum*. The valid range is 0..255.

Return Value

none

See Also

spe_context_run;

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

spe_callback_handler_deregister

C Specification

```
#include <libspe2.h>
void spe_callback_handler_deregister (unsigned int callnum)
```

Description

The **spe_callback_handler_deregister** function deregisters a user-defined function specified by the function pointer *handler* as the library callback function identified by *callnum*.

Linux Note: The *callnum*s 0 and 1 are reserved for C99 and Posix calls respectively and cannot be deregistered.

Parameters

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

Return Value

none

See Also

spe_context_run; spe_callback_handler_register;

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_context_create

SPE_EVENTS_ENABLE	Event handling will 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 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_ISOLATE	This context will execute on an SPU in the isolation mode. The specified SPE program must be correctly formatted for isolated execution.

spe_gang_context_create

<none>	<none defined today>

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** will return 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*.
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. Linux Note: 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 will generate 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	A library callback returned a non-zero exit value, which is provided in *spe_callback_error.* *spe_status* contains the information about the failed library callback (*spe_status* & 0x3fff0000 is the stop code which led to the library callback)

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_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 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 will block until all *count* mailbox messages have been read.
SPE_MBOX_ANY_BLOCKING	The call will block until at least one mailbox message has been read.
SPE_MBOX_ANY_NONBLOCKING	The call will read 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