SPE Runtime Management Library

Version 2.0

September 29, 2007

Printed in the United States of America June 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

June 15, 2006

Contents

Chapter 1

Overview

The libspe2 functionality is split into 4 libraries:

- **libspe-base** This library provides the basic infrastructure to manage and use SPEs. The central data structure is a SPE context spe_context. It contains all information necessary to manage an SPE, run code on it, communicate with it, and so on. To use the libspe-base library, the header file **spebase.h** has to be included and and an application needs to link against **libspebase.a** or **libspebase.so**.

- **libspe-event** This is a convenience library for the handling of events generated by an SPE. It is based on libspe-base and epoll. Since the spe_context introduced by libspe-base contains the file descriptors to mailboxes etc, any other event handling mechanism could also be implemented based on libspe-base.

1.1 Terminology

- **main thread** usually the application main thread running on a PPE

- **SPE thread** a thread that uses SPEs. Execution starts on the PPE. Execution shifts between PPE and an SPE back and fro, e.g., PPE services system calls for SPE transparently

1.2 Usage Scenarios

1.2.1 Single-threaded sample

Note: In the new model, it is not necessary to have a main thread - the SPE thread can be the only application thread. It may run parts of its code on PPE and then start an SPE, e.g., for an accelerated function. The main thread is needed only if you want to use multiple SPEs concurrently. The following minimalistic sample illustrates the basic steps:

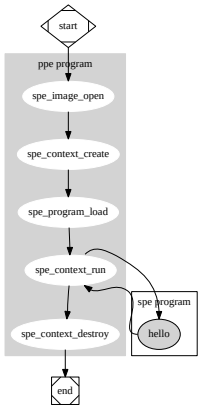

Figure 1.1: Simple program

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

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

        program = spe_image_open("hello");

        ctx = spe_context_create(flags, 0);
        spe_program_load(ctx, program);
        spe_context_run(ctx, &entry, flags, argp, envp, NULL);
        spe_context_destroy(ctx);
}
```

Here is the same sample with some error checking:

```
#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;
}
```

1.2.2 Multi-threaded sample

This illustrates a threaded sample using the pthread library:

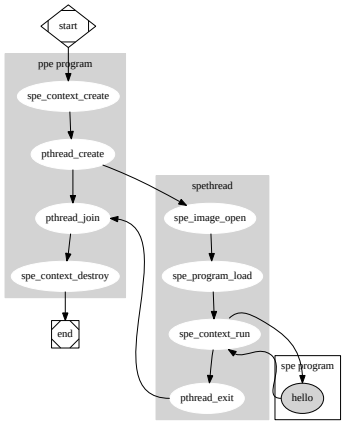

Figure 1.2: Simple pthread program

```
#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;
}
```

Here is the same sample with some error checking:

```
#include <errno.h>
#include <stdio.h>
#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);

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

```
        arg_ptr = (struct thread_args *) arg;

        program = spe_image_open("hello");
        if (!program) {
                perror("spe_image_open");
                pthread_exit(NULL);
        }

        if (spe_program_load(arg_ptr->ctx, program)) {
                perror("spe_program_load");
                pthread_exit(NULL);
        }

        rc = spe_context_run(arg_ptr->ctx, &entry, flags, arg_ptr->argp, arg_ptr->envp, NULL);
        if (rc < 0)
                perror("spe_context_run");

        pthread_exit(NULL);
}

int main() {
        int thread_id;
        pthread_t pts;
        spe_context_ptr_t ctx;
        struct thread_args t_args;
        int value = 1;
        int flags = 0;

        if (!(ctx = spe_context_create(flags, NULL))) {
                perror("spe_create_context");
                return -2;
        }

        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;
}
```

1.2.3 Problem state mapping samples

This illustrates accessing the MFC Local Store Address Register.

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

int main(void)
{
        spe_context_ptr_t ctx;
        int flags = SPE_MAP_PS;
        struct spe_mfc_command_area * mfc_cmd_area;
        struct spe_spu_control_area * spu_control_area;
        unsigned int MFC_LSA;
        unsigned int status;

        printf("starting ..\n");
        ctx = spe_context_create(flags, NULL);
```

```
        mfc_cmd_area = spe_ps_area_get(ctx, SPE_MFC_COMMAND_AREA);
        printf("mfc_cmd_area is: %p\n", mfc_cmd_area);
        MFC_LSA = mfc_cmd_area->MFC_LSA;
        spu_control_area = spe_ps_area_get(ctx, SPE_CONTROL_AREA);
        status = spu_control_area->SPU_Status;
        spe_context_destroy(ctx);
        printf("%d done\n", status);
        printf("%d done\n", MFC_LSA);
}
```

1.2.4 Event samples

This illustrates a sample using the event libary. The event, which we receive is of course that the spu program has stopped, because otherwise we would not get there.

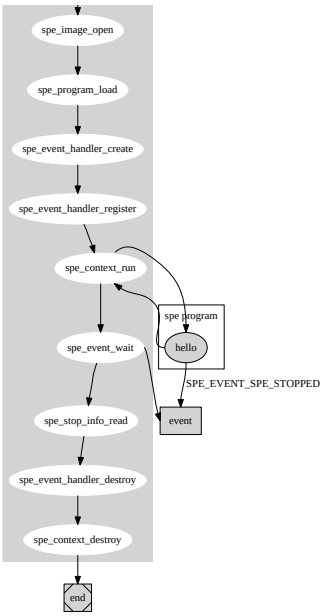

Figure 1.3: Simple event program

```
#include <libspe2.h>

#define MAX_EVENTS 8
#define SIZE 8
#define COUNT 1

int main()
{
        int i, rc, event_count;
        spe_event_handler_ptr_t evhandler;
        spe_event_unit_t event;
        spe_context_ptr_t ctx;
        spe_event_unit_t events[MAX_EVENTS];
        spe_stop_info_t stop_info;
        spe_program_handle_t * program;
        unsigned int entry = SPE_DEFAULT_ENTRY;
        void * argp = NULL;
        void * envp = NULL;
```

```
        /* Create a context. */
        ctx = spe_context_create(SPE_EVENTS_ENABLE, NULL);
        if (ctx == NULL) {
                perror("spe_context_create");
                return -2;
        }

        /* load the program. */
        program = spe_image_open("hello");
        if (!program) {
                perror("spe_open_image");
                return -1;
        }

        if (spe_program_load(ctx, program)) {
                perror("spe_program_load");
                return -3;
        }

        /* Create a handle. */
        evhandler = spe_event_handler_create();

        /* Register events. */
        event.events = SPE_EVENT_SPE_STOPPED;
        event.spe = ctx;
        rc = spe_event_handler_register(evhandler, &event);

        /* run the context */
        rc = spe_context_run(ctx, &entry, 0, argp, envp, &stop_info);
        if (rc < 0)
                perror("spe_context_run");

        /* Get events. */
        event_count = spe_event_wait(evhandler, events, MAX_EVENTS, 0);
        printf("event_count: %d\n", event_count);

        /* Handle events. */
        for (i = 0; i < event_count; i++) {
                printf("event %d: %d\n", i, events[i].events);
                if (events[i].events & SPE_EVENT_SPE_STOPPED) {
                        printf("received SPE_EVENT_SPE_STOPPED\n");
                        rc = spe_stop_info_read(events[i].spe, &stop_info);
                        printf("exit_code: %d\n", stop_info.result.spe_exit_code);
                }
        }

        /* Destroy the handle. */
        spe_event_handler_destroy(evhandler);

        /* Destroy the context. */
        spe_context_destroy(ctx);

        return 0;
}
```

Events are more useful in multithreaded environments:

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

#define MAX_EVENTS 8
#define SIZE 8
```

```
#define COUNT 1

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

void * spe_thread(void * arg);

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

        arg_ptr = (struct thread_args *) arg;

        program = spe_image_open("hellointr");
        if (!program) {
                perror("spe_image_open");
                pthread_exit(NULL);
        }

        if (spe_program_load(arg_ptr->ctx, program)) {
                perror("spe_program_load");
                pthread_exit(NULL);
        }

        rc = spe_context_run(arg_ptr->ctx, &entry, flags, arg_ptr->argp, arg_ptr->envp, NULL);
        if (rc < 0)
                perror("spe_context_run");

        pthread_exit(NULL);
}

int main() {
        int thread_id;
        int i, rc, event_count;
        pthread_t pts;
        spe_context_ptr_t ctx;
        struct thread_args t_args;
        int value = 1;
        int flags = SPE_EVENTS_ENABLE;
        spe_event_handler_ptr_t evhandler;
        spe_event_unit_t event;
        spe_event_unit_t events[MAX_EVENTS];
        unsigned int mbox_data[COUNT];
        spe_stop_info_t stop_info;
        int cont;

        if (!(ctx = spe_context_create(flags, NULL))) {
                perror("spe_create_context");
                return -2;
        }

        /* Create a handle. */
        evhandler = spe_event_handler_create();

        /* Register events. */
        event.events = SPE_EVENT_OUT_INTR_MBOX | SPE_EVENT_SPE_STOPPED;
        event.spe = ctx;
        rc = spe_event_handler_register(evhandler, &event);

        /* start pthread */
```

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

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


        /* Get events. */
        cont = 1;
        while (cont) {
                event_count = spe_event_wait(evhandler, events, MAX_EVENTS, -1);
                printf("event_count %d\n", event_count);

                /* Handle events. */
                for (i = 0; i < event_count; i++) {
                        printf("event %d: %d\n", i, events[i].events);
                        if (events[i].events & SPE_EVENT_OUT_INTR_MBOX) {
                                printf("SPE_EVENT_OUT_INTR_MBOX\n");
                                rc = spe_out_intr_mbox_read(events[i].spe,
                                                            mbox_data,
                                                            COUNT,
                                                            SPE_MBOX_ANY_BLOCKING);
                        }
                        if (events[i].events & SPE_EVENT_SPE_STOPPED) {
                                printf("SPE_EVENT_SPE_STOPPED\n");
                                rc = spe_stop_info_read(events[i].spe, &stop_info);
                                printf("stop_reason: %d\n", stop_info.stop_reason);
                                cont = 0;
                        }
                }
        }

        pthread_join (pts, NULL);

        /* Destroy the handle. */
        spe_event_handler_destroy(evhandler);

        /* Destroy the context. */
        spe_context_destroy (ctx);

        return 0;
}
```

Chapter 2

libspe2 Data Structure Documentation

2.1 spe_context Struct Reference

Data Fields

- spe_program_handle_t handle
- struct spe_context_base_priv * base_private
- struct spe_context_event_priv * event_private

2.1.1 Detailed Description

SPE context The SPE context is one of the base data structures for the libspe2 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 libspe2 API calls.

Definition at line 64 of file libspe2-types.h.

2.1.2 Field Documentation

2.1.2.1 spe_program_handle_t handle

Definition at line 72 of file libspe2-types.h.

2.1.2.2 struct spe_context_base_priv* base_private [read]

Definition at line 76 of file libspe2-types.h.

2.1.2.3 struct spe_context_event_priv* event_private [read]

Definition at line 77 of file libspe2-types.h.

2.2 spe_event_data_t Union Reference

Data Fields

- void * ptr
- unsigned int u32
- unsigned long long u64

2.2.1 Detailed Description

spe_event_data_t User data to be associated with an event

Definition at line 143 of file libspe2-types.h.

2.2.2 Field Documentation

2.2.2.1 void* ptr

Definition at line 145 of file libspe2-types.h.

2.2.2.2 unsigned int u32

Definition at line 146 of file libspe2-types.h.

2.2.2.3 unsigned long long u64

Definition at line 147 of file libspe2-types.h.

2.3 spe_event_unit_t Struct Reference

Data Fields

- unsigned int events
- spe_context_ptr_t spe
- spe_event_data_t data

2.3.1 Detailed Description

spe_event_t

Definition at line 152 of file libspe2-types.h.

2.3.2 Field Documentation

2.3.2.1 unsigned int events

Definition at line 154 of file libspe2-types.h.

2.3.2.2 spe_context_ptr_t spe

Definition at line 155 of file libspe2-types.h.

2.3.2.3 spe_event_data_t data

Definition at line 156 of file libspe2-types.h.

2.4 spe_gang_context Struct Reference

Data Fields

- struct spe_gang_context_base_priv * base_private
- struct spe_gang_context_event_priv * event_private

2.4.1 Detailed Description

SPE gang context The SPE gang context is one of the base data structures for the libspe2 implementation. It holds all persistent information about a group of SPE contexts that should be treated as a gang, i.e., be execute 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 libspe2 API calls.

Definition at line 94 of file libspe2-types.h.

2.4.2 Field Documentation

2.4.2.1 struct spe_gang_context_base_priv* base_private [read]

Definition at line 99 of file libspe2-types.h.

2.4.2.2 struct spe_gang_context_event_priv* event_private [read]

Definition at line 100 of file libspe2-types.h.

2.5 spe_program_handle_t Struct Reference

Data Fields

- unsigned int handle_size
- void * elf_image
- void * toe_shadow

2.5.1 Detailed Description

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.

Definition at line 43 of file libspe2-types.h.

2.5.2 Field Documentation

2.5.2.1 unsigned int handle_size

Definition at line 49 of file libspe2-types.h.

2.5.2.2 void* elf_image

Definition at line 50 of file libspe2-types.h.

2.5.2.3 void* toe_shadow

Definition at line 51 of file libspe2-types.h.

2.6 spe_stop_info_t Struct Reference

Data Fields

- 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

2.6.1 Detailed Description

spe_stop_info_t

Definition at line 118 of file libspe2-types.h.

2.6.2 Field Documentation

2.6.2.1 unsigned int stop_reason

Definition at line 119 of file libspe2-types.h.

2.6.2.2 int spe_exit_code

Definition at line 121 of file libspe2-types.h.

2.6.2.3 int spe_signal_code

Definition at line 122 of file libspe2-types.h.

2.6.2.4 int spe_runtime_error

Definition at line 123 of file libspe2-types.h.

2.6.2.5 int spe_runtime_exception

Definition at line 124 of file libspe2-types.h.

2.6.2.6 int spe_runtime_fatal

Definition at line 125 of file libspe2-types.h.

2.6.2.7 int spe_callback_error

Definition at line 126 of file libspe2-types.h.

2.6.2.8 int spe_isolation_error

Definition at line 127 of file libspe2-types.h.

2.6.2.9 void* __reserved_ptr

Definition at line 129 of file libspe2-types.h.

2.6.2.10 unsigned long long __reserved_u64

Definition at line 130 of file libspe2-types.h.

2.6.2.11 union { ... } result

2.6.2.12 int spu_status

Definition at line 132 of file libspe2-types.h.

Chapter 3

libspe2 File Documentation

3.1 design.txt File Reference

3.2 libspe2-types.h File Reference

Data Structures

- struct spe_program_handle_t
- struct spe_context
- struct spe_gang_context
- struct spe_stop_info_t
- union spe_event_data_t
- struct spe_event_unit_t

Defines

- #define SPE_CFG_SIGNOTIFY1_OR 0x00000010
- #define SPE_CFG_SIGNOTIFY2_OR 0x00000020
- #define SPE_MAP_PS 0x00000040
- #define SPE_ISOLATE 0x00000080
- #define SPE_ISOLATE_EMULATE 0x00000100
- #define SPE_EVENTS_ENABLE 0x00001000
- #define SPE_AFFINITY_MEMORY 0x00002000
- #define SPE_EXIT 1
- #define SPE_STOP_AND_SIGNAL 2
- #define SPE_RUNTIME_ERROR 3
- #define SPE_RUNTIME_EXCEPTION 4
- #define SPE_RUNTIME_FATAL 5
- #define SPE_CALLBACK_ERROR 6
- #define SPE_ISOLATION_ERROR 7
- #define SPE_SPU_STOPPED_BY_STOP 0x02
- #define SPE_SPU_HALT 0x04
- #define SPE_SPU_WAITING_ON_CHANNEL 0x08
- #define SPE_SPU_SINGLE_STEP 0x10
- #define SPE_SPU_INVALID_INSTR 0x20
- #define SPE_SPU_INVALID_CHANNEL 0x40
- #define SPE_DMA_ALIGNMENT 0x0008
- #define SPE_DMA_SEGMENTATION 0x0020
- #define SPE_DMA_STORAGE 0x0040
- #define SIGSPE SIGURG
- #define SPE_EVENT_OUT_INTR_MBOX 0x00000001
- #define SPE_EVENT_IN_MBOX 0x00000002
- #define SPE_EVENT_TAG_GROUP 0x00000004
- #define SPE_EVENT_SPE_STOPPED 0x00000008
- #define SPE_EVENT_ALL_EVENTS
- #define SPE_MBOX_ALL_BLOCKING 1
- #define SPE_MBOX_ANY_BLOCKING 2
- #define SPE_MBOX_ANY_NONBLOCKING 3
- #define SPE_TAG_ALL 1
- #define SPE_TAG_ANY 2
- #define SPE_TAG_IMMEDIATE 3
- #define SPE_DEFAULT_ENTRY UINT_MAX
- #define SPE_RUN_USER_REGS 0x00000001

- #define SPE_NO_CALLBACKS 0x00000002
- #define SPE_CALLBACK_NEW 1
- #define SPE_CALLBACK_UPDATE 2
- #define SPE_COUNT_PHYSICAL_CPU_NODES 1
- #define SPE_COUNT_PHYSICAL_SPES 2
- #define SPE_COUNT_USABLE_SPES 3
- #define SPE_SIG_NOTIFY_REG_1 0x0001
- #define SPE_SIG_NOTIFY_REG_2 0x0002

Typedefs

- typedef struct spe_context * spe_context_ptr_t
- typedef struct spe_gang_context * spe_gang_context_ptr_t
- typedef void * spe_event_handler_ptr_t
- typedef int spe_event_handler_t

Enumerations

- enum ps_area {

  SPE_MSSYNC_AREA, SPE_MFC_COMMAND_AREA, SPE_CONTROL_AREA, SPE_SIG_-NOTIFY_1_AREA,

  SPE_SIG_NOTIFY_2_AREA }

3.2.1 Define Documentation

3.2.1.1 #define SIGSPE SIGURG

SIGSPE maps to SIGURG

Definition at line 219 of file libspe2-types.h.

3.2.1.2 #define SPE_AFFINITY_MEMORY 0x00002000

Definition at line 182 of file libspe2-types.h.

3.2.1.3 #define SPE_CALLBACK_ERROR 6

Definition at line 194 of file libspe2-types.h.

3.2.1.4 #define SPE_CALLBACK_NEW 1

Definition at line 260 of file libspe2-types.h.

3.2.1.5 #define SPE_CALLBACK_UPDATE 2

Definition at line 261 of file libspe2-types.h.

3.2.1.6 #define SPE_CFG_SIGNOTIFY1_OR 0x00000010

Flags for spe_context_create

Definition at line 176 of file libspe2-types.h.

3.2.1.7 #define SPE_CFG_SIGNOTIFY2_OR 0x00000020

Definition at line 177 of file libspe2-types.h.

3.2.1.8 #define SPE_COUNT_PHYSICAL_CPU_NODES 1

Definition at line 265 of file libspe2-types.h.

3.2.1.9 #define SPE_COUNT_PHYSICAL_SPES 2

Definition at line 266 of file libspe2-types.h.

3.2.1.10 #define SPE_COUNT_USABLE_SPES 3

Definition at line 267 of file libspe2-types.h.

3.2.1.11 #define SPE_DEFAULT_ENTRY UINT_MAX

Flags for _base_spe_context_run

Definition at line 253 of file libspe2-types.h.

3.2.1.12 #define SPE_DMA_ALIGNMENT 0x0008

Runtime exceptions

Definition at line 210 of file libspe2-types.h.

3.2.1.13 #define SPE_DMA_SEGMENTATION 0x0020

Definition at line 212 of file libspe2-types.h.

3.2.1.14 #define SPE_DMA_STORAGE 0x0040

Definition at line 213 of file libspe2-types.h.

3.2.1.15 #define SPE_EVENT_ALL_EVENTS

Value:

```
SPE_EVENT_OUT_INTR_MBOX | \
                                        SPE_EVENT_IN_MBOX | \
                                        SPE_EVENT_TAG_GROUP | \
                                        SPE_EVENT_SPE_STOPPED
```

Definition at line 229 of file libspe2-types.h.

3.2.1.16 #define SPE_EVENT_IN_MBOX 0x00000002

Definition at line 225 of file libspe2-types.h.

3.2.1.17 #define SPE_EVENT_OUT_INTR_MBOX 0x00000001

Supported SPE events

Definition at line 224 of file libspe2-types.h.

3.2.1.18 #define SPE_EVENT_SPE_STOPPED 0x00000008

Definition at line 227 of file libspe2-types.h.

3.2.1.19 #define SPE_EVENT_TAG_GROUP 0x00000004

Definition at line 226 of file libspe2-types.h.

3.2.1.20 #define SPE_EVENTS_ENABLE 0x00001000

Definition at line 181 of file libspe2-types.h.

3.2.1.21 #define SPE_EXIT 1

Symbolic constants for stop reasons as returned in spe_stop_info_t

Definition at line 189 of file libspe2-types.h.

3.2.1.22 #define SPE_ISOLATE 0x00000080

Definition at line 179 of file libspe2-types.h.

3.2.1.23 #define SPE_ISOLATE_EMULATE 0x00000100

Definition at line 180 of file libspe2-types.h.

3.2.1.24 #define SPE_ISOLATION_ERROR 7

Definition at line 195 of file libspe2-types.h.

3.2.1.25 #define SPE_MAP_PS 0x00000040

Definition at line 178 of file libspe2-types.h.

3.2.1.26 #define SPE_MBOX_ALL_BLOCKING 1

Behavior flags for mailbox read/write functions

Definition at line 237 of file libspe2-types.h.

3.2.1.27 #define SPE_MBOX_ANY_BLOCKING 2

Definition at line 238 of file libspe2-types.h.

3.2.1.28 #define SPE_MBOX_ANY_NONBLOCKING 3

Definition at line 239 of file libspe2-types.h.

3.2.1.29 #define SPE_NO_CALLBACKS 0x00000002

Definition at line 255 of file libspe2-types.h.

3.2.1.30 #define SPE_RUN_USER_REGS 0x00000001

Definition at line 254 of file libspe2-types.h.

3.2.1.31 #define SPE_RUNTIME_ERROR 3

Definition at line 191 of file libspe2-types.h.

3.2.1.32 #define SPE_RUNTIME_EXCEPTION 4

Definition at line 192 of file libspe2-types.h.

3.2.1.33 #define SPE_RUNTIME_FATAL 5

Definition at line 193 of file libspe2-types.h.

3.2.1.34 #define SPE_SIG_NOTIFY_REG_1 0x0001

Signal Targets

Definition at line 272 of file libspe2-types.h.

3.2.1.35 #define SPE_SIG_NOTIFY_REG_2 0x0002

Definition at line 273 of file libspe2-types.h.

3.2.1.36 #define SPE_SPU_HALT 0x04

Definition at line 201 of file libspe2-types.h.

3.2.1.37 #define SPE_SPU_INVALID_CHANNEL 0x40

Definition at line 205 of file libspe2-types.h.

3.2.1.38 #define SPE_SPU_INVALID_INSTR 0x20

Definition at line 204 of file libspe2-types.h.

3.2.1.39 #define SPE_SPU_SINGLE_STEP 0x10

Definition at line 203 of file libspe2-types.h.

3.2.1.40 #define SPE_SPU_STOPPED_BY_STOP 0x02

Runtime errors

Definition at line 200 of file libspe2-types.h.

3.2.1.41 #define SPE_SPU_WAITING_ON_CHANNEL 0x08

Definition at line 202 of file libspe2-types.h.

3.2.1.42 #define SPE_STOP_AND_SIGNAL 2

Definition at line 190 of file libspe2-types.h.

3.2.1.43 #define SPE_TAG_ALL 1

Behavior flags tag status functions

Definition at line 245 of file libspe2-types.h.

3.2.1.44 #define SPE_TAG_ANY 2

Definition at line 246 of file libspe2-types.h.

3.2.1.45 #define SPE_TAG_IMMEDIATE 3

Definition at line 247 of file libspe2-types.h.

3.2.2 Typedef Documentation

3.2.2.1 typedef struct spe_context* spe_context_ptr_t

spe_context_ptr_t This pointer serves as the identifier for a specific SPE context throughout the API (where needed)

Definition at line 83 of file libspe2-types.h.

3.2.2.2 typedef void* spe_event_handler_ptr_t

Definition at line 159 of file libspe2-types.h.

3.2.2.3 typedef int spe_event_handler_t

Definition at line 160 of file libspe2-types.h.

3.2.2.4 typedef struct spe_gang_context* spe_gang_context_ptr_t

spe_gang_context_ptr_t This pointer serves as the identifier for a specific SPE gang context throughout the API (where needed)

Definition at line 106 of file libspe2-types.h.

3.2.3 Enumeration Type Documentation

3.2.3.1 enum ps_area

Enumerator:

SPE_MSSYNC_AREA

SPE_MFC_COMMAND_AREA

SPE_CONTROL_AREA

SPE_SIG_NOTIFY_1_AREA

SPE_SIG_NOTIFY_2_AREA

Definition at line 171 of file libspe2-types.h.

3.3 libspe2.h File Reference

Functions

- spe_context_ptr_t spe_context_create (unsigned int flags, spe_gang_context_ptr_t gang)
- spe_context_ptr_t spe_context_create_affinity (unsigned int flags, spe_context_ptr_t affinity_neighbor, spe_gang_context_ptr_t gang)
- int spe_context_destroy (spe_context_ptr_t spe)
- spe_gang_context_ptr_t spe_gang_context_create (unsigned int flags)
- int spe_gang_context_destroy (spe_gang_context_ptr_t gang)
- spe_program_handle_t * spe_image_open (const char *filename)
- int spe_image_close (spe_program_handle_t *program)
- int spe_program_load (spe_context_ptr_t spe, spe_program_handle_t *program)
- int spe_context_run (spe_context_ptr_t spe, unsigned int *entry, unsigned int runflags, void *argp, void *envp, spe_stop_info_t *stopinfo)
- int spe_stop_info_read (spe_context_ptr_t spe, spe_stop_info_t *stopinfo)
- spe_event_handler_ptr_t spe_event_handler_create (void)
- int spe_event_handler_destroy (spe_event_handler_ptr_t evhandler)
- int spe_event_handler_register (spe_event_handler_ptr_t evhandler, spe_event_unit_t *event)
- int spe_event_handler_deregister (spe_event_handler_ptr_t evhandler, spe_event_unit_t *event)
- int spe_event_wait (spe_event_handler_ptr_t evhandler, spe_event_unit_t *events, int max_events, int timeout)
- int spe_mfcio_put (spe_context_ptr_t spe, unsigned int ls, 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 ls, 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 ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)
- int spe_mfcio_get (spe_context_ptr_t spe, unsigned int ls, 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 ls, 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 ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)
- int spe_mfcio_tag_status_read (spe_context_ptr_t spe, unsigned int mask, unsigned int behavior, unsigned int *tag_status)
- int spe_out_mbox_read (spe_context_ptr_t spe, unsigned int *mbox_data, int count)
- int spe_out_mbox_status (spe_context_ptr_t spe)
- int spe_in_mbox_write (spe_context_ptr_t spe, unsigned int *mbox_data, int count, unsigned int behavior)
- int spe_in_mbox_status (spe_context_ptr_t spe)
- int spe_out_intr_mbox_read (spe_context_ptr_t spe, unsigned int *mbox_data, int count, unsigned int behavior)
- int spe_out_intr_mbox_status (spe_context_ptr_t spe)
- int spe_mssync_start (spe_context_ptr_t spe)
- int spe_mssync_status (spe_context_ptr_t spe)
- int spe_signal_write (spe_context_ptr_t spe, unsigned int signal_reg, unsigned int data)
- void * spe_ls_area_get (spe_context_ptr_t spe)
- int spe_ls_size_get (spe_context_ptr_t spe)
- void * spe_ps_area_get (spe_context_ptr_t spe, enum ps_area area)
- int spe_callback_handler_register (void *handler, unsigned int callnum, unsigned int mode)

- int spe_callback_handler_deregister (unsigned int callnum)

- void * spe_callback_handler_query (unsigned int callnum)

- int spe_cpu_info_get (int info_requested, int cpu_node)

3.3.1 Function Documentation

3.3.1.1 int spe_callback_handler_deregister (unsigned int *callnum*)

3.3.1.2 void* spe_callback_handler_query (unsigned int *callnum*)

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

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

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

3.3.1.6 int spe_context_destroy (spe_context_ptr_t *spe*)

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

3.3.1.8 int spe_cpu_info_get (int *info_requested*, int *cpu_node*)

3.3.1.9 spe_event_handler_ptr_t spe_event_handler_create (void)

3.3.1.10 int spe_event_handler_deregister (spe_event_handler_ptr_t *evhandler*, spe_event_unit_t * *event*)

3.3.1.11 int spe_event_handler_destroy (spe_event_handler_ptr_t *evhandler*)

3.3.1.12 int spe_event_handler_register (spe_event_handler_ptr_t *evhandler*, spe_event_unit_t * *event*)

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

3.3.1.14 spe_gang_context_ptr_t spe_gang_context_create (unsigned int *flags*)

3.3.1.15 int spe_gang_context_destroy (spe_gang_context_ptr_t *gang*)

3.3.1.16 int spe_image_close (spe_program_handle_t * *program*)

3.3.1.17 spe_program_handle_t* spe_image_open (const char * *filename*)

3.3.1.18 int spe_in_mbox_status (spe_context_ptr_t *spe*)

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

3.3.1.20 void* spe_ls_area_get (spe_context_ptr_t *spe*)

3.3.1.21 int spe_ls_size_get (spe_context_ptr_t *spe*)

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

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

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

3.3.1.25 int spe_mfcio_put (spe_context_ptr_t *spe*, unsigned int *ls*, void * *ea*, unsigned int *size*,

Index