

---

# SPE Runtime Management Library

---

Version 2.0

September 29, 2007

©Copyright International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation 2003, 2004, 2005, 2006

All Rights Reserved

Printed in the United States of America June 2006

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

IBM PowerPC

IBM Logo PowerPC Architecture

Other company, product, and service names may be trademarks or service marks of others. All information contained in this document is subject to change without notice. The products described in this document are NOT intended for use in applications such as implantation, life support, or other hazardous uses where malfunction could result in death, bodily injury, or catastrophic property damage. The information contained in this document does not affect or change IBM product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of IBM or third parties. All information contained in this document was obtained in specific environments, and is presented as an illustration. The results obtained in other operating environments may vary.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN “AS IS” BASIS. In no event will IBM be liable for damages arising directly or indirectly from any use of the information contained in this document.

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](http://ibm.com)

The IBM semiconductor solutions home page can be found at [ibm.com/chips](http://ibm.com/chips)

June 15, 2006

# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Overview</b>                                       | <b>1</b>  |
| 1.1      | Terminology . . . . .                                 | 1         |
| 1.2      | Usage Scenarios . . . . .                             | 1         |
| <b>2</b> | <b>libspe2 Data Structure Documentation</b>           | <b>5</b>  |
| 2.1      | addr64 Union Reference . . . . .                      | 5         |
| 2.2      | fd_attr Struct Reference . . . . .                    | 6         |
| 2.3      | image_handle Struct Reference . . . . .               | 7         |
| 2.4      | mfc_command_parameter_area Struct Reference . . . . . | 8         |
| 2.5      | spe_context Struct Reference . . . . .                | 10        |
| 2.6      | spe_context_base_priv Struct Reference . . . . .      | 12        |
| 2.7      | spe_context_event_priv_t Struct Reference . . . . .   | 16        |
| 2.8      | spe_context_info Struct Reference . . . . .           | 18        |
| 2.9      | spe_event_data_t Union Reference . . . . .            | 19        |
| 2.10     | spe_event_unit_t Struct Reference . . . . .           | 20        |
| 2.11     | spe_gang_context Struct Reference . . . . .           | 22        |
| 2.12     | spe_gang_context_base_priv Struct Reference . . . . . | 23        |
| 2.13     | spe_ld_info Struct Reference . . . . .                | 24        |
| 2.14     | spe_mfc_command_area_t Struct Reference . . . . .     | 25        |
| 2.15     | spe_mssync_area_t Struct Reference . . . . .          | 28        |
| 2.16     | spe_program_handle_t Struct Reference . . . . .       | 29        |
| 2.17     | spe_reg128 Struct Reference . . . . .                 | 30        |
| 2.18     | spe_sig_notify_1_area_t Struct Reference . . . . .    | 31        |
| 2.19     | spe_sig_notify_2_area_t Struct Reference . . . . .    | 32        |
| 2.20     | spe_spu_control_area_t Struct Reference . . . . .     | 33        |
| 2.21     | spe_stop_info_t Struct Reference . . . . .            | 35        |
| <b>3</b> | <b>libspe2 File Documentation</b>                     | <b>37</b> |
| 3.1      | accessors.c File Reference . . . . .                  | 37        |

|      |  |     |
|------|--|-----|
| 3.2  | cbea_map.h File Reference . . . . .      | 41  |
| 3.3  | create.c File Reference . . . . .        | 42  |
| 3.4  | create.h File Reference . . . . .        | 49  |
| 3.5  | design.txt File Reference . . . . .      | 51  |
| 3.6  | dma.c File Reference . . . . .           | 52  |
| 3.7  | dma.h File Reference . . . . .           | 58  |
| 3.8  | elf_loader.c File Reference . . . . .    | 60  |
| 3.9  | elf_loader.h File Reference . . . . .    | 65  |
| 3.10 | handler_utils.h File Reference . . . . . | 70  |
| 3.11 | image.c File Reference . . . . .         | 73  |
| 3.12 | info.c File Reference . . . . .          | 76  |
| 3.13 | info.h File Reference . . . . .          | 79  |
| 3.14 | lib_builtin.c File Reference . . . . .   | 82  |
| 3.15 | lib_builtin.h File Reference . . . . .   | 86  |
| 3.16 | libspe2-types.h File Reference . . . . . | 88  |
| 3.17 | libspe2.h File Reference . . . . .       | 97  |
| 3.18 | load.c File Reference . . . . .          | 101 |
| 3.19 | mbox.c File Reference . . . . .          | 104 |
| 3.20 | mbox.h File Reference . . . . .          | 111 |
| 3.21 | run.c File Reference . . . . .           | 112 |
| 3.22 | spe_event.c File Reference . . . . .     | 118 |
| 3.23 | spebase.h File Reference . . . . .       | 128 |
| 3.24 | speevent.h File Reference . . . . .      | 165 |

# 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 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:

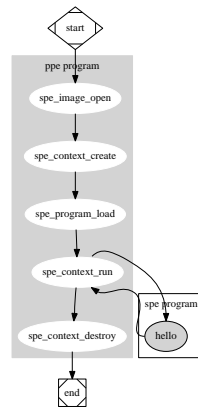


Figure 1.1: Simple program

Here is the same sample with some error checking:

### 1.2.2 Multi-threaded sample

This illustrates a threaded sample using the pthread library:

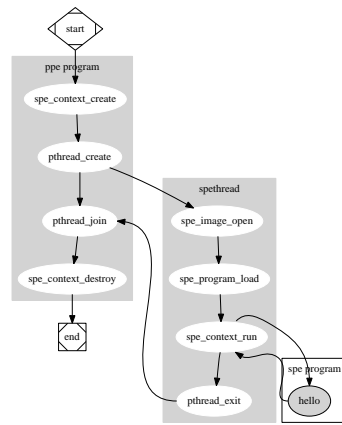


Figure 1.2: Simple pthread program

Here is the same sample with some error checking:

### 1.2.3 Problem state mapping samples

This illustrates accessing the MFC Local Store Address Register.

### 1.2.4 Event samples

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

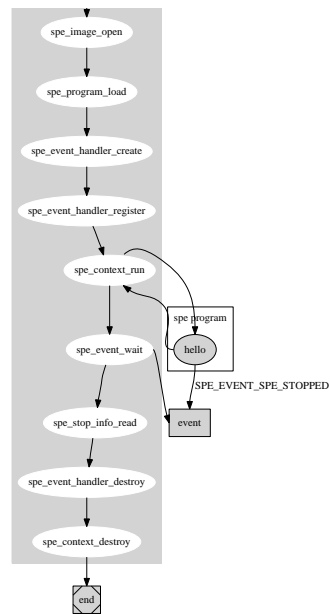


Figure 1.3: Simple event program

Events are more useful in multithreaded environments:



## Chapter 2

# libspe2 Data Structure Documentation

### 2.1 addr64 Union Reference

```
#include <elf_loader.h>
```

#### Data Fields

- unsigned long long [ull](#)
- unsigned int [ui](#) [2]

#### 2.1.1 Detailed Description

Definition at line 28 of file elf\_loader.h.

#### 2.1.2 Field Documentation

##### 2.1.2.1 unsigned long long ull

Definition at line 30 of file elf\_loader.h.

Referenced by `_base_spe_context_run()`.

##### 2.1.2.2 unsigned int ui[2]

Definition at line 31 of file elf\_loader.h.

Referenced by `_base_spe_context_run()`.

The documentation for this union was generated from the following file:

- [elf\\_loader.h](#)

## 2.2 fd\_attr Struct Reference

### Data Fields

- const char \* [name](#)
- int [mode](#)

### 2.2.1 Detailed Description

Definition at line 37 of file create.c.

### 2.2.2 Field Documentation

#### 2.2.2.1 const char\* name

Definition at line 38 of file create.c.

Referenced by `_base_spe_open_if_closed()`.

#### 2.2.2.2 int mode

Definition at line 39 of file create.c.

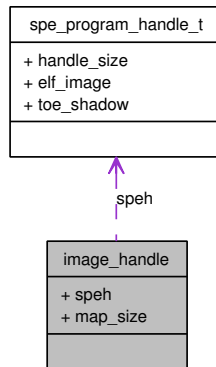
Referenced by `_base_spe_open_if_closed()`.

The documentation for this struct was generated from the following file:

- [create.c](#)

## 2.3 image\_handle Struct Reference

Collaboration diagram for image\_handle:



### Data Fields

- [spe\\_program\\_handle\\_t speh](#)
- unsigned int [map\\_size](#)

#### 2.3.1 Detailed Description

Definition at line 32 of file `image.c`.

#### 2.3.2 Field Documentation

##### 2.3.2.1 spe\_program\_handle\_t speh

Definition at line 33 of file `image.c`.

Referenced by `_base_spe_image_close()`, and `_base_spe_image_open()`.

##### 2.3.2.2 unsigned int map\_size

Definition at line 34 of file `image.c`.

Referenced by `_base_spe_image_close()`, and `_base_spe_image_open()`.

The documentation for this struct was generated from the following file:

- [image.c](#)

## 2.4 mfc\_command\_parameter\_area Struct Reference

```
#include <dma.h>
```

### Data Fields

- [uint32\\_t pad](#)
- [uint32\\_t lsa](#)
- [uint64\\_t ea](#)
- [uint16\\_t size](#)
- [uint16\\_t tag](#)
- [uint16\\_t class](#)
- [uint16\\_t cmd](#)

### 2.4.1 Detailed Description

Definition at line 27 of file dma.h.

### 2.4.2 Field Documentation

#### 2.4.2.1 [uint32\\_t pad](#)

Definition at line 28 of file dma.h.

#### 2.4.2.2 [uint32\\_t lsa](#)

Definition at line 29 of file dma.h.

#### 2.4.2.3 [uint64\\_t ea](#)

Definition at line 30 of file dma.h.

#### 2.4.2.4 [uint16\\_t size](#)

Definition at line 31 of file dma.h.

#### 2.4.2.5 [uint16\\_t tag](#)

Definition at line 32 of file dma.h.

#### 2.4.2.6 [uint16\\_t class](#)

Definition at line 33 of file dma.h.

### 2.4.2.7 uint16\_t cmd

Definition at line 34 of file dma.h.

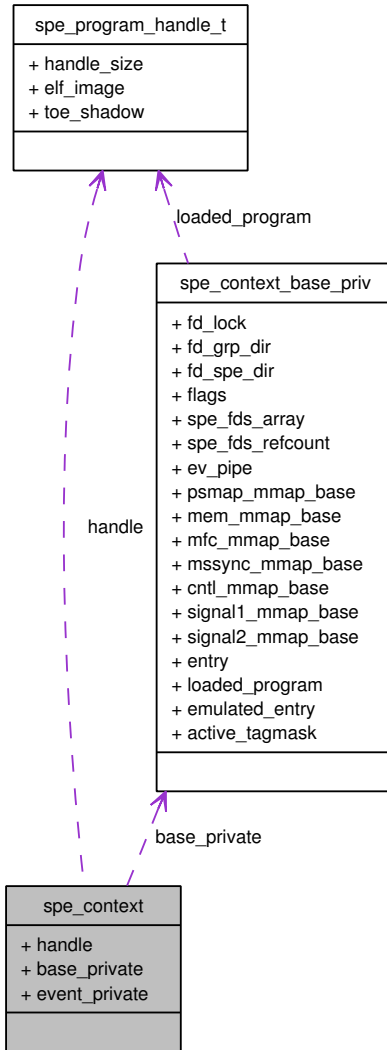
The documentation for this struct was generated from the following file:

- [dma.h](#)

## 2.5 spe\_context Struct Reference

```
#include <libspe2-types.h>
```

Collaboration diagram for spe\_context:



### Data Fields

- [spe\\_program\\_handle\\_t](#) `handle`
- struct [spe\\_context\\_base\\_priv](#) \* `base_private`
- struct [spe\\_context\\_event\\_priv](#) \* `event_private`

#### 2.5.1 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.

## 2.5.2 Field Documentation

### 2.5.2.1 spe\_program\_handle\_t handle

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

### 2.5.2.2 struct spe\_context\_base\_priv\* base\_private [read]

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

Referenced by \_\_base\_spe\_spe\_dir\_get(), \_\_base\_spe\_stop\_event\_source\_get(), \_\_base\_spe\_stop\_event\_target\_get(), \_base\_spe\_close\_if\_open(), \_base\_spe\_context\_create(), \_base\_spe\_context\_lock(), \_base\_spe\_context\_run(), \_base\_spe\_context\_unlock(), \_base\_spe\_handle\_library\_callback(), \_base\_spe\_in\_mbox\_status(), \_base\_spe\_in\_mbox\_write(), \_base\_spe\_ls\_area\_get(), \_base\_spe\_mfcio\_tag\_status\_read(), \_base\_spe\_mssync\_start(), \_base\_spe\_mssync\_status(), \_base\_spe\_open\_if\_closed(), \_base\_spe\_out\_intr\_mbox\_status(), \_base\_spe\_out\_mbox\_read(), \_base\_spe\_out\_mbox\_status(), \_base\_spe\_program\_load(), \_base\_spe\_program\_load\_complete(), \_base\_spe\_ps\_area\_get(), \_base\_spe\_signal\_write(), and \_event\_spe\_event\_handler\_register().

### 2.5.2.3 struct spe\_context\_event\_priv\* event\_private [read]

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

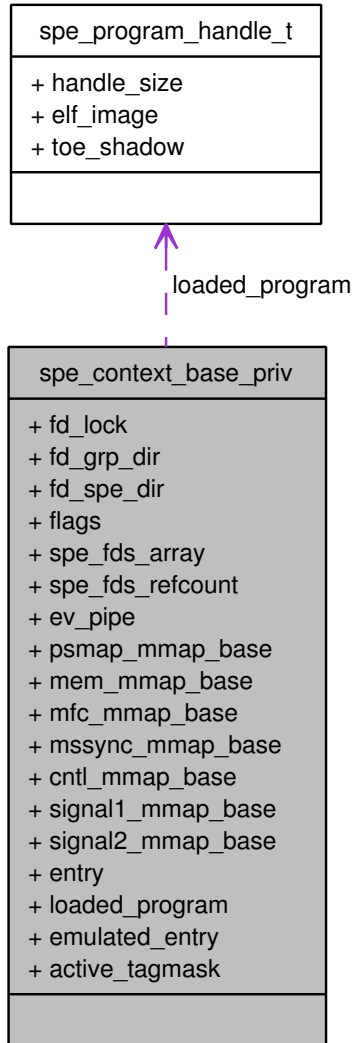
The documentation for this struct was generated from the following file:

- [libspe2-types.h](#)

## 2.6 spe\_context\_base\_priv Struct Reference

```
#include <spebase.h>
```

Collaboration diagram for spe\_context\_base\_priv:



### Data Fields

- pthread\_mutex\_t `fd_lock` [NUM\_MBOX\_FDS]
- int `fd_grp_dir`
- int `fd_spe_dir`
- unsigned int `flags`
- int `spe_fds_array` [NUM\_MBOX\_FDS]
- int `spe_fds_refcount` [NUM\_MBOX\_FDS]
- int `ev_pipe` [2]
- void \* `psmap_mmap_base`
- void \* `mem_mmap_base`



- void \* [mfc\\_mmap\\_base](#)
- void \* [mssync\\_mmap\\_base](#)
- void \* [cntl\\_mmap\\_base](#)
- void \* [signal1\\_mmap\\_base](#)
- void \* [signal2\\_mmap\\_base](#)
- int [entry](#)
- [spe\\_program\\_handle\\_t](#) \* [loaded\\_program](#)
- int [emulated\\_entry](#)
- int [active\\_tagmask](#)

### 2.6.1 Detailed Description

Definition at line 61 of file `spebase.h`.

### 2.6.2 Field Documentation

#### 2.6.2.1 pthread\_mutex\_t fd\_lock[NUM\_MBOX\_FDS]

Definition at line 65 of file `spebase.h`.

Referenced by `_base_spe_context_create()`, `_base_spe_context_lock()`, and `_base_spe_context_unlock()`.

#### 2.6.2.2 int fd\_grp\_dir

Definition at line 68 of file `spebase.h`.

#### 2.6.2.3 int fd\_spe\_dir

Definition at line 71 of file `spebase.h`.

Referenced by `__base_spe_spe_dir_get()`, `_base_spe_context_create()`, `_base_spe_context_run()`, `_base_spe_open_if_closed()`, and `_base_spe_program_load_complete()`.

#### 2.6.2.4 unsigned int flags

Definition at line 74 of file `spebase.h`.

Referenced by `_base_spe_context_create()`, `_base_spe_context_run()`, `_base_spe_handle_library_callback()`, `_base_spe_in_mbox_status()`, `_base_spe_in_mbox_write()`, `_base_spe_mfcio_tag_status_read()`, `_base_spe_mssync_start()`, `_base_spe_mssync_status()`, `_base_spe_out_intr_mbox_status()`, `_base_spe_out_mbox_read()`, `_base_spe_out_mbox_status()`, `_base_spe_program_load()`, `_base_spe_signal_write()`, and `_event_spe_event_handler_register()`.

#### 2.6.2.5 int spe\_fds\_array[NUM\_MBOX\_FDS]

Definition at line 77 of file `spebase.h`.

Referenced by `_base_spe_close_if_open()`, `_base_spe_context_create()`, and `_base_spe_open_if_closed()`.

**2.6.2.6 int spe\_fds\_refcount[NUM\_MBOX\_FDS]**

Definition at line 78 of file spebase.h.

Referenced by `_base_spe_close_if_open()`, and `_base_spe_open_if_closed()`.

**2.6.2.7 int ev\_pipe[2]**

Definition at line 81 of file spebase.h.

Referenced by `__base_spe_stop_event_source_get()`, and `__base_spe_stop_event_target_get()`.

**2.6.2.8 void\* psmmap\_mmap\_base**

Definition at line 84 of file spebase.h.

Referenced by `_base_spe_context_create()`.

**2.6.2.9 void\* mem\_mmap\_base**

Definition at line 85 of file spebase.h.

Referenced by `_base_spe_context_create()`, `_base_spe_context_run()`, `_base_spe_handle_library_callback()`, `_base_spe_ls_area_get()`, and `_base_spe_program_load()`.

**2.6.2.10 void\* mfc\_mmap\_base**

Definition at line 86 of file spebase.h.

Referenced by `_base_spe_context_create()`, and `_base_spe_ps_area_get()`.

**2.6.2.11 void\* mssync\_mmap\_base**

Definition at line 87 of file spebase.h.

Referenced by `_base_spe_context_create()`, `_base_spe_mssync_start()`, `_base_spe_mssync_status()`, and `_base_spe_ps_area_get()`.

**2.6.2.12 void\* cntl\_mmap\_base**

Definition at line 88 of file spebase.h.

Referenced by `_base_spe_context_create()`, `_base_spe_in_mbox_status()`, `_base_spe_out_intr_mbox_status()`, `_base_spe_out_mbox_status()`, and `_base_spe_ps_area_get()`.

**2.6.2.13 void\* signal1\_mmap\_base**

Definition at line 89 of file spebase.h.

Referenced by `_base_spe_context_create()`, `_base_spe_ps_area_get()`, and `_base_spe_signal_write()`.

**2.6.2.14 void\* signal2\_mmap\_base**

Definition at line 90 of file spebase.h.

Referenced by `_base_spe_context_create()`, `_base_spe_ps_area_get()`, and `_base_spe_signal_write()`.

**2.6.2.15 int entry**

Definition at line 93 of file spebase.h.

Referenced by `_base_spe_context_run()`, and `_base_spe_program_load()`.

**2.6.2.16 spe\_program\_handle\_t\* loaded\_program**

Definition at line 99 of file spebase.h.

Referenced by `_base_spe_context_create()`, `_base_spe_program_load()`, and `_base_spe_program_load_complete()`.

**2.6.2.17 int emulated\_entry**

Definition at line 103 of file spebase.h.

Referenced by `_base_spe_context_run()`, and `_base_spe_program_load()`.

**2.6.2.18 int active\_tagmask**

Definition at line 108 of file spebase.h.

Referenced by `_base_spe_mfcio_tag_status_read()`.

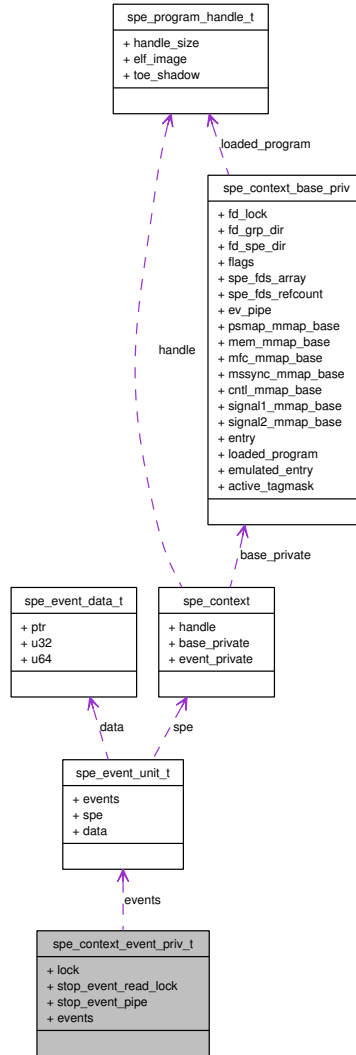
The documentation for this struct was generated from the following file:

- [spebase.h](#)

## 2.7 spe\_context\_event\_priv\_t Struct Reference

```
#include <speevent.h>
```

Collaboration diagram for spe\_context\_event\_priv\_t:



### Data Fields

- pthread\_mutex\_t [lock](#)
- pthread\_mutex\_t [stop\\_event\\_read\\_lock](#)
- int [stop\\_event\\_pipe](#) [2]
- [spe\\_event\\_unit\\_t events](#) [\_\_\_NUM\_SPE\_EVENT\_TYPES]

#### 2.7.1 Detailed Description

Definition at line 35 of file `speevent.h`.

## 2.7.2 Field Documentation

### 2.7.2.1 pthread\_mutex\_t lock

Definition at line 37 of file speevent.h.

Referenced by `_event_spe_context_finalize()`, and `_event_spe_context_initialize()`.

### 2.7.2.2 pthread\_mutex\_t stop\_event\_read\_lock

Definition at line 38 of file speevent.h.

Referenced by `_event_spe_context_finalize()`, `_event_spe_context_initialize()`, and `_event_spe_stop_info_read()`.

### 2.7.2.3 int stop\_event\_pipe[2]

Definition at line 39 of file speevent.h.

Referenced by `_event_spe_context_finalize()`, `_event_spe_context_initialize()`, `_event_spe_context_run()`, `_event_spe_event_handler_deregister()`, `_event_spe_event_handler_register()`, and `_event_spe_stop_info_read()`.

### 2.7.2.4 spe\_event\_unit\_t events[\_\_NUM\_SPE\_EVENT\_TYPES]

Definition at line 40 of file speevent.h.

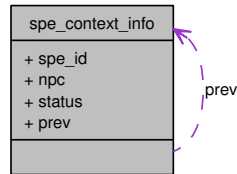
Referenced by `_event_spe_context_initialize()`, `_event_spe_event_handler_deregister()`, and `_event_spe_event_handler_register()`.

The documentation for this struct was generated from the following file:

- [speevent.h](#)

## 2.8 spe\_context\_info Struct Reference

Collaboration diagram for spe\_context\_info:



### Data Fields

- int [spe\\_id](#)
- unsigned int [npc](#)
- unsigned int [status](#)
- struct [spe\\_context\\_info](#) \* [prev](#)

### 2.8.1 Detailed Description

Definition at line 39 of file run.c.

### 2.8.2 Field Documentation

#### 2.8.2.1 int spe\_id

Definition at line 40 of file run.c.

Referenced by `_base_spe_context_run()`.

#### 2.8.2.2 unsigned int npc

Definition at line 41 of file run.c.

Referenced by `_base_spe_context_run()`.

#### 2.8.2.3 unsigned int status

Definition at line 42 of file run.c.

Referenced by `_base_spe_context_run()`.

#### 2.8.2.4 struct spe\_context\_info\* prev [read]

Definition at line 43 of file run.c.

Referenced by `_base_spe_context_run()`.

The documentation for this struct was generated from the following file:

- [run.c](#)

## 2.9 `spe_event_data_t` Union Reference

```
#include <libspe2-types.h>
```

### Data Fields

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

### 2.9.1 Detailed Description

`spe_event_data_t` User data to be associated with an event

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

### 2.9.2 Field Documentation

#### 2.9.2.1 void\* ptr

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

Referenced by `_event_spe_event_handler_register()`.

#### 2.9.2.2 unsigned int u32

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

#### 2.9.2.3 unsigned long long u64

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

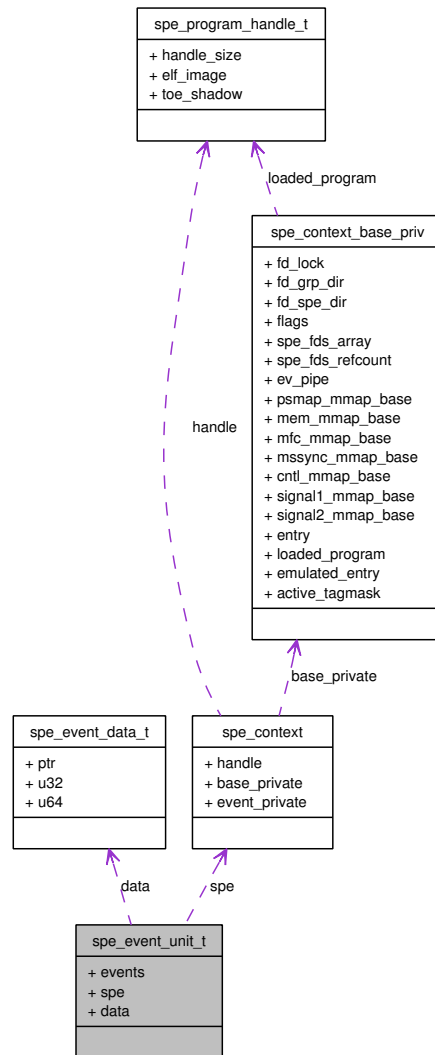
The documentation for this union was generated from the following file:

- `libspe2-types.h`

## 2.10 spe\_event\_unit\_t Struct Reference

```
#include <libspe2-types.h>
```

Collaboration diagram for spe\_event\_unit\_t:



### Data Fields

- unsigned int [events](#)
- [spe\\_context\\_ptr\\_t](#) `spe`
- [spe\\_event\\_data\\_t](#) `data`

#### 2.10.1 Detailed Description

`spe_event_t`

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



## 2.10.2 Field Documentation

### 2.10.2.1 `unsigned int` events

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

Referenced by `_event_spe_event_handler_deregister()`, and `_event_spe_event_handler_register()`.

### 2.10.2.2 `spe_context_ptr_t` spe

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

Referenced by `_event_spe_context_initialize()`, `_event_spe_event_handler_deregister()`, `_event_spe_event_handler_register()`, and `_event_spe_event_wait()`.

### 2.10.2.3 `spe_event_data_t` data

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

Referenced by `_event_spe_event_handler_register()`.

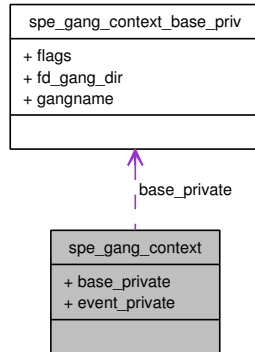
The documentation for this struct was generated from the following file:

- [libspe2-types.h](#)

## 2.11 spe\_gang\_context Struct Reference

```
#include <libspe2-types.h>
```

Collaboration diagram for spe\_gang\_context:



### Data Fields

- struct [spe\\_gang\\_context\\_base\\_priv](#) \* [base\\_private](#)
- struct [spe\\_gang\\_context\\_event\\_priv](#) \* [event\\_private](#)

#### 2.11.1 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.

#### 2.11.2 Field Documentation

##### 2.11.2.1 struct [spe\\_gang\\_context\\_base\\_priv](#)\* [base\\_private](#) [read]

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

Referenced by [\\_base\\_spe\\_context\\_create\(\)](#), and [\\_base\\_spe\\_gang\\_context\\_create\(\)](#).

##### 2.11.2.2 struct [spe\\_gang\\_context\\_event\\_priv](#)\* [event\\_private](#) [read]

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

The documentation for this struct was generated from the following file:

- [libspe2-types.h](#)

## 2.12 spe\_gang\_context\_base\_priv Struct Reference

```
#include <spebase.h>
```

### Data Fields

- unsigned int [flags](#)
- int [fd\\_gang\\_dir](#)
- char [gangname](#) [256]

### 2.12.1 Detailed Description

[spe\\_context](#): This holds the persistent information of a SPU instance it is created by `spe_create_context()`

Definition at line 150 of file `spebase.h`.

### 2.12.2 Field Documentation

#### 2.12.2.1 unsigned int flags

Definition at line 153 of file `spebase.h`.

#### 2.12.2.2 int fd\_gang\_dir

Definition at line 156 of file `spebase.h`.

#### 2.12.2.3 char gangname[256]

Definition at line 158 of file `spebase.h`.

Referenced by `_base_spe_context_create()`, and `_base_spe_gang_context_create()`.

The documentation for this struct was generated from the following file:

- [spebase.h](#)

## 2.13 spe\_ld\_info Struct Reference

```
#include <elf_loader.h>
```

### Data Fields

- unsigned int [entry](#)

### 2.13.1 Detailed Description

Definition at line 34 of file elf\_loader.h.

### 2.13.2 Field Documentation

#### 2.13.2.1 unsigned int entry

Definition at line 36 of file elf\_loader.h.

Referenced by `_base_spe_load_spe_elf()`, and `_base_spe_program_load()`.

The documentation for this struct was generated from the following file:

- [elf\\_loader.h](#)

## 2.14 spe\_mfc\_command\_area\_t Struct Reference

```
#include <cbea_map.h>
```

### Data Fields

- 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](#)

### 2.14.1 Detailed Description

Definition at line 34 of file cbea\_map.h.

### 2.14.2 Field Documentation

#### 2.14.2.1 unsigned char reserved\_0\_3[4]

Definition at line 35 of file cbea\_map.h.

#### 2.14.2.2 unsigned int MFC\_LSA

Definition at line 36 of file cbea\_map.h.

#### 2.14.2.3 unsigned int MFC\_EAH

Definition at line 37 of file cbea\_map.h.

#### 2.14.2.4 unsigned int MFC\_EAL

Definition at line 38 of file cbea\_map.h.

**2.14.2.5 unsigned int MFC\_Size\_Tag**

Definition at line 39 of file cbea\_map.h.

**2.14.2.6 unsigned int MFC\_ClassID\_CMD**

Definition at line 41 of file cbea\_map.h.

**2.14.2.7 unsigned int MFC\_CMDStatus**

Definition at line 42 of file cbea\_map.h.

**2.14.2.8 union { ... }****2.14.2.9 unsigned char reserved\_18\_103[236]**

Definition at line 44 of file cbea\_map.h.

**2.14.2.10 unsigned int MFC\_QStatus**

Definition at line 45 of file cbea\_map.h.

**2.14.2.11 unsigned char reserved\_108\_203[252]**

Definition at line 46 of file cbea\_map.h.

**2.14.2.12 unsigned int Prxy\_QueryType**

Definition at line 47 of file cbea\_map.h.

**2.14.2.13 unsigned char reserved\_208\_21B[20]**

Definition at line 48 of file cbea\_map.h.

**2.14.2.14 unsigned int Prxy\_QueryMask**

Definition at line 49 of file cbea\_map.h.

**2.14.2.15 unsigned char reserved\_220\_22B[12]**

Definition at line 50 of file cbea\_map.h.

**2.14.2.16 unsigned int Prxy\_TagStatus**

Definition at line 51 of file cbea\_map.h.

The documentation for this struct was generated from the following file:

- [cbea\\_map.h](#)

## 2.15 spe\_mssync\_area\_t Struct Reference

```
#include <cbea_map.h>
```

### Data Fields

- unsigned int [MFC\\_MSSync](#)

### 2.15.1 Detailed Description

Definition at line 30 of file cbea\_map.h.

### 2.15.2 Field Documentation

#### 2.15.2.1 unsigned int MFC\_MSSync

Definition at line 31 of file cbea\_map.h.

The documentation for this struct was generated from the following file:

- [cbea\\_map.h](#)



## 2.16 spe\_program\_handle\_t Struct Reference

```
#include <libspe2-types.h>
```

### Data Fields

- unsigned int [handle\\_size](#)
- void \* [elf\\_image](#)
- void \* [toe\\_shadow](#)

### 2.16.1 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`.

### 2.16.2 Field Documentation

#### 2.16.2.1 unsigned int handle\_size

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

Referenced by `_base_spe_image_open()`.

#### 2.16.2.2 void\* elf\_image

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

Referenced by `_base_spe_image_close()`, `_base_spe_image_open()`, `_base_spe_load_spe_elf()`, `_base_spe_parse_isolated_elf()`, `_base_spe_program_load_complete()`, `_base_spe_toe_ear()`, and `_base_spe_verify_spe_elf_image()`.

#### 2.16.2.3 void\* toe\_shadow

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

Referenced by `_base_spe_image_close()`, `_base_spe_image_open()`, and `_base_spe_toe_ear()`.

The documentation for this struct was generated from the following file:

- [libspe2-types.h](#)

## 2.17 spe\_reg128 Struct Reference

```
#include <handler_utils.h>
```

### Data Fields

- unsigned int [slot](#) [4]

### 2.17.1 Detailed Description

Definition at line 23 of file handler\_utils.h.

### 2.17.2 Field Documentation

#### 2.17.2.1 unsigned int slot[4]

Definition at line 24 of file handler\_utils.h.

The documentation for this struct was generated from the following file:

- [handler\\_utils.h](#)

## 2.18 spe\_sig\_notify\_1\_area\_t Struct Reference

```
#include <cbea_map.h>
```

### Data Fields

- unsigned char [reserved\\_0\\_B](#) [12]
- unsigned int [SPU\\_Sig\\_Notify\\_1](#)

### 2.18.1 Detailed Description

Definition at line 69 of file cbea\_map.h.

### 2.18.2 Field Documentation

#### 2.18.2.1 unsigned char reserved\_0\_B[12]

Definition at line 70 of file cbea\_map.h.

#### 2.18.2.2 unsigned int SPU\_Sig\_Notify\_1

Definition at line 71 of file cbea\_map.h.

Referenced by `_base_spe_signal_write()`.

The documentation for this struct was generated from the following file:

- [cbea\\_map.h](#)

## 2.19 spe\_sig\_notify\_2\_area\_t Struct Reference

```
#include <cbea_map.h>
```

### Data Fields

- unsigned char [reserved\\_0\\_B](#) [12]
- unsigned int [SPU\\_Sig\\_Notify\\_2](#)

### 2.19.1 Detailed Description

Definition at line 74 of file cbea\_map.h.

### 2.19.2 Field Documentation

#### 2.19.2.1 unsigned char reserved\_0\_B[12]

Definition at line 75 of file cbea\_map.h.

#### 2.19.2.2 unsigned int SPU\_Sig\_Notify\_2

Definition at line 76 of file cbea\_map.h.

Referenced by `_base_spe_signal_write()`.

The documentation for this struct was generated from the following file:

- [cbea\\_map.h](#)

## 2.20 spe\_spu\_control\_area\_t Struct Reference

```
#include <cbea_map.h>
```

### Data Fields

- 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](#)

### 2.20.1 Detailed Description

Definition at line 54 of file cbea\_map.h.

### 2.20.2 Field Documentation

#### 2.20.2.1 unsigned char reserved\_0\_3[4]

Definition at line 55 of file cbea\_map.h.

#### 2.20.2.2 unsigned int SPU\_Out\_Mbox

Definition at line 56 of file cbea\_map.h.

#### 2.20.2.3 unsigned char reserved\_8\_B[4]

Definition at line 57 of file cbea\_map.h.

#### 2.20.2.4 unsigned int SPU\_In\_Mbox

Definition at line 58 of file cbea\_map.h.

#### 2.20.2.5 unsigned char reserved\_10\_13[4]

Definition at line 59 of file cbea\_map.h.

**2.20.2.6 unsigned int SPU\_Mbox\_Stat**

Definition at line 60 of file cbea\_map.h.

**2.20.2.7 unsigned char reserved\_18\_1B[4]**

Definition at line 61 of file cbea\_map.h.

**2.20.2.8 unsigned int SPU\_RunCntl**

Definition at line 62 of file cbea\_map.h.

**2.20.2.9 unsigned char reserved\_20\_23[4]**

Definition at line 63 of file cbea\_map.h.

**2.20.2.10 unsigned int SPU\_Status**

Definition at line 64 of file cbea\_map.h.

**2.20.2.11 unsigned char reserved\_28\_33[12]**

Definition at line 65 of file cbea\_map.h.

**2.20.2.12 unsigned int SPU\_NPC**

Definition at line 66 of file cbea\_map.h.

The documentation for this struct was generated from the following file:

- [cbea\\_map.h](#)

## 2.21 spe\_stop\_info\_t Struct Reference

```
#include <libspe2-types.h>
```

### 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](#)
- int [spu\\_status](#)

### 2.21.1 Detailed Description

[spe\\_stop\\_info\\_t](#)

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

### 2.21.2 Field Documentation

#### 2.21.2.1 unsigned int stop\_reason

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

Referenced by `_base_spe_context_run()`.

#### 2.21.2.2 int spe\_exit\_code

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

Referenced by `_base_spe_context_run()`.

#### 2.21.2.3 int spe\_signal\_code

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

Referenced by `_base_spe_context_run()`.

#### 2.21.2.4 int spe\_runtime\_error

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

Referenced by `_base_spe_context_run()`.

**2.21.2.5 int spe\_runtime\_exception**

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

Referenced by `_base_spe_context_run()`.

**2.21.2.6 int spe\_runtime\_fatal**

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

Referenced by `_base_spe_context_run()`.

**2.21.2.7 int spe\_callback\_error**

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

Referenced by `_base_spe_context_run()`.

**2.21.2.8 int spe\_isolation\_error**

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

Referenced by `_base_spe_context_run()`.

**2.21.2.9 void\* \_\_reserved\_ptr**

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

**2.21.2.10 unsigned long long \_\_reserved\_u64**

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

**2.21.2.11 union { ... } result**

Referenced by `_base_spe_context_run()`.

**2.21.2.12 int spu\_status**

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

Referenced by `_base_spe_context_run()`.

The documentation for this struct was generated from the following file:

- [libspe2-types.h](#)



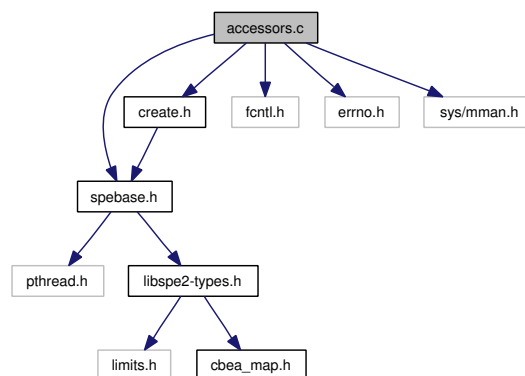
## Chapter 3

# libspe2 File Documentation

### 3.1 accessors.c File Reference

```
#include "spebase.h"
#include "create.h"
#include <fcntl.h>
#include <errno.h>
#include <sys/mman.h>
```

Include dependency graph for accessors.c:



### Functions

- `void * __base_spe_ps_area_get (spe_context_ptr_t spe, enum ps_area area)`
- `void * __base_spe_ls_area_get (spe_context_ptr_t spe)`
- `__attribute__((noinline))`
- `int __base_spe_event_source_acquire (spe_context_ptr_t spe, enum fd_name fdesc)`
- `void __base_spe_event_source_release (struct spe_context *spe, enum fd_name fdesc)`
- `int __base_spe_spe_dir_get (spe_context_ptr_t spe)`
- `int __base_spe_stop_event_source_get (spe_context_ptr_t spe)`
- `int __base_spe_stop_event_target_get (spe_context_ptr_t spe)`

- `int __base_spe_ls_size_get (spe_context_ptr_t spe)`

### 3.1.1 Function Documentation

#### 3.1.1.1 `__attribute__((noinline))`

Definition at line 69 of file accessors.c.

```
70 {
71     return;
72 }
```

#### 3.1.1.2 `int __base_spe_event_source_acquire (spe_context_ptr_t spe, enum fd_name fdesc)`

Definition at line 74 of file accessors.c.

References `_base_spe_open_if_closed()`.

Referenced by `_event_spe_event_handler_deregister()`, and `_event_spe_event_handler_register()`.

```
75 {
76     return _base_spe_open_if_closed(spe, fdesc, 0);
77 }
```

Here is the call graph for this function:



#### 3.1.1.3 `void __base_spe_event_source_release (struct spe_context * spectx, enum fd_name fdesc)`

`__base_spe_event_source_release` releases the file descriptor to the specified event source

##### Parameters:

*spectx* Specifies the SPE context

*fdesc* Specifies the event source

Definition at line 79 of file accessors.c.

```
80 {
81     _base_spe_close_if_open(spe, fdesc);
82 }
```

#### 3.1.1.4 `int __base_spe_spe_dir_get (spe_context_ptr_t spe)`

Definition at line 84 of file accessors.c.

References `spe_context::base_private`, and `spe_context_base_priv::fd_spe_dir`.

```
85 {
86     return spe->base_private->fd_spe_dir;
87 }
```

**3.1.1.5 int \_\_base\_spe\_stop\_event\_source\_get (spe\_context\_ptr\_t spe)**

speevent users read from this end

Definition at line 92 of file accessors.c.

```

93 {
94     return spe->base_private->ev_pipe[1];
95 }
```

**3.1.1.6 int \_\_base\_spe\_stop\_event\_target\_get (spe\_context\_ptr\_t spe)**

speevent writes to this end

Definition at line 100 of file accessors.c.

```

101 {
102     return spe->base_private->ev_pipe[0];
103 }
```

**3.1.1.7 void\* \_base\_spe\_ls\_area\_get (spe\_context\_ptr\_t spe)**

Definition at line 64 of file accessors.c.

References spe\_context::base\_private, and spe\_context\_base\_priv::mem\_mmap\_base.

```

65 {
66     return spe->base_private->mem_mmap_base;
67 }
```

**3.1.1.8 int \_base\_spe\_ls\_size\_get (spe\_context\_ptr\_t spe)**

\_base\_spe\_ls\_size\_get returns the size of the local store area

**Parameters:**

*spectx* Specifies the SPE context

Definition at line 105 of file accessors.c.

```

106 {
107     return LS_SIZE;
108 }
```

**3.1.1.9 void\* \_base\_spe\_ps\_area\_get (spe\_context\_ptr\_t spe, enum ps\_area area)**

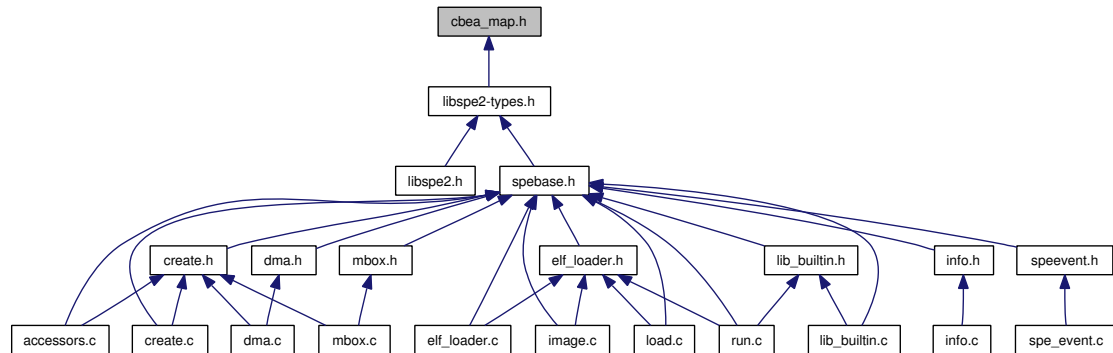
Definition at line 30 of file accessors.c.

References spe\_context::base\_private, spe\_context\_base\_priv::cntl\_mmap\_base, spe\_context\_base\_priv::mfc\_mmap\_base, spe\_context\_base\_priv::mssync\_mmap\_base, spe\_context\_base\_priv::signal1\_mmap\_base, spe\_context\_base\_priv::signal2\_mmap\_base, SPE\_CONTROL\_AREA, SPE\_MFC\_COMMAND\_AREA, SPE\_MSSYNC\_AREA, SPE\_SIG\_NOTIFY\_1\_AREA, and SPE\_SIG\_NOTIFY\_2\_AREA.

```
31 {
32     void *ptr;
33
34     switch (area) {
35         case SPE_MSSYNC_AREA:
36             ptr = spe->base_private->mssync_mmap_base;
37             break;
38         case SPE_MFC_COMMAND_AREA:
39             ptr = spe->base_private->mfc_mmap_base;
40             break;
41         case SPE_CONTROL_AREA:
42             ptr = spe->base_private->cntl_mmap_base;
43             break;
44         case SPE_SIG_NOTIFY_1_AREA:
45             ptr = spe->base_private->signal1_mmap_base;
46             break;
47         case SPE_SIG_NOTIFY_2_AREA:
48             ptr = spe->base_private->signal2_mmap_base;
49             break;
50         default:
51             errno = EINVAL;
52             return NULL;
53             break;
54     }
55
56     if (ptr == MAP_FAILED) {
57         errno = EACCES;
58         return NULL;
59     }
60
61     return ptr;
62 }
```

## 3.2 cbea\_map.h File Reference

This graph shows which files directly or indirectly include this file:



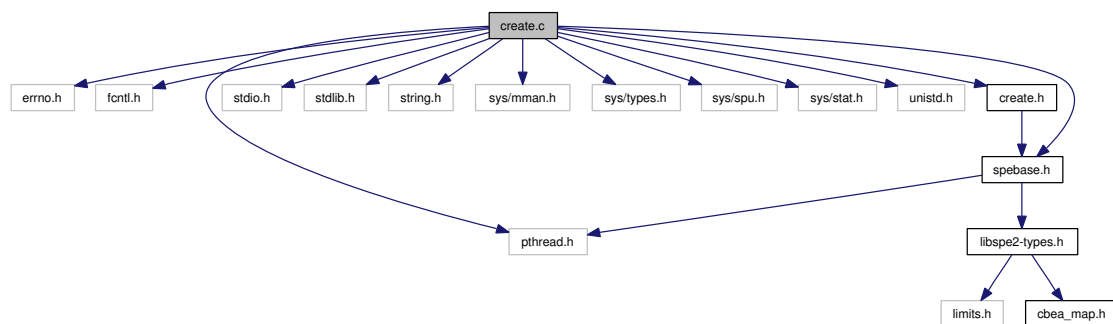
### Data Structures

- struct [spe\\_mssync\\_area\\_t](#)
- struct [spe\\_mfc\\_command\\_area\\_t](#)
- struct [spe\\_spu\\_control\\_area\\_t](#)
- struct [spe\\_sig\\_notify\\_1\\_area\\_t](#)
- struct [spe\\_sig\\_notify\\_2\\_area\\_t](#)

### 3.3 create.c File Reference

```
#include <errno.h>
#include <fcntl.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/spu.h>
#include <sys/stat.h>
#include <unistd.h>
#include "create.h"
#include "spebase.h"
```

Include dependency graph for create.c:



### Data Structures

- struct [fd\\_attr](#)

### Functions

- void [\\_base\\_spe\\_context\\_lock](#) (spe\_context\_ptr\_t spe, enum [fd\\_name](#) fdesc)
- void [\\_base\\_spe\\_context\\_unlock](#) (spe\_context\_ptr\_t spe, enum [fd\\_name](#) fdesc)
- int [\\_base\\_spe\\_open\\_if\\_closed](#) (struct [spe\\_context](#) \*spe, enum [fd\\_name](#) fdesc, int locked)
- void [\\_base\\_spe\\_close\\_if\\_open](#) (struct [spe\\_context](#) \*spe, enum [fd\\_name](#) fdesc)
- [spe\\_context\\_ptr\\_t](#) [\\_base\\_spe\\_context\\_create](#) (unsigned int flags, [spe\\_gang\\_context\\_ptr\\_t](#) gctx, [spe\\_context\\_ptr\\_t](#) aff\_spe)
- [spe\\_gang\\_context\\_ptr\\_t](#) [\\_base\\_spe\\_gang\\_context\\_create](#) (unsigned int flags)
- int [\\_base\\_spe\\_context\\_destroy](#) (spe\_context\_ptr\_t spe)
- int [\\_base\\_spe\\_gang\\_context\\_destroy](#) (spe\_gang\_context\_ptr\_t gctx)

### 3.3.1 Function Documentation

#### 3.3.1.1 void \_base\_spe\_close\_if\_open (struct spe\_context \* *spe*, enum fd\_name *fdesc*)

Definition at line 125 of file create.c.

Referenced by \_\_base\_spe\_event\_source\_release(), and \_base\_spe\_signal\_write().

```

126 {
127     _base_spe_context_lock(spe, fdesc);
128
129     if (spe->base_private->spe_fds_array[(int)fdesc] != -1 &&
130         spe->base_private->spe_fds_refcount[(int)fdesc] == 1) {
131
132         spe->base_private->spe_fds_refcount[(int)fdesc]--;
133         close(spe->base_private->spe_fds_array[(int)fdesc]);
134
135         spe->base_private->spe_fds_array[(int)fdesc] = -1;
136     } else if (spe->base_private->spe_fds_refcount[(int)fdesc] > 0) {
137         spe->base_private->spe_fds_refcount[(int)fdesc]--;
138     }
139
140     _base_spe_context_unlock(spe, fdesc);
141 }
```

#### 3.3.1.2 spe\_context\_ptr\_t \_base\_spe\_context\_create (unsigned int *flags*, spe\_gang\_context\_ptr\_t *gctx*, spe\_context\_ptr\_t *aff\_spe*)

\_base\_spe\_context\_create creates a single SPE context, i.e., the corresponding directory is created in SPUFS either as a subdirectory of a gang or individually (maybe this is best considered a gang of one)

##### Parameters:

*flags*

*gctx* specify NULL if not belonging to a gang

*aff\_spe* specify NULL to skip affinity information

Definition at line 183 of file create.c.

```

185 {
186     char pathname[256];
187     int i, aff_spe_fd = 0;
188     unsigned int spu_createflags = 0;
189     struct spe_context *spe = NULL;
190     struct spe_context_base_priv *priv;
191
192     /* We need a loader present to run in emulated isolated mode */
193     if (flags & SPE_ISOLATE_EMULATE
194         && !_base_spe_emulated_loader_present()) {
195         errno = EINVAL;
196         return NULL;
197     }
198
199     /* Put some sane defaults into the SPE context */
200     spe = malloc(sizeof(*spe));
201     if (!spe) {
202         DEBUG_PRINTF("ERROR: Could not allocate spe context.\n");
203         return NULL;
204     }
205     memset(spe, 0, sizeof(*spe));
```

```

206
207     spe->base_private = malloc(sizeof(*spe->base_private));
208     if (!spe->base_private) {
209         DEBUG_PRINTF("ERROR: Could not allocate "
210                     "spe->base_private context.\n");
211         free(spe);
212         return NULL;
213     }
214
215     /* just a convenience variable */
216     priv = spe->base_private;
217
218     priv->fd_spe_dir = -1;
219     priv->mem_mmap_base = MAP_FAILED;
220     priv->psmap_mmap_base = MAP_FAILED;
221     priv->mssync_mmap_base = MAP_FAILED;
222     priv->mfc_mmap_base = MAP_FAILED;
223     priv->cntl_mmap_base = MAP_FAILED;
224     priv->signall_mmap_base = MAP_FAILED;
225     priv->signal2_mmap_base = MAP_FAILED;
226     priv->loaded_program = NULL;
227
228     for (i = 0; i < NUM_MBOX_FDS; i++) {
229         priv->spe_fds_array[i] = -1;
230         pthread_mutex_init(&priv->fd_lock[i], NULL);
231     }
232
233     /* initialise spu_createflags */
234     if (flags & SPE_ISOLATE) {
235         flags |= SPE_MAP_PS;
236         spu_createflags |= SPU_CREATE_ISOLATE | SPU_CREATE_NOSCHED;
237     }
238
239     if (flags & SPE_EVENTS_ENABLE)
240         spu_createflags |= SPU_CREATE_EVENTS_ENABLED;
241
242     if (aff_spe)
243         spu_createflags |= SPU_CREATE_AFFINITY_SPU;
244
245     if (flags & SPE_AFFINITY_MEMORY)
246         spu_createflags |= SPU_CREATE_AFFINITY_MEM;
247
248     /* Make the SPUDFS directory for the SPE */
249     if (gctx == NULL)
250         sprintf(pathname, "/spu/spethread-%i-%lu",
251                 getpid(), (unsigned long)spe);
252     else
253         sprintf(pathname, "/spu/%s/spethread-%i-%lu",
254                 gctx->base_private->gangname, getpid(),
255                 (unsigned long)spe);
256
257     if (aff_spe)
258         aff_spe_fd = aff_spe->base_private->fd_spe_dir;
259
260     priv->fd_spe_dir = spu_create(pathname, spu_createflags,
261                                 S_IRUSR | S_IWUSR | S_IXUSR, aff_spe_fd);
262
263     if (priv->fd_spe_dir < 0) {
264         DEBUG_PRINTF("ERROR: Could not create SPE %s\n", pathname);
265         perror("spu_create()");
266         free_spe_context(spe);
267         /* we mask most errors, but leave ENODEV */
268         if (errno != ENODEV)
269             errno = EFAULT;
270         return NULL;
271     }
272

```



```

273     priv->flags = flags;
274
275     /* Map the required areas into process memory */
276     priv->mem_mmap_base = mapfileat(priv->fd_spe_dir, "mem", LS_SIZE);
277     if (priv->mem_mmap_base == MAP_FAILED) {
278         DEBUG_PRINTF("ERROR: Could not map SPE memory.\n");
279         free_spe_context(spe);
280         errno = ENOMEM;
281         return NULL;
282     }
283
284     if (flags & SPE_MAP_PS) {
285         /* It's possible to map the entire problem state area with
286          * one mmap - try this first */
287         priv->psmap_mmap_base = mapfileat(priv->fd_spe_dir,
288                                         "psmap", PSMAP_SIZE);
289
290         if (priv->psmap_mmap_base != MAP_FAILED) {
291             priv->mssync_mmap_base =
292                 priv->psmap_mmap_base + MSSYNC_OFFSET;
293             priv->mfc_mmap_base =
294                 priv->psmap_mmap_base + MFC_OFFSET;
295             priv->cntl_mmap_base =
296                 priv->psmap_mmap_base + CNTL_OFFSET;
297             priv->signal1_mmap_base =
298                 priv->psmap_mmap_base + SIGNAL1_OFFSET;
299             priv->signal2_mmap_base =
300                 priv->psmap_mmap_base + SIGNAL2_OFFSET;
301
302         } else {
303             /* map each region separately */
304             priv->mfc_mmap_base =
305                 mapfileat(priv->fd_spe_dir, "mfc", MFC_SIZE);
306             priv->mssync_mmap_base =
307                 mapfileat(priv->fd_spe_dir, "mss", MSS_SIZE);
308             priv->cntl_mmap_base =
309                 mapfileat(priv->fd_spe_dir, "cntl", CNTL_SIZE);
310             priv->signal1_mmap_base =
311                 mapfileat(priv->fd_spe_dir, "signal1",
312                           SIGNAL_SIZE);
313             priv->signal2_mmap_base =
314                 mapfileat(priv->fd_spe_dir, "signal2",
315                           SIGNAL_SIZE);
316
317             if (priv->mfc_mmap_base == MAP_FAILED ||
318                 priv->cntl_mmap_base == MAP_FAILED ||
319                 priv->signal1_mmap_base == MAP_FAILED ||
320                 priv->signal2_mmap_base == MAP_FAILED) {
321                 DEBUG_PRINTF("ERROR: Could not map SPE "
322                             "PS memory.\n");
323                 free_spe_context(spe);
324                 errno = ENOMEM;
325                 return NULL;
326             }
327         }
328     }
329
330     if (flags & SPE_CFG_SIGNOTIFY1_OR) {
331         if (setsignotify(priv->fd_spe_dir, "signal1_type")) {
332             DEBUG_PRINTF("ERROR: Could not open SPE "
333                         "signal1_type file.\n");
334             free_spe_context(spe);
335             errno = EFAULT;
336             return NULL;
337         }
338     }
339

```

```

340         if (flags & SPE_CFG_SIGNOTIFY2_OR) {
341             if (setsignotify(priv->fd_spe_dir, "signal2_type")) {
342                 DEBUG_PRINTF("ERROR: Could not open SPE "
343                             "signal2_type file.\n");
344                 free_spe_context(spe);
345                 errno = EFAULT;
346                 return NULL;
347             }
348         }
349
350     return spe;
351 }

```

### 3.3.1.3 int \_base\_spe\_context\_destroy (spe\_context\_ptr\_t *spectx*)

`_base_spe_context_destroy` cleans up what is left when an SPE executable has exited. Closes open file handles and unmaps memory areas.

#### Parameters:

*spectx* Specifies the SPE context

Definition at line 406 of file create.c.

```

407 {
408     int ret = free_spe_context(spe);
409
410     __spe_context_update_event();
411
412     return ret;
413 }

```

### 3.3.1.4 void \_base\_spe\_context\_lock (spe\_context\_ptr\_t *spe*, enum fd\_name *fd*)

`_base_spe_context_lock` locks members of the SPE context

#### Parameters:

*spectx* Specifies the SPE context

*fd* Specifies the file

Definition at line 91 of file create.c.

Referenced by `_base_spe_close_if_open()`, and `_base_spe_open_if_closed()`.

```

92 {
93     pthread_mutex_lock(&spe->base_private->fd_lock[fdesc]);
94 }

```

### 3.3.1.5 void \_base\_spe\_context\_unlock (spe\_context\_ptr\_t *spe*, enum fd\_name *fd*)

`_base_spe_context_unlock` unlocks members of the SPE context

#### Parameters:

*spectx* Specifies the SPE context

*fd* Specifies the file

Definition at line 96 of file create.c.

Referenced by `_base_spe_close_if_open()`, and `_base_spe_open_if_closed()`.

```

97 {
98     pthread_mutex_unlock(&spe->base_private->fd_lock[fdesc]);
99 }
```

### 3.3.1.6 `spe_gang_context_ptr_t _base_spe_gang_context_create (unsigned int flags)`

creates the directory in SPUFS that will contain all SPEs that are considered a gang Note: I would like to generalize this to a "group" or "set" Additional attributes maintained at the group level should be used to define scheduling constraints such "temporal" (e.g., scheduled all at the same time, i.e., a gang) "topology" (e.g., "closeness" of SPEs for optimal communication)

Definition at line 364 of file create.c.

```

365 {
366     char pathname[256];
367     struct spe_gang_context_base_priv *pgctx = NULL;
368     struct spe_gang_context *gctx = NULL;
369
370     gctx = malloc(sizeof(*gctx));
371     if (!gctx) {
372         DEBUG_PRINTF("ERROR: Could not allocate spe context.\n");
373         return NULL;
374     }
375     memset(gctx, 0, sizeof(*gctx));
376
377     pgctx = malloc(sizeof(*pgctx));
378     if (!pgctx) {
379         DEBUG_PRINTF("ERROR: Could not allocate spe context.\n");
380         free(gctx);
381         return NULL;
382     }
383     memset(pgctx, 0, sizeof(*pgctx));
384
385     gctx->base_private = pgctx;
386
387     sprintf(gctx->base_private->gangname, "gang-%i-%lu", getpid(),
388             (unsigned long)gctx);
389     sprintf(pathname, "/spu/%s", gctx->base_private->gangname);
390
391     gctx->base_private->fd_gang_dir = spu_create(pathname, SPU_CREATE_GANG,
392         S_IRUSR | S_IWUSR | S_IXUSR);
393
394     if (gctx->base_private->fd_gang_dir < 0) {
395         DEBUG_PRINTF("ERROR: Could not create Gang %s\n", pathname);
396         free_spe_gang_context(gctx);
397         errno = EFAULT;
398         return NULL;
399     }
400
401     gctx->base_private->flags = flags;
402
403     return gctx;
404 }
```

### 3.3.1.7 int \_base\_spe\_gang\_context\_destroy (spe\_gang\_context\_ptr\_t gctx)

\_base\_spe\_gang\_context\_destroy destroys a gang context and frees associated resources

#### Parameters:

*gctx* Specifies the SPE gang context

Definition at line 415 of file create.c.

```
416 {
417     return free_spe_gang_context(gctx);
418 }
```

### 3.3.1.8 int \_base\_spe\_open\_if\_closed (struct spe\_context \*spe, enum fd\_name fdesc, int locked)

Definition at line 101 of file create.c.

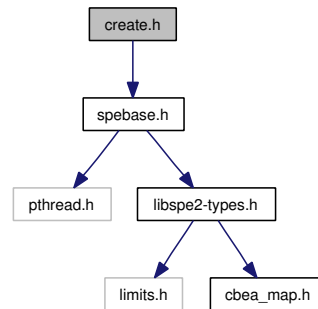
Referenced by \_\_base\_spe\_event\_source\_acquire(), \_base\_spe\_in\_mbox\_status(), \_base\_spe\_in\_mbox\_write(), \_base\_spe\_mssync\_start(), \_base\_spe\_mssync\_status(), \_base\_spe\_out\_intr\_mbox\_read(), \_base\_spe\_out\_intr\_mbox\_status(), \_base\_spe\_out\_mbox\_read(), \_base\_spe\_out\_mbox\_status(), and \_base\_spe\_signal\_write().

```
102 {
103     if (!locked)
104         _base_spe_context_lock(spe, fdesc);
105
106     /* already open? */
107     if (spe->base_private->spe_fds_array[fdesc] != -1) {
108         spe->base_private->spe_fds_refcount[fdesc]++;
109     } else {
110         spe->base_private->spe_fds_array[fdesc] =
111             openat(spe->base_private->fd_spe_dir,
112                  spe_fd_attr[fdesc].name,
113                  spe_fd_attr[fdesc].mode);
114
115         if (spe->base_private->spe_fds_array[(int)fdesc] > 0)
116             spe->base_private->spe_fds_refcount[(int)fdesc]++;
117     }
118
119     if (!locked)
120         _base_spe_context_unlock(spe, fdesc);
121
122     return spe->base_private->spe_fds_array[(int)fdesc];
123 }
```

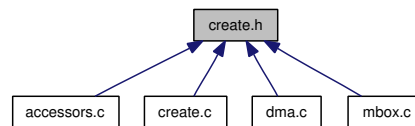
## 3.4 create.h File Reference

```
#include "spebase.h"
```

Include dependency graph for create.h:



This graph shows which files directly or indirectly include this file:



## Functions

- [int \\_base\\_spe\\_open\\_if\\_closed](#) (struct [spe\\_context](#) \*spe, enum [fd\\_name](#) fdesc, int locked)
- [void \\_base\\_spe\\_close\\_if\\_open](#) (struct [spe\\_context](#) \*spe, enum [fd\\_name](#) fdesc)

### 3.4.1 Function Documentation

#### 3.4.1.1 void \_base\_spe\_close\_if\_open (struct [spe\\_context](#) \*spe, enum [fd\\_name](#) fdesc)

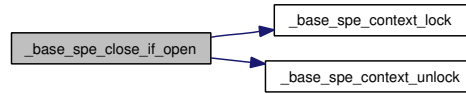
Definition at line 125 of file [create.c](#).

References [\\_base\\_spe\\_context\\_lock\(\)](#), [\\_base\\_spe\\_context\\_unlock\(\)](#), [spe\\_context::base\\_private](#), [spe\\_context\\_base\\_priv::spe\\_fds\\_array](#), and [spe\\_context\\_base\\_priv::spe\\_fds\\_refcount](#).

```

126 {
127     _base_spe_context_lock(spe, fdesc);
128
129     if (spe->base_private->spe_fds_array[(int)fdesc] != -1 &&
130         spe->base_private->spe_fds_refcount[(int)fdesc] == 1) {
131
132         spe->base_private->spe_fds_refcount[(int)fdesc]--;
133         close(spe->base_private->spe_fds_array[(int)fdesc]);
134
135         spe->base_private->spe_fds_array[(int)fdesc] = -1;
136     } else if (spe->base_private->spe_fds_refcount[(int)fdesc] > 0) {
137         spe->base_private->spe_fds_refcount[(int)fdesc]--;
138     }
139
140     _base_spe_context_unlock(spe, fdesc);
141 }
  
```

Here is the call graph for this function:



### 3.4.1.2 `int _base_spe_open_if_closed (struct spe_context *spe, enum fd_name fdesc, int locked)`

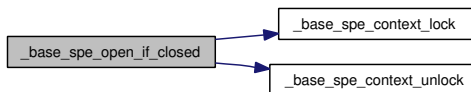
Definition at line 101 of file create.c.

References `_base_spe_context_lock()`, `_base_spe_context_unlock()`, `spe_context::base_private`, `spe_context_base_priv::fd_spe_dir`, `fd_attr::mode`, `fd_attr::name`, `spe_context_base_priv::spe_fds_array`, and `spe_context_base_priv::spe_fds_refcount`.

```

102 {
103     if (!locked)
104         _base_spe_context_lock(spe, fdesc);
105
106     /* already open? */
107     if (spe->base_private->spe_fds_array[fdesc] != -1) {
108         spe->base_private->spe_fds_refcount[fdesc]++;
109     } else {
110         spe->base_private->spe_fds_array[fdesc] =
111             openat(spe->base_private->fd_spe_dir,
112                  spe_fd_attr[fdesc].name,
113                  spe_fd_attr[fdesc].mode);
114
115         if (spe->base_private->spe_fds_array[(int)fdesc] > 0)
116             spe->base_private->spe_fds_refcount[(int)fdesc]++;
117     }
118
119     if (!locked)
120         _base_spe_context_unlock(spe, fdesc);
121
122     return spe->base_private->spe_fds_array[(int)fdesc];
123 }
  
```

Here is the call graph for this function:

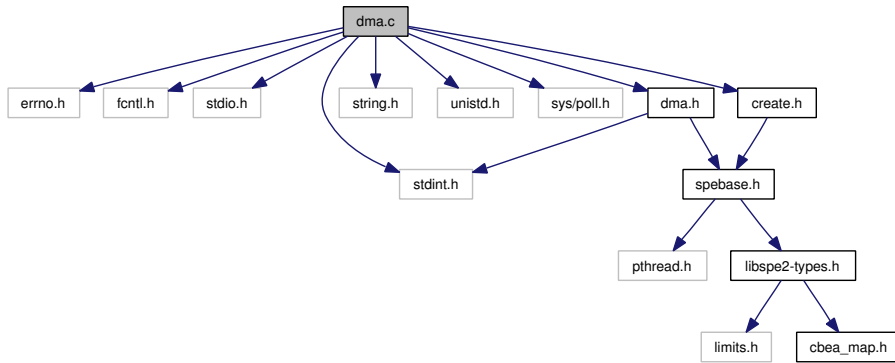


## 3.5 design.txt File Reference

### 3.6 dma.c File Reference

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <sys/poll.h>
#include "create.h"
#include "dma.h"
```

Include dependency graph for dma.c:



### Functions

- `int _base_spe_mfcio_put (spe_context_ptr_t spectx, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)`
- `int _base_spe_mfcio_putb (spe_context_ptr_t spectx, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)`
- `int _base_spe_mfcio_putf (spe_context_ptr_t spectx, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)`
- `int _base_spe_mfcio_get (spe_context_ptr_t spectx, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)`
- `int _base_spe_mfcio_getb (spe_context_ptr_t spectx, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)`
- `int _base_spe_mfcio_getf (spe_context_ptr_t spectx, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)`
- `int _base_spe_mfcio_tag_status_read (spe_context_ptr_t spectx, unsigned int mask, unsigned int behavior, unsigned int *tag_status)`
- `int _base_spe_mssync_start (spe_context_ptr_t spectx)`
- `int _base_spe_mssync_status (spe_context_ptr_t spectx)`



### 3.6.1 Function Documentation

#### 3.6.1.1 `int _base_spe_mfcio_get (spe_context_ptr_t spectx, unsigned int ls, void * ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)`

The `_base_spe_mfcio_get` function places a get DMA command on the proxy command queue of the SPE thread specified by `speid`. The get command transfers `size` bytes of data starting at the effective address specified by `ea` to the local store address specified by `ls`. The DMA is identified by the tag id specified by `tag` and performed according to the transfer class and replacement class specified by `tid` and `rid` respectively.

##### Parameters:

- spectx* Specifies the SPE context
- ls* 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.
- tid* Specifies the transfer class identifier of the DMA command.
- rid* Specifies the replacement class identifier of the DMA command.

##### Returns:

On success, return 0. On failure, -1 is returned.

Definition at line 160 of file `dma.c`.

```
167 {
168     return spe_do_mfc_get(spectx, ls, ea, size, tag, tid, rid, MFC_CMD_GET);
169 }
```

#### 3.6.1.2 `int _base_spe_mfcio_getb (spe_context_ptr_t spectx, unsigned int ls, void * ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)`

The `_base_spe_mfcio_getb` function is identical to `_base_spe_mfcio_get` except that it places a `getb` (get with barrier) DMA command on the proxy command queue. The barrier form ensures that this command and all 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.

##### Parameters:

- spectx* Specifies the SPE context
- ls* 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.
- tid* Specifies the transfer class identifier of the DMA command.
- rid* Specifies the replacement class identifier of the DMA command.

##### Returns:

On success, return 0. On failure, -1 is returned.

Definition at line 171 of file `dma.c`.

```
178 {
179     return spe_do_mfc_get(spectx, ls, ea, size, tag, rid, rid, MFC_CMD_GETB);
180 }
```

### 3.6.1.3 `int _base_spe_mfcio_getf(spe_context_ptr_t spectx, unsigned int ls, void * ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)`

The `_base_spe_mfcio_getf` function is identical to `_base_spe_mfcio_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.

#### Parameters:

- spectx* Specifies the SPE context
- ls* 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.
- tid* Specifies the transfer class identifier of the DMA command.
- rid* Specifies the replacement class identifier of the DMA command.

#### Returns:

On success, return 0. On failure, -1 is returned.

Definition at line 182 of file `dma.c`.

```
189 {
190     return spe_do_mfc_get(spectx, ls, ea, size, tag, tid, rid, MFC_CMD_GETF);
191 }
```

### 3.6.1.4 `int _base_spe_mfcio_put(spe_context_ptr_t spectx, unsigned int ls, void * ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)`

The `_base_spe_mfcio_put` function places a `put` DMA command on the proxy command queue of the SPE thread specified by `speid`. The `put` command transfers `size` bytes of data starting at the local store address specified by `ls` 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.

#### Parameters:

- spectx* Specifies the SPE context
- ls* 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.
- tid* Specifies the transfer class identifier of the DMA command.
- rid* Specifies the replacement class identifier of the DMA command.

#### Returns:

On success, return 0. On failure, -1 is returned.

Definition at line 126 of file `dma.c`.

```
133 {
134     return spe_do_mfc_put(spectx, ls, ea, size, tag, tid, rid, MFC_CMD_PUT);
135 }
```

### 3.6.1.5 `int _base_spe_mfcio_putb (spe_context_ptr_t spectx, unsigned int ls, void * ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)`

The `_base_spe_mfcio_putb` function is identical to `_base_spe_mfcio_put` except that it places a putb (put with barrier) DMA command on the proxy command queue. The barrier form ensures that this command and all 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.

#### Parameters:

- spectx* Specifies the SPE context
- ls* 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.
- tid* Specifies the transfer class identifier of the DMA command.
- rid* Specifies the replacement class identifier of the DMA command.

#### Returns:

On success, return 0. On failure, -1 is returned.

Definition at line 137 of file dma.c.

```
144 {
145     return spe_do_mfc_put(spectx, ls, ea, size, tag, tid, rid, MFC_CMD_PUTB);
146 }
```

### 3.6.1.6 `int _base_spe_mfcio_putf (spe_context_ptr_t spectx, unsigned int ls, void * ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)`

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

#### Parameters:

- spectx* Specifies the SPE context
- ls* 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.
- tid* Specifies the transfer class identifier of the DMA command.
- rid* Specifies the replacement class identifier of the DMA command.

#### Returns:

On success, return 0. On failure, -1 is returned.

Definition at line 148 of file dma.c.

```
155 {
156     return spe_do_mfc_put(spectx, ls, ea, size, tag, tid, rid, MFC_CMD_PUTF);
157 }
```

### 3.6.1.7 `int _base_spe_mfcio_tag_status_read (spe_context_ptr_t spectx, unsigned int mask, unsigned int behavior, unsigned int * tag_status)`

`_base_spe_mfcio_tag_status_read`

No Idea

Definition at line 307 of file dma.c.

```

308 {
309     if ( mask != 0 ) {
310         if (!(spectx->base_private->flags & SPE_MAP_PS))
311             mask = 0;
312     } else {
313         if ((spectx->base_private->flags & SPE_MAP_PS))
314             mask = spectx->base_private->active_tagmask;
315     }
316
317     if (!tag_status) {
318         errno = EINVAL;
319         return -1;
320     }
321
322     switch (behavior) {
323     case SPE_TAG_ALL:
324         return spe_mfcio_tag_status_read_all(spectx, mask, tag_status);
325     case SPE_TAG_ANY:
326         return spe_mfcio_tag_status_read_any(spectx, mask, tag_status);
327     case SPE_TAG_IMMEDIATE:
328         return spe_mfcio_tag_status_read_immediate(spectx, mask, tag_status);
329     default:
330         errno = EINVAL;
331         return -1;
332     }
333 }
```

### 3.6.1.8 `int _base_spe_mssync_start (spe_context_ptr_t spectx)`

`_base_spe_mssync_start` starts Multisource Synchronisation

#### Parameters:

*spectx* Specifies the SPE context

Definition at line 335 of file dma.c.

```

336 {
337     int ret, fd;
338     unsigned int data = 1; /* Any value can be written here */
339
340     volatile struct spe_mssync_area *mss_area =
341         spectx->base_private->mssync_mmap_base;
342
343     if (spectx->base_private->flags & SPE_MAP_PS) {
344         mss_area->MFC_MSSync = data;
345         return 0;
346     } else {
347         fd = _base_spe_open_if_closed(spectx, FD_MSS, 0);
348         if (fd != -1) {
349             ret = write(fd, &data, sizeof (data));
350             if ((ret < 0) && (errno != EIO)) {
351                 perror("spe_mssync_start: internal error");

```

```
352             }
353             return ret < 0 ? -1 : 0;
354         } else
355             return -1;
356     }
357 }
```

### 3.6.1.9 int \_base\_spe\_mssync\_status (spe\_context\_ptr\_t *spectx*)

\_base\_spe\_mssync\_status retrieves status of Multisource Synchronisation

#### Parameters:

*spectx* Specifies the SPE context

Definition at line 359 of file dma.c.

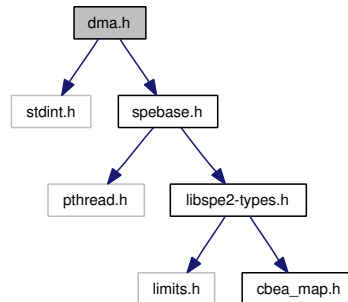
```
360 {
361     int ret, fd;
362     unsigned int data;
363
364     volatile struct spe_mssync_area *mss_area =
365         spectx->base_private->mssync_mmap_base;
366
367     if (spectx->base_private->flags & SPE_MAP_PS) {
368         return mss_area->MFC_MSSync;
369     } else {
370         fd = _base_spe_open_if_closed(spectx, FD_MSS, 0);
371         if (fd != -1) {
372             ret = read(fd, &data, sizeof (data));
373             if ((ret < 0) && (errno != EIO)) {
374                 perror("spe_mssync_start: internal error");
375             }
376             return ret < 0 ? -1 : data;
377         } else
378             return -1;
379     }
380 }
```

## 3.7 dma.h File Reference

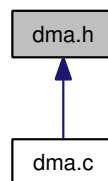
```
#include <stdint.h>
```

```
#include "spebase.h"
```

Include dependency graph for dma.h:



This graph shows which files directly or indirectly include this file:



## Data Structures

- struct [mfc\\_command\\_parameter\\_area](#)

## Enumerations

- enum [mfc\\_cmd](#) {  
[MFC\\_CMD\\_PUT](#) = 0x20, [MFC\\_CMD\\_PUTB](#) = 0x21, [MFC\\_CMD\\_PUTF](#) = 0x22, [MFC\\_CMD\\_GET](#) = 0x40,  
[MFC\\_CMD\\_GETB](#) = 0x41, [MFC\\_CMD\\_GETF](#) = 0x42 }

### 3.7.1 Enumeration Type Documentation

#### 3.7.1.1 enum mfc\_cmd

Enumerator:

*MFC\_CMD\_PUT*  
*MFC\_CMD\_PUTB*  
*MFC\_CMD\_PUTF*  
*MFC\_CMD\_GET*

***MFC\_CMD\_GETB******MFC\_CMD\_GETF***

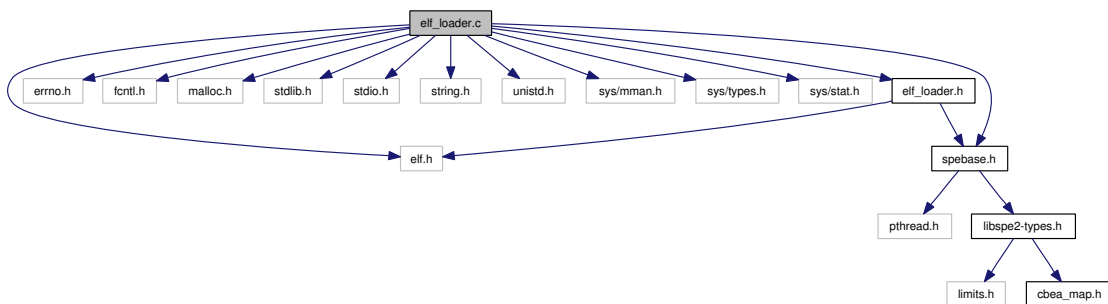
Definition at line 37 of file dma.h.

```
37         {
38             MFC_CMD_PUT   = 0x20,
39             MFC_CMD_PUTB  = 0x21,
40             MFC_CMD_PUTF  = 0x22,
41             MFC_CMD_GET   = 0x40,
42             MFC_CMD_GETB  = 0x41,
43             MFC_CMD_GETF  = 0x42,
44     };
```

### 3.8 elf\_loader.c File Reference

```
#include <elf.h>
#include <errno.h>
#include <fcntl.h>
#include <malloc.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "elf_loader.h"
#include "spebase.h"
```

Include dependency graph for elf\_loader.c:



#### Defines

- `#define __PRINTF(fmt, args...) { fprintf(stderr,fmt , ## args); }`
- `#define DEBUG_PRINTF(fmt, args...)`
- `#define TAG`

#### Functions

- `int _base_spe_verify_spe_elf_image (spe_program_handle_t *handle)`
- `int _base_spe_parse_isolated_elf (spe_program_handle_t *handle, uint64_t *addr, uint32_t *size)`
- `int _base_spe_load_spe_elf (spe_program_handle_t *handle, void *ld_buffer, struct spe_ld_info *ld_info)`
- `int _base_spe_toe_ear (spe_program_handle_t *speh)`



## 3.8.1 Define Documentation

### 3.8.1.1 #define \_\_PRINTF(fmt, args...) { fprintf(stderr,fmt , ## args); }

Definition at line 40 of file elf\_loader.c.

### 3.8.1.2 #define DEBUG\_PRINTF(fmt, args...)

Definition at line 45 of file elf\_loader.c.

Referenced by `_base_spe_context_create()`, `_base_spe_context_run()`, `_base_spe_count_physical_cpus()`, `_base_spe_count_physical_spes()`, `_base_spe_gang_context_create()`, `_base_spe_handle_library_callback()`, `_base_spe_load_spe_elf()`, `_base_spe_out_mbox_read()`, `_base_spe_parse_isolated_elf()`, `_base_spe_program_load()`, and `_base_spe_program_load_complete()`.

### 3.8.1.3 #define TAG

Definition at line 46 of file elf\_loader.c.

## 3.8.2 Function Documentation

### 3.8.2.1 int \_base\_spe\_load\_spe\_elf (spe\_program\_handle\_t \* handle, void \* ld\_buffer, struct spe\_ld\_info \* ld\_info)

Definition at line 201 of file elf\_loader.c.

Referenced by `_base_spe_program_load()`.

```

202 {
203     Elf32_Ehdr *ehdr;
204     Elf32_Phdr *phdr;
205     Elf32_Phdr *ph, *prev_ph;
206
207     Elf32_Shdr *shdr;
208     Elf32_Shdr *sh;
209
210     Elf32_Off toe_addr = 0;
211     long toe_size = 0;
212
213     char* str_table = 0;
214
215     int num_load_seg = 0;
216     void *elf_start;
217     int ret;
218
219     DEBUG_PRINTF ("load_spe_elf(%p, %p)\n", handle, ld_buffer);
220
221     elf_start = handle->elf_image;
222
223     DEBUG_PRINTF ("load_spe_elf(%p, %p)\n", handle->elf_image, ld_buffer);
224     ehdr = (Elf32_Ehdr *) (handle->elf_image);
225
226     /* Check for a Valid SPE ELF Image (again) */
227     if ((ret=check_spe_elf(ehdr)))
228         return ret;
229
230     /* Start processing headers */
231     phdr = (Elf32_Phdr *) ((char *) ehdr + ehdr->e_phoff);

```

```

232     shdr = (Elf32_Shdr *) ((char *) ehdr + ehdr->e_shoff);
233     str_table = (char*)ehdr + shdr[ehdr->e_shstrndx].sh_offset;
234
235     /* traverse the sections to locate the toe segment */
236     /* by specification, the toe sections are grouped together in a segment */
237     for (sh = shdr; sh < &shdr[ehdr->e_shnum]; ++sh)
238     {
239         DEBUG_PRINTF("section name: %s ( start: 0x%04x, size: 0x%04x)\n", str_table+sh->sh_name,
240             if (strcmp(".toe", str_table+sh->sh_name) == 0) {
241                 DEBUG_PRINTF("section offset: %d\n", sh->sh_offset);
242                 toe_size += sh->sh_size;
243                 if ((toe_addr == 0) || (toe_addr > sh->sh_addr))
244                     toe_addr = sh->sh_addr;
245             }
246             /* Disabled : Actually not needed, only good for testing
247             if (strcmp(".bss", str_table+sh->sh_name) == 0) {
248                 DEBUG_PRINTF("zeroing .bss section:\n");
249                 DEBUG_PRINTF("section offset: 0x%04x\n", sh->sh_offset);
250                 DEBUG_PRINTF("section size: 0x%04x\n", sh->sh_size);
251                 memset(ld_buffer + sh->sh_offset, 0, sh->sh_size);
252             } */
253
254 #ifdef DEBUG
255             if (strcmp(".note.spu_name", str_table+sh->sh_name) == 0)
256                 display_debug_output(elf_start, sh);
257 #endif /*DEBUG*/
258     }
259
260     /*
261     * Load all PT_LOAD segments onto the SPE local store buffer.
262     */
263     DEBUG_PRINTF("Segments: 0x%x\n", ehdr->e_phnum);
264     for (ph = phdr, prev_ph = NULL; ph < &phdr[ehdr->e_phnum]; ++ph) {
265         switch (ph->p_type) {
266             case PT_LOAD:
267                 if (!overlay(ph, prev_ph)) {
268                     if (ph->p_filesz < ph->p_memsz) {
269                         DEBUG_PRINTF("padding loaded image with zeros:\n");
270                         DEBUG_PRINTF("start: 0x%04x\n", ph->p_vaddr + ph->p_filesz);
271                         DEBUG_PRINTF("length: 0x%04x\n", ph->p_memsz - ph->p_filesz);
272                         memset(ld_buffer + ph->p_vaddr + ph->p_filesz, 0, ph->p_memsz);
273                     }
274                     copy_to_ld_buffer(handle, ld_buffer, ph,
275                         toe_addr, toe_size);
276                     num_load_seg++;
277                 }
278                 break;
279             case PT_NOTE:
280                 DEBUG_PRINTF("SPE_LOAD found PT_NOTE\n");
281                 break;
282         }
283     }
284     if (num_load_seg == 0)
285     {
286         DEBUG_PRINTF ("no segments to load");
287         errno = EINVAL;
288         return -errno;
289     }
290
291     /* Remember where the code wants to be started */
292     ld_info->entry = ehdr->e_entry;
293     DEBUG_PRINTF ("entry = 0x%x\n", ehdr->e_entry);
294
295     return 0;
296
297 }

```

### 3.8.2.2 int \_base\_spe\_parse\_isolated\_elf (spe\_program\_handle\_t \* *handle*, uint64\_t \* *addr*, uint32\_t \* *size*)

Definition at line 111 of file elf\_loader.c.

```

113 {
114     Elf32_Ehdr *ehdr = (Elf32_Ehdr *)handle->elf_image;
115     Elf32_Phdr *phdr;
116
117     if (!ehdr) {
118         DEBUG_PRINTF("No ELF image has been loaded\n");
119         errno = EINVAL;
120         return -errno;
121     }
122
123     if (ehdr->e_phentsize != sizeof(*phdr)) {
124         DEBUG_PRINTF("Invalid program header format (phdr size=%d)\n",
125                     ehdr->e_phentsize);
126         errno = EINVAL;
127         return -errno;
128     }
129
130     if (ehdr->e_phnum != 1) {
131         DEBUG_PRINTF("Invalid program header count (%d), expected 1\n",
132                     ehdr->e_phnum);
133         errno = EINVAL;
134         return -errno;
135     }
136
137     phdr = (Elf32_Phdr *) (handle->elf_image + ehdr->e_phoff);
138
139     if (phdr->p_type != PT_LOAD || phdr->p_memsz == 0) {
140         DEBUG_PRINTF("SPE program segment is not loadable (type=%x)\n",
141                     phdr->p_type);
142         errno = EINVAL;
143         return -errno;
144     }
145
146     if (addr)
147         *addr = (uint64_t) (unsigned long)
148                 (handle->elf_image + phdr->p_offset);
149
150     if (size)
151         *size = phdr->p_memsz;
152
153     return 0;
154 }
```

### 3.8.2.3 int \_base\_spe\_toe\_ea (spe\_program\_handle\_t \* *speh*)

Definition at line 354 of file elf\_loader.c.

Referenced by \_base\_spe\_image\_open().

```

355 {
356     Elf32_Ehdr *ehdr;
357     Elf32_Shdr *shdr, *sh;
358     char *str_table;
359     char **ch;
360     int ret;
361     long toe_size;
362
363     ehdr = (Elf32_Ehdr*) (speh->elf_image);
```

```

364     shdr = (Elf32_Shdr*) ((char*) ehdr + ehdr->e_shoff);
365     str_table = (char*)ehdr + shdr[ehdr->e_shstrndx].sh_offset;
366
367     toe_size = 0;
368     for (sh = shdr; sh < &shdr[ehdr->e_shnum]; ++sh)
369         if (strcmp(".toe", str_table + sh->sh_name) == 0)
370             toe_size += sh->sh_size;
371
372     ret = 0;
373     if (toe_size > 0) {
374         for (sh = shdr; sh < &shdr[ehdr->e_shnum]; ++sh)
375             if (sh->sh_type == SHT_SYMTAB || sh->sh_type ==
376                 SHT_DYNSYM)
377                 ret = toe_check_syms(ehdr, sh);
378         if (!ret && toe_size != 16) {
379             /* Paranoia */
380             fprintf(stderr, "Unexpected toe size of %ld\n",
381                 toe_size);
382             errno = EINVAL;
383             ret = 1;
384         }
385     }
386     if (!ret && toe_size) {
387         /*
388          * Allocate toe_shadow, and fill it with elf_image.
389          */
390         speh->toe_shadow = malloc(toe_size);
391         if (speh->toe_shadow) {
392             ch = (char**) speh->toe_shadow;
393             if (sizeof(char*) == 8) {
394                 ch[0] = (char*) speh->elf_image;
395                 ch[1] = 0;
396             } else {
397                 ch[0] = 0;
398                 ch[1] = (char*) speh->elf_image;
399                 ch[2] = 0;
400                 ch[3] = 0;
401             }
402         } else {
403             errno = ENOMEM;
404             ret = 1;
405         }
406     }
407     return ret;
408 }

```

### 3.8.2.4 int \_base\_spe\_verify\_spe\_elf\_image (spe\_program\_handle\_t \* *handle*)

verifies integrity of an SPE image

Definition at line 99 of file elf\_loader.c.

Referenced by \_base\_spe\_emulated\_loader\_present(), and \_base\_spe\_image\_open().

```

100 {
101     Elf32_Ehdr *ehdr;
102     void *elf_start;
103
104     elf_start = handle->elf_image;
105     ehdr = (Elf32_Ehdr *) (handle->elf_image);
106
107     return check_spe_elf(ehdr);
108 }

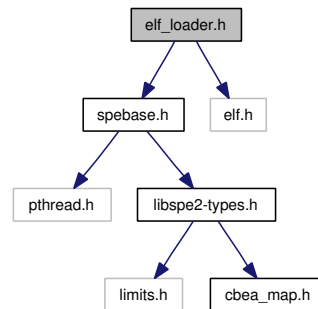
```

## 3.9 elf\_loader.h File Reference

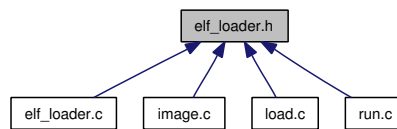
```
#include "spebase.h"
```

```
#include <elf.h>
```

Include dependency graph for elf\_loader.h:



This graph shows which files directly or indirectly include this file:



### Data Structures

- union [addr64](#)
- struct [spe\\_ld\\_info](#)

### Defines

- #define [LS\\_SIZE](#) 0x40000
- #define [SPE\\_LDR\\_PROG\\_start](#) (LS\_SIZE - 512)
- #define [SPE\\_LDR\\_PARAMS\\_start](#) (LS\_SIZE - 128)

### Functions

- int [\\_base\\_spe\\_verify\\_spe\\_elf\\_image](#) ([spe\\_program\\_handle\\_t](#) \*handle)
- int [\\_base\\_spe\\_load\\_spe\\_elf](#) ([spe\\_program\\_handle\\_t](#) \*handle, void \*ld\_buffer, struct [spe\\_ld\\_info](#) \*ld\_info)
- int [\\_base\\_spe\\_parse\\_isolated\\_elf](#) ([spe\\_program\\_handle\\_t](#) \*handle, uint64\_t \*addr, uint32\_t \*size)
- int [\\_base\\_spe\\_toe\\_ear](#) ([spe\\_program\\_handle\\_t](#) \*speh)

### 3.9.1 Define Documentation

#### 3.9.1.1 #define LS\_SIZE 0x40000

Definition at line 23 of file elf\_loader.h.

Referenced by `_base_spe_context_create()`, `_base_spe_context_run()`, and `_base_spe_ls_size_get()`.

#### 3.9.1.2 #define SPE\_LDR\_PARAMS\_start (LS\_SIZE - 128)

Definition at line 26 of file elf\_loader.h.

#### 3.9.1.3 #define SPE\_LDR\_PROG\_start (LS\_SIZE - 512)

Definition at line 25 of file elf\_loader.h.

### 3.9.2 Function Documentation

#### 3.9.2.1 int \_base\_spe\_load\_spe\_elf (spe\_program\_handle\_t \* handle, void \* ld\_buffer, struct spe\_ld\_info \* ld\_info)

Definition at line 201 of file elf\_loader.c.

References `DEBUG_PRINTF`, `spe_program_handle_t::elf_image`, and `spe_ld_info::entry`.

```

202 {
203     Elf32_Ehdr *ehdr;
204     Elf32_Phdr *phdr;
205     Elf32_Phdr *ph, *prev_ph;
206
207     Elf32_Shdr *shdr;
208     Elf32_Shdr *sh;
209
210     Elf32_Off toe_addr = 0;
211     long toe_size = 0;
212
213     char* str_table = 0;
214
215     int num_load_seg = 0;
216     void *elf_start;
217     int ret;
218
219     DEBUG_PRINTF ("load_spe_elf(%p, %p)\n", handle, ld_buffer);
220
221     elf_start = handle->elf_image;
222
223     DEBUG_PRINTF ("load_spe_elf(%p, %p)\n", handle->elf_image, ld_buffer);
224     ehdr = (Elf32_Ehdr *) (handle->elf_image);
225
226     /* Check for a Valid SPE ELF Image (again) */
227     if ((ret=check_spe_elf(ehdr))
228         return ret;
229
230     /* Start processing headers */
231     phdr = (Elf32_Phdr *) ((char *) ehdr + ehdr->e_phoff);
232     shdr = (Elf32_Shdr *) ((char *) ehdr + ehdr->e_shoff);
233     str_table = (char*)ehdr + shdr[ehdr->e_shstrndx].sh_offset;
234
235     /* traverse the sections to locate the toe segment */

```

```

236     /* by specification, the toe sections are grouped together in a segment */
237     for (sh = shdr; sh < &shdr[ehdr->e_shnum]; ++sh)
238     {
239         DEBUG_PRINTF("section name: %s ( start: 0x%04x, size: 0x%04x)\n", str_table+sh->sh_name,
240             if (strcmp(".toe", str_table+sh->sh_name) == 0) {
241                 DEBUG_PRINTF("section offset: %d\n", sh->sh_offset);
242                 toe_size += sh->sh_size;
243                 if ((toe_addr == 0) || (toe_addr > sh->sh_addr))
244                     toe_addr = sh->sh_addr;
245             }
246         /* Disabled : Actually not needed, only good for testing
247         if (strcmp(".bss", str_table+sh->sh_name) == 0) {
248             DEBUG_PRINTF("zeroing .bss section:\n");
249             DEBUG_PRINTF("section offset: 0x%04x\n", sh->sh_offset);
250             DEBUG_PRINTF("section size: 0x%04x\n", sh->sh_size);
251             memset(ld_buffer + sh->sh_offset, 0, sh->sh_size);
252         } */
253
254 #ifdef DEBUG
255         if (strcmp(".note.spu_name", str_table+sh->sh_name) == 0)
256             display_debug_output(elf_start, sh);
257 #endif /*DEBUG*/
258     }
259
260     /*
261     * Load all PT_LOAD segments onto the SPE local store buffer.
262     */
263     DEBUG_PRINTF("Segments: 0x%x\n", ehdr->e_phnum);
264     for (ph = phdr, prev_ph = NULL; ph < &phdr[ehdr->e_phnum]; ++ph) {
265         switch (ph->p_type) {
266             case PT_LOAD:
267                 if (!overlay(ph, prev_ph)) {
268                     if (ph->p_filesz < ph->p_memsz) {
269                         DEBUG_PRINTF("padding loaded image with zeros:\n");
270                         DEBUG_PRINTF("start: 0x%04x\n", ph->p_vaddr + ph->p_filesz);
271                         DEBUG_PRINTF("length: 0x%04x\n", ph->p_memsz - ph->p_filesz);
272                         memset(ld_buffer + ph->p_vaddr + ph->p_filesz, 0, ph->p_memsz - ph->p_filesz);
273                     }
274                     copy_to_ld_buffer(handle, ld_buffer, ph,
275                         toe_addr, toe_size);
276                     num_load_seg++;
277                 }
278                 break;
279             case PT_NOTE:
280                 DEBUG_PRINTF("SPE_LOAD found PT_NOTE\n");
281                 break;
282         }
283     }
284     if (num_load_seg == 0)
285     {
286         DEBUG_PRINTF ("no segments to load");
287         errno = EINVAL;
288         return -errno;
289     }
290
291     /* Remember where the code wants to be started */
292     ld_info->entry = ehdr->e_entry;
293     DEBUG_PRINTF ("entry = 0x%x\n", ehdr->e_entry);
294
295     return 0;
296
297 }

```

### 3.9.2.2 `int _base_spe_parse_isolated_elf (spe_program_handle_t * handle, uint64_t * addr, uint32_t * size)`

Definition at line 111 of file elf\_loader.c.

References `DEBUG_PRINTF`, and `spe_program_handle_t::elf_image`.

```

113 {
114     Elf32_Ehdr *ehdr = (Elf32_Ehdr *)handle->elf_image;
115     Elf32_Phdr *phdr;
116
117     if (!ehdr) {
118         DEBUG_PRINTF("No ELF image has been loaded\n");
119         errno = EINVAL;
120         return -errno;
121     }
122
123     if (ehdr->e_phentsize != sizeof(*phdr)) {
124         DEBUG_PRINTF("Invalid program header format (phdr size=%d)\n",
125                     ehdr->e_phentsize);
126         errno = EINVAL;
127         return -errno;
128     }
129
130     if (ehdr->e_phnum != 1) {
131         DEBUG_PRINTF("Invalid program header count (%d), expected 1\n",
132                     ehdr->e_phnum);
133         errno = EINVAL;
134         return -errno;
135     }
136
137     phdr = (Elf32_Phdr *) (handle->elf_image + ehdr->e_phoff);
138
139     if (phdr->p_type != PT_LOAD || phdr->p_memsz == 0) {
140         DEBUG_PRINTF("SPE program segment is not loadable (type=%x)\n",
141                     phdr->p_type);
142         errno = EINVAL;
143         return -errno;
144     }
145
146     if (addr)
147         *addr = (uint64_t) (unsigned long)
148                 (handle->elf_image + phdr->p_offset);
149
150     if (size)
151         *size = phdr->p_memsz;
152
153     return 0;
154 }
```

### 3.9.2.3 `int _base_spe_toe_eat (spe_program_handle_t * speh)`

Definition at line 354 of file elf\_loader.c.

References `spe_program_handle_t::elf_image`, and `spe_program_handle_t::toe_shadow`.

```

355 {
356     Elf32_Ehdr *ehdr;
357     Elf32_Shdr *shdr, *sh;
358     char *str_table;
359     char **ch;
360     int ret;
361     long toe_size;
```



```

362
363     ehdr = (Elf32_Ehdr*) (speh->elf_image);
364     shdr = (Elf32_Shdr*) ((char*) ehdr + ehdr->e_shoff);
365     str_table = (char*)ehdr + shdr[ehdr->e_shstrndx].sh_offset;
366
367     toe_size = 0;
368     for (sh = shdr; sh < &shdr[ehdr->e_shnum]; ++sh)
369         if (strcmp(".toe", str_table + sh->sh_name) == 0)
370             toe_size += sh->sh_size;
371
372     ret = 0;
373     if (toe_size > 0) {
374         for (sh = shdr; sh < &shdr[ehdr->e_shnum]; ++sh)
375             if (sh->sh_type == SHT_SYMTAB || sh->sh_type ==
376                 SHT_DYNSYM)
377                 ret = toe_check_syms(ehdr, sh);
378         if (!ret && toe_size != 16) {
379             /* Paranoia */
380             fprintf(stderr, "Unexpected toe size of %ld\n",
381                 toe_size);
382             errno = EINVAL;
383             ret = 1;
384         }
385     }
386     if (!ret && toe_size) {
387         /*
388          * Allocate toe_shadow, and fill it with elf_image.
389          */
390         speh->toe_shadow = malloc(toe_size);
391         if (speh->toe_shadow) {
392             ch = (char**) speh->toe_shadow;
393             if (sizeof(char*) == 8) {
394                 ch[0] = (char*) speh->elf_image;
395                 ch[1] = 0;
396             } else {
397                 ch[0] = 0;
398                 ch[1] = (char*) speh->elf_image;
399                 ch[2] = 0;
400                 ch[3] = 0;
401             }
402         } else {
403             errno = ENOMEM;
404             ret = 1;
405         }
406     }
407     return ret;
408 }

```

### 3.9.2.4 int \_base\_spe\_verify\_spe\_elf\_image (spe\_program\_handle\_t \* handle)

verifies integrity of an SPE image

Definition at line 99 of file elf\_loader.c.

References `spe_program_handle_t::elf_image`.

```

100 {
101     Elf32_Ehdr *ehdr;
102     void *elf_start;
103
104     elf_start = handle->elf_image;
105     ehdr = (Elf32_Ehdr *) (handle->elf_image);
106
107     return check_spe_elf(ehdr);
108 }

```

## 3.10 handler\_utils.h File Reference

### Data Structures

- struct [spe\\_reg128](#)

### Defines

- #define [LS\\_SIZE](#) 0x40000
- #define [LS\\_ADDR\\_MASK](#) (LS\_SIZE - 1)
- #define [\\_\\_PRINTF](#)(fmt, args...) { fprintf(stderr,fmt , ## args); }
- #define [DEBUG\\_PRINTF](#)(fmt, args...)
- #define [LS\\_ARG\\_ADDR](#)(\_index) (&((struct [spe\\_reg128](#) \*) ((char \*) ls + ls\_args))[\_index])
- #define [DECL\\_RET](#)() struct [spe\\_reg128](#) \*ret = LS\_ARG\_ADDR(0)
- #define [GET\\_LS\\_PTR](#)(\_off) (void \*) ((char \*) ls + ((\_off) & LS\_ADDR\_MASK))
- #define [GET\\_LS\\_PTR\\_NULL](#)(\_off) ((\_off) ? GET\_LS\_PTR(\_off) : NULL)
- #define [DECL\\_0\\_ARGS](#)() unsigned int ls\_args = (opdata & 0xfffff)
- #define [DECL\\_1\\_ARGS](#)()
- #define [DECL\\_2\\_ARGS](#)()
- #define [DECL\\_3\\_ARGS](#)()
- #define [DECL\\_4\\_ARGS](#)()
- #define [DECL\\_5\\_ARGS](#)()
- #define [DECL\\_6\\_ARGS](#)()
- #define [PUT\\_LS\\_RC](#)(\_a, \_b, \_c, \_d)

#### 3.10.1 Define Documentation

##### 3.10.1.1 #define \_\_PRINTF(fmt, args...) { fprintf(stderr,fmt , ## args); }

Definition at line 32 of file handler\_utils.h.

##### 3.10.1.2 #define DEBUG\_PRINTF(fmt, args...)

Definition at line 36 of file handler\_utils.h.

##### 3.10.1.3 #define DECL\_0\_ARGS() unsigned int ls\_args = (opdata & 0xfffff)

Definition at line 51 of file handler\_utils.h.

##### 3.10.1.4 #define DECL\_1\_ARGS()

**Value:**

```
DECL_0_ARGS(); \
    struct spe_reg128 *arg0 = LS_ARG_ADDR(0)
```

Definition at line 54 of file handler\_utils.h.

### 3.10.1.5 #define DECL\_2\_ARGS()

**Value:**

```
DECL_1_ARGS(); \
    struct spe_reg128 *arg1 = LS_ARG_ADDR(1)
```

Definition at line 58 of file handler\_utils.h.

### 3.10.1.6 #define DECL\_3\_ARGS()

**Value:**

```
DECL_2_ARGS(); \
    struct spe_reg128 *arg2 = LS_ARG_ADDR(2)
```

Definition at line 62 of file handler\_utils.h.

### 3.10.1.7 #define DECL\_4\_ARGS()

**Value:**

```
DECL_3_ARGS(); \
    struct spe_reg128 *arg3 = LS_ARG_ADDR(3)
```

Definition at line 66 of file handler\_utils.h.

### 3.10.1.8 #define DECL\_5\_ARGS()

**Value:**

```
DECL_4_ARGS(); \
    struct spe_reg128 *arg4 = LS_ARG_ADDR(4)
```

Definition at line 70 of file handler\_utils.h.

### 3.10.1.9 #define DECL\_6\_ARGS()

**Value:**

```
DECL_5_ARGS(); \
    struct spe_reg128 *arg5 = LS_ARG_ADDR(5)
```

Definition at line 74 of file handler\_utils.h.

### 3.10.1.10 #define DECL\_RET() struct spe\_reg128 \*ret = LS\_ARG\_ADDR(0)

Definition at line 42 of file handler\_utils.h.

**3.10.1.11 #define GET\_LS\_PTR(\_off) (void \*) ((char \*) ls + (\_off) & LS\_ADDR\_MASK)**

Definition at line 45 of file handler\_utils.h.

**3.10.1.12 #define GET\_LS\_PTR\_NULL(\_off) ((\_off) ? GET\_LS\_PTR(\_off) : NULL)**

Definition at line 48 of file handler\_utils.h.

**3.10.1.13 #define LS\_ADDR\_MASK (LS\_SIZE - 1)**

Definition at line 29 of file handler\_utils.h.

**3.10.1.14 #define LS\_ARG\_ADDR(\_index) (&((struct spe\_reg128 \*) ((char \*) ls + ls\_args))[\_index])**

Definition at line 39 of file handler\_utils.h.

**3.10.1.15 #define LS\_SIZE 0x40000**

Definition at line 28 of file handler\_utils.h.

**3.10.1.16 #define PUT\_LS\_RC(\_a, \_b, \_c, \_d)**

**Value:**

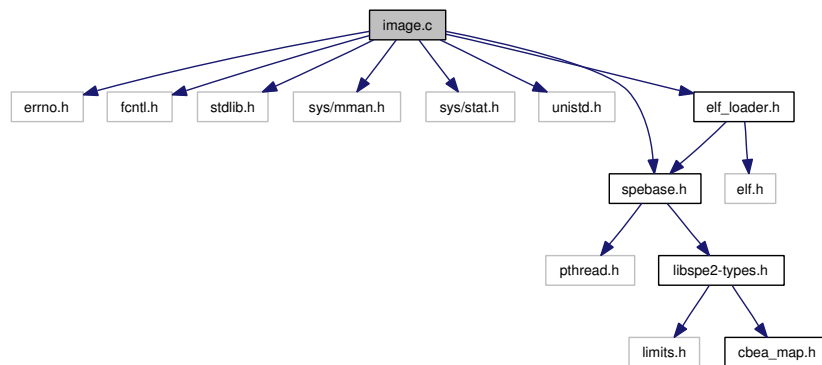
```
ret->slot[0] = (unsigned int) (_a);           \
ret->slot[1] = (unsigned int) (_b);           \
ret->slot[2] = (unsigned int) (_c);           \
ret->slot[3] = (unsigned int) (_d);           \
__asm__ __volatile__ ("sync" : : : "memory")
```

Definition at line 78 of file handler\_utils.h.

## 3.11 image.c File Reference

```
#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#include "elf_loader.h"
#include "spebase.h"
```

Include dependency graph for image.c:



## Data Structures

- struct [image\\_handle](#)

## Functions

- [spe\\_program\\_handle\\_t \\* \\_base\\_spe\\_image\\_open](#) (const char \*filename)
- [int \\_base\\_spe\\_image\\_close](#) (spe\_program\_handle\_t \*handle)

### 3.11.1 Function Documentation

#### 3.11.1.1 [int \\_base\\_spe\\_image\\_close](#) (spe\_program\_handle\_t \* *handle*)

[\\_base\\_spe\\_image\\_close](#) unmaps an SPE ELF object that was previously mapped using [spe\\_open\\_image](#).

#### Parameters:

*handle* handle to open file

#### Return values:

0 On success, [spe\\_close\\_image](#) returns 0.

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

Possible values for errno:

EINVAL From `spe_close_image`, this indicates that the file, specified by `filename`, was not previously mapped by a call to `spe_open_image`.

Definition at line 96 of file `image.c`.

```

97 {
98     int ret = 0;
99     struct image_handle *ih;
100
101     if (!handle) {
102         errno = EINVAL;
103         return -1;
104     }
105
106     ih = (struct image_handle *)handle;
107
108     if (!ih->speh.elf_image || !ih->map_size) {
109         errno = EINVAL;
110         return -1;
111     }
112
113     if (ih->speh.toe_shadow)
114         free(ih->speh.toe_shadow);
115
116     ret = munmap(ih->speh.elf_image, ih->map_size);
117     free(handle);
118
119     return ret;
120 }
```

### 3.11.1.2 `spe_program_handle_t* _base_spe_image_open (const char *filename)`

`_base_spe_image_open` maps an SPE ELF executable indicated by `filename` into system memory and returns the mapped address appropriate for use by the `spe_create_thread` API. It is often more convenient/appropriate to use the loading methodologies where SPE ELF objects are converted to PPE static or shared libraries with symbols which point to the SPE ELF objects after these special libraries are loaded. These libraries are then linked with the associated PPE code to provide a direct symbol reference to the SPE ELF object. The symbols in this scheme are equivalent to the address returned from the `spe_open_image` function. SPE ELF objects loaded using this function are not shared with other processes, but SPE ELF objects loaded using the other scheme, mentioned above, can be shared if so desired.

#### Parameters:

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

#### Returns:

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 appropriately.

Possible values for errno include:

EACCES The calling process does not have permission to access the specified file.

EFAULT The filename parameter points to an address that was not contained in the calling process's address space.

A number of other errno values could be returned by the `open(2)`, `fstat(2)`, `mmap(2)`, `munmap(2)`, or `close(2)` system calls which may be utilized by the `spe_open_image` or `spe_close_image` functions.

**See also:**

spe\_create\_thread

Definition at line 37 of file image.c.

```

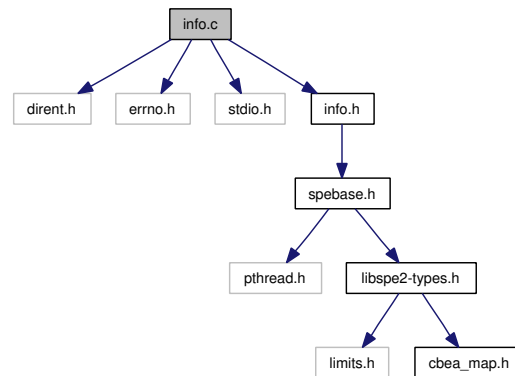
38 {
39     /* allocate an extra integer in the spe handle to keep the mapped size information */
40     struct image_handle *ret;
41     int binfd = -1, f_stat;
42     struct stat statbuf;
43     size_t ps = getpagesize ();
44
45     ret = malloc(sizeof(struct image_handle));
46     if (!ret)
47         return NULL;
48
49     ret->speh.handle_size = sizeof(spe_program_handle_t);
50     ret->speh.toe_shadow = NULL;
51
52     binfd = open(filename, O_RDONLY);
53     if (binfd < 0)
54         goto ret_err;
55
56     f_stat = fstat(binfd, &statbuf);
57     if (f_stat < 0)
58         goto ret_err;
59
60     /* Sanity: is it executable ?
61     */
62     if(!(statbuf.st_mode & (S_IXUSR | S_IXGRP | S_IXOTH))) {
63         errno=EACCES;
64         goto ret_err;
65     }
66
67     /* now store the size at the extra allocated space */
68     ret->map_size = (statbuf.st_size + ps - 1) & ~(ps - 1);
69
70     ret->speh.elf_image = mmap(NULL, ret->map_size,
71                               PROT_WRITE | PROT_READ,
72                               MAP_PRIVATE, binfd, 0);
73     if (ret->speh.elf_image == MAP_FAILED)
74         goto ret_err;
75
76     /*Verify that this is a valid SPE ELF object*/
77     if(!_base_spe_verify_spe_elf_image((spe_program_handle_t *)ret))
78         goto ret_err;
79
80     if (_base_spe_toe_ear(&ret->speh))
81         goto ret_err;
82
83     /* ok */
84     close(binfd);
85     return (spe_program_handle_t *)ret;
86
87     /* err & cleanup */
88 ret_err:
89     if (binfd >= 0)
90         close(binfd);
91
92     free(ret);
93     return NULL;
94 }

```

## 3.12 info.c File Reference

```
#include <dirent.h>
#include <errno.h>
#include <stdio.h>
#include "info.h"
```

Include dependency graph for info.c:



## Functions

- [int \\_base\\_spe\\_count\\_physical\\_cpus](#) (int cpu\_node)
- [int \\_base\\_spe\\_count\\_usable\\_spes](#) (int cpu\_node)
- [int \\_base\\_spe\\_count\\_physical\\_spes](#) (int cpu\_node)
- [int \\_base\\_spe\\_cpu\\_info\\_get](#) (int info\_requested, int cpu\_node)

### 3.12.1 Function Documentation

#### 3.12.1.1 int \_base\_spe\_count\_physical\_cpus (int cpu\_node)

Definition at line 30 of file info.c.

Referenced by [\\_base\\_spe\\_count\\_physical\\_spes\(\)](#), and [\\_base\\_spe\\_cpu\\_info\\_get\(\)](#).

```

31 {
32     const char    *buff = "/sys/devices/system/cpu";
33     DIR          *dirp;
34     int ret = -2;
35     struct dirent *dp;
36
37     DEBUG_PRINTF ("spe_count_physical_cpus() \n");
38
39     // make sure, cpu_node is in the correct range
40     if (cpu_node != -1) {
41         errno = EINVAL;
42         return -1;
43     }
44
45     // Count number of CPUs in /sys/devices/system/cpu
46     if ((dirp=opendir(buff))==NULL) {
```



```

47         fprintf(stderr, "Error opening %s ", buff);
48         perror("dirlist");
49         errno = EINVAL;
50         return -1;
51     }
52     while((dptr=readdir(dirp)) ) {
53         ret++;
54     }
55     closedir(dirp);
56     return ret/THREADS_PER_BE;
57 }

```

### 3.12.1.2 int \_base\_spe\_count\_physical\_spes (int *cpu\_node*)

Definition at line 71 of file info.c.

Referenced by `_base_spe_count_usable_spes()`, and `_base_spe_cpu_info_get()`.

```

72 {
73     const char      *buff = "/sys/devices/system/spu";
74     DIR             *dirp;
75     int ret = -2;
76     struct dirent   *dptr;
77     int no_of_bes;
78
79     DEBUG_PRINTF ("spe_count_physical_spes()\n");
80
81     // make sure, cpu_node is in the correct range
82     no_of_bes = _base_spe_count_physical_cpus(-1);
83     if (cpu_node < -1 || cpu_node >= no_of_bes ) {
84         errno = EINVAL;
85         return -1;
86     }
87
88     // Count number of SPUs in /sys/devices/system/spu
89     if((dirp=opendir(buff))==NULL) {
90         fprintf(stderr, "Error opening %s ", buff);
91         perror("dirlist");
92         errno = EINVAL;
93         return -1;
94     }
95     while((dptr=readdir(dirp)) ) {
96         ret++;
97     }
98     closedir(dirp);
99
100     if(cpu_node != -1) ret /= no_of_bes; // FIXME
101     return ret;
102 }

```

### 3.12.1.3 int \_base\_spe\_count\_usable\_spes (int *cpu\_node*)

Definition at line 62 of file info.c.

Referenced by `_base_spe_cpu_info_get()`.

```

63 {
64     return _base_spe_count_physical_spes(cpu_node); // FIXME
65 }

```

### 3.12.1.4 int \_base\_spe\_cpu\_info\_get (int *info\_requested*, int *cpu\_node*)

\_base\_spe\_info\_get

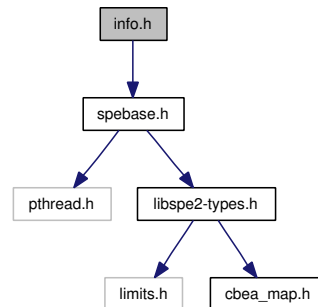
Definition at line 105 of file info.c.

```
105                                     {
106     int ret = 0;
107     errno = 0;
108
109     switch (info_requested) {
110     case SPE_COUNT_PHYSICAL_CPU_NODES:
111         ret = _base_spe_count_physical_cpus (cpu_node);
112         break;
113     case SPE_COUNT_PHYSICAL_SPES:
114         ret = _base_spe_count_physical_spes (cpu_node);
115         break;
116     case SPE_COUNT_USABLE_SPES:
117         ret = _base_spe_count_usable_spes (cpu_node);
118         break;
119     default:
120         errno = EINVAL;
121         ret = -1;
122     }
123     return ret;
124 }
```

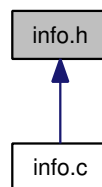
## 3.13 info.h File Reference

```
#include "spebase.h"
```

Include dependency graph for info.h:



This graph shows which files directly or indirectly include this file:



### Defines

- `#define` [THREADS\\_PER\\_BE](#) 2

### Functions

- [int \\_base\\_spe\\_count\\_physical\\_cpus](#) (int *cpu\_node*)
- [int \\_base\\_spe\\_count\\_physical\\_spes](#) (int *cpu\_node*)
- [int \\_base\\_spe\\_count\\_usable\\_spes](#) (int *cpu\_node*)

#### 3.13.1 Define Documentation

##### 3.13.1.1 `#define` [THREADS\\_PER\\_BE](#) 2

Definition at line 25 of file info.h.

Referenced by [\\_base\\_spe\\_count\\_physical\\_cpus](#)().

#### 3.13.2 Function Documentation

##### 3.13.2.1 [int \\_base\\_spe\\_count\\_physical\\_cpus](#) (int *cpu\_node*)

Definition at line 30 of file info.c.

References `DEBUG_PRINTF`, and `THREADS_PER_BE`.

```

31 {
32     const char    *buff = "/sys/devices/system/cpu";
33     DIR           *dirp;
34     int ret = -2;
35     struct dirent *dptr;
36
37     DEBUG_PRINTF ("spe_count_physical_cpus()\n");
38
39     // make sure, cpu_node is in the correct range
40     if (cpu_node != -1) {
41         errno = EINVAL;
42         return -1;
43     }
44
45     // Count number of CPUs in /sys/devices/system/cpu
46     if((dirp=opendir(buff))==NULL) {
47         fprintf(stderr,"Error opening %s ",buff);
48         perror("dirlist");
49         errno = EINVAL;
50         return -1;
51     }
52     while((dptr=readdir(dirp))) {
53         ret++;
54     }
55     closedir(dirp);
56     return ret/THREADS_PER_BE;
57 }

```

### 3.13.2.2 `int _base_spe_count_physical_spes (int cpu_node)`

Definition at line 71 of file `info.c`.

References `_base_spe_count_physical_cpus()`, and `DEBUG_PRINTF`.

```

72 {
73     const char    *buff = "/sys/devices/system/spu";
74     DIR           *dirp;
75     int ret = -2;
76     struct dirent *dptr;
77     int no_of_bes;
78
79     DEBUG_PRINTF ("spe_count_physical_spes()\n");
80
81     // make sure, cpu_node is in the correct range
82     no_of_bes = _base_spe_count_physical_cpus(-1);
83     if (cpu_node < -1 || cpu_node >= no_of_bes ) {
84         errno = EINVAL;
85         return -1;
86     }
87
88     // Count number of SPU's in /sys/devices/system/spu
89     if((dirp=opendir(buff))==NULL) {
90         fprintf(stderr,"Error opening %s ",buff);
91         perror("dirlist");
92         errno = EINVAL;
93         return -1;
94     }
95     while((dptr=readdir(dirp))) {
96         ret++;
97     }
98     closedir(dirp);
99

```

```
100         if(cpu_node != -1) ret /= no_of_bes; // FIXME
101         return ret;
102     }
```

Here is the call graph for this function:



### 3.13.2.3 `int _base_spe_count_usable_spes (int cpu_node)`

Definition at line 62 of file `info.c`.

References `_base_spe_count_physical_spes()`.

```
63 {
64     return _base_spe_count_physical_spes(cpu_node); // FIXME
65 }
```

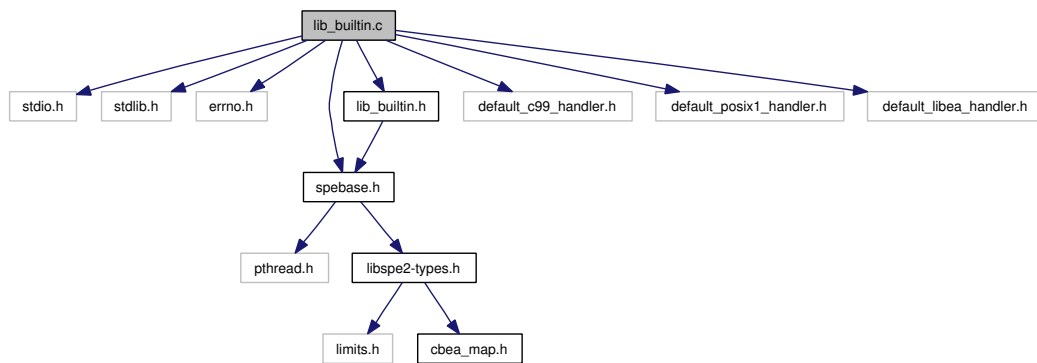
Here is the call graph for this function:



## 3.14 lib\_builtin.c File Reference

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include "spebase.h"
#include "lib_builtin.h"
#include "default_c99_handler.h"
#include "default_posix1_handler.h"
#include "default_libea_handler.h"
```

Include dependency graph for lib\_builtin.c:



### Defines

- #define [HANDLER\\_IDX\(x\)](#) (x & 0xff)

### Functions

- int [\\_base\\_spe\\_callback\\_handler\\_register](#) (void \*handler, unsigned int callnum, unsigned int mode)
- int [\\_base\\_spe\\_callback\\_handler\\_deregister](#) (unsigned int callnum)
- void \* [\\_base\\_spe\\_callback\\_handler\\_query](#) (unsigned int callnum)
- int [\\_base\\_spe\\_handle\\_library\\_callback](#) (struct [spe\\_context](#) \*spe, int callnum, unsigned int npc)

#### 3.14.1 Define Documentation

##### 3.14.1.1 #define HANDLER\_IDX(x) (x & 0xff)

Definition at line 29 of file lib\_builtin.c.

## 3.14.2 Function Documentation

### 3.14.2.1 int \_base\_spe\_callback\_handler\_deregister (unsigned int *callnum*)

unregister a handler function for the specified number NOTE: unregistering a handler from call zero and one is ignored.

Definition at line 78 of file lib\_builtin.c.

```
79 {
80     errno = 0;
81     if (callnum > MAX_CALLNUM) {
82         errno = EINVAL;
83         return -1;
84     }
85     if (callnum < RESERVED) {
86         errno = EACCES;
87         return -1;
88     }
89     if (handlers[callnum] == NULL) {
90         errno = ESRCH;
91         return -1;
92     }
93
94     handlers[callnum] = NULL;
95     return 0;
96 }
```

### 3.14.2.2 void\* \_base\_spe\_callback\_handler\_query (unsigned int *callnum*)

query a handler function for the specified number

Definition at line 98 of file lib\_builtin.c.

```
99 {
100     errno = 0;
101
102     if (callnum > MAX_CALLNUM) {
103         errno = EINVAL;
104         return NULL;
105     }
106     if (handlers[callnum] == NULL) {
107         errno = ESRCH;
108         return NULL;
109     }
110     return handlers[callnum];
111 }
```

### 3.14.2.3 int \_base\_spe\_callback\_handler\_register (void \* *handler*, unsigned int *callnum*, unsigned int *mode*)

register a handler function for the specified number NOTE: registering a handler to call zero and one is ignored.

Definition at line 40 of file lib\_builtin.c.

```
41 {
42     errno = 0;
```

```

43
44     if (callnum > MAX_CALLNUM) {
45         errno = EINVAL;
46         return -1;
47     }
48
49     switch(mode){
50     case SPE_CALLBACK_NEW:
51         if (callnum < RESERVED) {
52             errno = EACCES;
53             return -1;
54         }
55         if (handlers[callnum] != NULL) {
56             errno = EACCES;
57             return -1;
58         }
59         handlers[callnum] = handler;
60         break;
61
62     case SPE_CALLBACK_UPDATE:
63         if (handlers[callnum] == NULL) {
64             errno = ESRCH;
65             return -1;
66         }
67         handlers[callnum] = handler;
68         break;
69     default:
70         errno = EINVAL;
71         return -1;
72         break;
73     }
74     return 0;
75
76 }

```

### 3.14.2.4 `int _base_spe_handle_library_callback (struct spe_context * spe, int callnum, unsigned int npc)`

Definition at line 113 of file `lib_builtin.c`.

Referenced by `_base_spe_context_run()`.

```

115 {
116     int (*handler)(void *, unsigned int);
117     int rc;
118
119     errno = 0;
120     if (!handlers[callnum]) {
121         DEBUG_PRINTF ("No SPE library handler registered for this call.\n");
122         errno=ENOSYS;
123         return -1;
124     }
125
126     handler=handlers[callnum];
127
128     /* For emulated isolation mode, position the
129      * npc so that the buffer for the PPE-assisted
130      * library calls can be accessed. */
131     if (spe->base_private->flags & SPE_ISOLATE_EMULATE)
132         npc = SPE_EMULATE_PARAM_BUFFER;
133
134     rc = handler(spe->base_private->mem_mmap_base, npc);
135     if (rc) {
136         DEBUG_PRINTF ("SPE library call unsupported.\n");

```

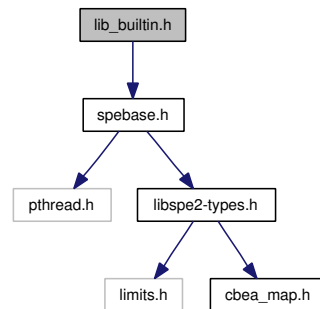


```
137             errno=ENOSYS;
138             return rc;
139         }
140         return 0;
141     }
```

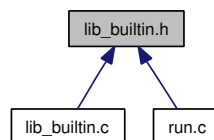
## 3.15 lib\_builtin.h File Reference

```
#include "spebase.h"
```

Include dependency graph for lib\_builtin.h:



This graph shows which files directly or indirectly include this file:



### Defines

- `#define MAX_CALLNUM 255`
- `#define RESERVED 4`

### Functions

- `int _base_spe_handle_library_callback (struct spe_context *spe, int callnum, unsigned int npc)`

#### 3.15.1 Define Documentation

##### 3.15.1.1 #define MAX\_CALLNUM 255

Definition at line 25 of file lib\_builtin.h.

Referenced by `_base_spe_callback_handler_deregister()`, `_base_spe_callback_handler_query()`, and `_base_spe_callback_handler_register()`.

##### 3.15.1.2 #define RESERVED 4

Definition at line 26 of file lib\_builtin.h.

Referenced by `_base_spe_callback_handler_deregister()`, and `_base_spe_callback_handler_register()`.

## 3.15.2 Function Documentation

### 3.15.2.1 int \_base\_spe\_handle\_library\_callback (struct spe\_context \* *spe*, int *callnum*, unsigned int *npc*)

Definition at line 113 of file lib\_builtin.c.

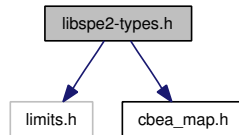
References `spe_context::base_private`, `DEBUG_PRINTF`, `spe_context_base_priv::flags`, `spe_context_base_priv::mem_mmap_base`, `SPE_EMULATE_PARAM_BUFFER`, and `SPE_ISOLATE_EMULATE`.

```
115 {
116     int (*handler)(void *, unsigned int);
117     int rc;
118
119     errno = 0;
120     if (!handlers[callnum]) {
121         DEBUG_PRINTF ("No SPE library handler registered for this call.\n");
122         errno=ENOSYS;
123         return -1;
124     }
125
126     handler=handlers[callnum];
127
128     /* For emulated isolation mode, position the
129      * npc so that the buffer for the PPE-assisted
130      * library calls can be accessed. */
131     if (spe->base_private->flags & SPE_ISOLATE_EMULATE)
132         npc = SPE_EMULATE_PARAM_BUFFER;
133
134     rc = handler(spe->base_private->mem_mmap_base, npc);
135     if (rc) {
136         DEBUG_PRINTF ("SPE library call unsupported.\n");
137         errno=ENOSYS;
138         return rc;
139     }
140     return 0;
141 }
```

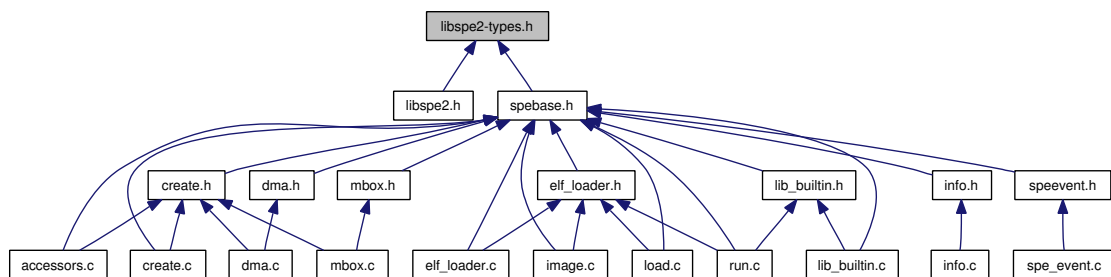
### 3.16 libspe2-types.h File Reference

```
#include <limits.h>
#include "cbea_map.h"
```

Include dependency graph for libspe2-types.h:



This graph shows which files directly or indirectly include this file:



#### 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](#) }

### 3.16.1 Define Documentation

#### 3.16.1.1 **#define SIGSPE SIGURG**

SIGSPE maps to SIGURG

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

#### 3.16.1.2 **#define SPE\_AFFINITY\_MEMORY 0x00002000**

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

Referenced by `_base_spe_context_create()`.

#### 3.16.1.3 **#define SPE\_CALLBACK\_ERROR 6**

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

Referenced by `_base_spe_context_run()`.

#### 3.16.1.4 **#define SPE\_CALLBACK\_NEW 1**

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

Referenced by `_base_spe_callback_handler_register()`.

#### 3.16.1.5 **#define SPE\_CALLBACK\_UPDATE 2**

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

Referenced by `_base_spe_callback_handler_register()`.

#### 3.16.1.6 **#define SPE\_CFG\_SIGNOTIFY1\_OR 0x00000010**

Flags for `spe_context_create`

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

Referenced by `_base_spe_context_create()`.

#### 3.16.1.7 **#define SPE\_CFG\_SIGNOTIFY2\_OR 0x00000020**

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

Referenced by `_base_spe_context_create()`.

#### 3.16.1.8 **#define SPE\_COUNT\_PHYSICAL\_CPU\_NODES 1**

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

Referenced by `_base_spe_cpu_info_get()`.

**3.16.1.9 #define SPE\_COUNT\_PHYSICAL\_SPES 2**

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

Referenced by `_base_spe_cpu_info_get()`.

**3.16.1.10 #define SPE\_COUNT\_USABLE\_SPES 3**

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

Referenced by `_base_spe_cpu_info_get()`.

**3.16.1.11 #define SPE\_DEFAULT\_ENTRY UINT\_MAX**

Flags for `_base_spe_context_run`

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

Referenced by `_base_spe_context_run()`.

**3.16.1.12 #define SPE\_DMA\_ALIGNMENT 0x0008**

Runtime exceptions

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

**3.16.1.13 #define SPE\_DMA\_SEGMENTATION 0x0020**

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

**3.16.1.14 #define SPE\_DMA\_STORAGE 0x0040**

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

**3.16.1.15 #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.

**3.16.1.16 #define SPE\_EVENT\_IN\_MBOX 0x00000002**

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

Referenced by `_event_spe_event_handler_deregister()`, and `_event_spe_event_handler_register()`.

**3.16.1.17 #define SPE\_EVENT\_OUT\_INTR\_MBOX 0x00000001**

Supported SPE events

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

Referenced by `_event_spe_event_handler_deregister()`, and `_event_spe_event_handler_register()`.

**3.16.1.18 #define SPE\_EVENT\_SPE\_STOPPED 0x00000008**

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

Referenced by `_event_spe_event_handler_deregister()`, and `_event_spe_event_handler_register()`.

**3.16.1.19 #define SPE\_EVENT\_TAG\_GROUP 0x00000004**

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

Referenced by `_event_spe_event_handler_deregister()`, and `_event_spe_event_handler_register()`.

**3.16.1.20 #define SPE\_EVENTS\_ENABLE 0x00001000**

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

Referenced by `_base_spe_context_create()`, and `_base_spe_context_run()`.

**3.16.1.21 #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.

Referenced by `_base_spe_context_run()`.

**3.16.1.22 #define SPE\_ISOLATE 0x00000080**

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

Referenced by `_base_spe_context_create()`, `_base_spe_context_run()`, and `_base_spe_program_load()`.

**3.16.1.23 #define SPE\_ISOLATE\_EMULATE 0x00000100**

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

Referenced by `_base_spe_context_create()`, `_base_spe_context_run()`, `_base_spe_handle_library_callback()`, and `_base_spe_program_load()`.

**3.16.1.24 #define SPE\_ISOLATION\_ERROR 7**

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

Referenced by `_base_spe_context_run()`.



**3.16.1.25 #define SPE\_MAP\_PS 0x00000040**

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

Referenced by `_base_spe_context_create()`, `_base_spe_in_mbox_status()`, `_base_spe_in_mbox_write()`, `_base_spe_mfcio_tag_status_read()`, `_base_spe_mssync_start()`, `_base_spe_mssync_status()`, `_base_spe_out_intr_mbox_status()`, `_base_spe_out_mbox_read()`, `_base_spe_out_mbox_status()`, `_base_spe_signal_write()`, and `_event_spe_event_handler_register()`.

**3.16.1.26 #define SPE\_MBOX\_ALL\_BLOCKING 1**

Behavior flags for mailbox read/write functions

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

Referenced by `_base_spe_in_mbox_write()`, and `_base_spe_out_intr_mbox_read()`.

**3.16.1.27 #define SPE\_MBOX\_ANY\_BLOCKING 2**

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

Referenced by `_base_spe_in_mbox_write()`, and `_base_spe_out_intr_mbox_read()`.

**3.16.1.28 #define SPE\_MBOX\_ANY\_NONBLOCKING 3**

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

Referenced by `_base_spe_in_mbox_write()`, and `_base_spe_out_intr_mbox_read()`.

**3.16.1.29 #define SPE\_NO\_CALLBACKS 0x00000002**

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

Referenced by `_base_spe_context_run()`.

**3.16.1.30 #define SPE\_RUN\_USER\_REGS 0x00000001**

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

Referenced by `_base_spe_context_run()`.

**3.16.1.31 #define SPE\_RUNTIME\_ERROR 3**

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

Referenced by `_base_spe_context_run()`.

**3.16.1.32 #define SPE\_RUNTIME\_EXCEPTION 4**

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

Referenced by `_base_spe_context_run()`.

**3.16.1.33 #define SPE\_RUNTIME\_FATAL 5**

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

Referenced by `_base_spe_context_run()`.

**3.16.1.34 #define SPE\_SIG\_NOTIFY\_REG\_1 0x0001**

Signal Targets

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

Referenced by `_base_spe_signal_write()`.

**3.16.1.35 #define SPE\_SIG\_NOTIFY\_REG\_2 0x0002**

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

Referenced by `_base_spe_signal_write()`.

**3.16.1.36 #define SPE\_SPU\_HALT 0x04**

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

Referenced by `_base_spe_context_run()`.

**3.16.1.37 #define SPE\_SPU\_INVALID\_CHANNEL 0x40**

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

Referenced by `_base_spe_context_run()`.

**3.16.1.38 #define SPE\_SPU\_INVALID\_INSTR 0x20**

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

Referenced by `_base_spe_context_run()`.

**3.16.1.39 #define SPE\_SPU\_SINGLE\_STEP 0x10**

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

**3.16.1.40 #define SPE\_SPU\_STOPPED\_BY\_STOP 0x02**

Runtime errors

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

Referenced by `_base_spe_context_run()`.

**3.16.1.41 #define SPE\_SPU\_WAITING\_ON\_CHANNEL 0x08**

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

Referenced by `_base_spe_context_run()`.

**3.16.1.42 #define SPE\_STOP\_AND\_SIGNAL 2**

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

Referenced by `_base_spe_context_run()`.

**3.16.1.43 #define SPE\_TAG\_ALL 1**

Behavior flags tag status functions

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

Referenced by `_base_spe_mfcio_tag_status_read()`.

**3.16.1.44 #define SPE\_TAG\_ANY 2**

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

Referenced by `_base_spe_mfcio_tag_status_read()`.

**3.16.1.45 #define SPE\_TAG\_IMMEDIATE 3**

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

Referenced by `_base_spe_mfcio_tag_status_read()`.

**3.16.2 Typedef Documentation****3.16.2.1 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.

**3.16.2.2 typedef void\* spe\_event\_handler\_ptr\_t**

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

**3.16.2.3 typedef int spe\_event\_handler\_t**

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

#### 3.16.2.4 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.

### 3.16.3 Enumeration Type Documentation

#### 3.16.3.1 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.

```
171 { SPE_MSSYNC_AREA, SPE_MFC_COMMAND_AREA, SPE_CONTROL_AREA, SPE_SIG_NOTIFY_1_AREA, SPE_SIG_NOTIFY_2_AREA
```

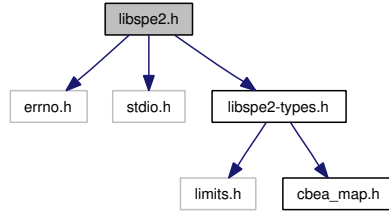
## 3.17 libspe2.h File Reference

```
#include <errno.h>
```

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

```
#include "libspe2-types.h"
```

Include dependency graph for libspe2.h:



## 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)



### 3.17.1 Function Documentation

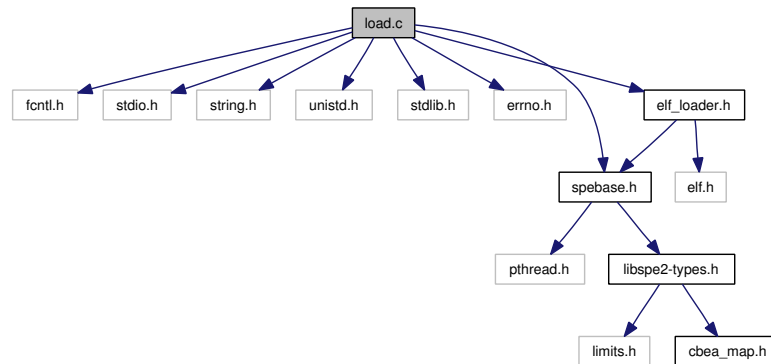
- 3.17.1.1 `int spe_callback_handler_deregister (unsigned int callnum)`
- 3.17.1.2 `void* spe_callback_handler_query (unsigned int callnum)`
- 3.17.1.3 `int spe_callback_handler_register (void * handler, unsigned int callnum, unsigned int mode)`
- 3.17.1.4 `spe_context_ptr_t spe_context_create (unsigned int flags, spe_gang_context_ptr_t gang)`
- 3.17.1.5 `spe_context_ptr_t spe_context_create_affinity (unsigned int flags, spe_context_ptr_t affinity_neighbor, spe_gang_context_ptr_t gang)`
- 3.17.1.6 `int spe_context_destroy (spe_context_ptr_t spe)`
- 3.17.1.7 `int spe_context_run (spe_context_ptr_t spe, unsigned int * entry, unsigned int runflags, void * argp, void * envp, spe_stop_info_t * stopinfo)`
- 3.17.1.8 `int spe_cpu_info_get (int info_requested, int cpu_node)`
- 3.17.1.9 `spe_event_handler_ptr_t spe_event_handler_create (void)`
- 3.17.1.10 `int spe_event_handler_deregister (spe_event_handler_ptr_t evhandler, spe_event_unit_t * event)`
- 3.17.1.11 `int spe_event_handler_destroy (spe_event_handler_ptr_t evhandler)`
- 3.17.1.12 `int spe_event_handler_register (spe_event_handler_ptr_t evhandler, spe_event_unit_t * event)`
- 3.17.1.13 `int spe_event_wait (spe_event_handler_ptr_t evhandler, spe_event_unit_t * events, int max_events, int timeout)`
- 3.17.1.14 `spe_gang_context_ptr_t spe_gang_context_create (unsigned int flags)`
- 3.17.1.15 `int spe_gang_context_destroy (spe_gang_context_ptr_t gang)`
- 3.17.1.16 `int spe_image_close (spe_program_handle_t * program)`
- 3.17.1.17 `spe_program_handle_t* spe_image_open (const char * filename)`
- 3.17.1.18 `int spe_in_mbox_status (spe_context_ptr_t spe)`
- 3.17.1.19 `int spe_in_mbox_write (spe_context_ptr_t spe, unsigned int * mbox_data, int count, unsigned int behavior)`
- 3.17.1.20 `void* spe_ls_area_get (spe_context_ptr_t spe)`
- 3.17.1.21 `int spe_ls_size_get (spe_context_ptr_t spe)`
- 3.17.1.22 `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)`
- 3.17.1.23 `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)`
- 3.17.1.24 `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)`
- 3.17.1.25 `int spe_mfcio_put (spe_context_ptr_t spe, unsigned int ls, void * ea, unsigned int size,`



## 3.18 load.c File Reference

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include "elf_loader.h"
#include "spebase.h"
```

Include dependency graph for load.c:



### Defines

- #define [SPE\\_EMULATED\\_LOADER\\_FILE](#) `"/usr/lib/spe/emulated-loader.bin"`

### Functions

- void [\\_base\\_spe\\_program\\_load\\_complete](#) ([spe\\_context\\_ptr\\_t](#) spectx)
- int [\\_base\\_spe\\_emulated\\_loader\\_present](#) (void)
- int [\\_base\\_spe\\_program\\_load](#) ([spe\\_context\\_ptr\\_t](#) spe, [spe\\_program\\_handle\\_t](#) \*program)

#### 3.18.1 Define Documentation

##### 3.18.1.1 #define SPE\_EMULATED\_LOADER\_FILE `"/usr/lib/spe/emulated-loader.bin"`

Definition at line 30 of file load.c.

#### 3.18.2 Function Documentation

##### 3.18.2.1 int \_base\_spe\_emulated\_loader\_present (void)

Check if the emulated loader is present in the filesystem

**Returns:**

Non-zero if the loader is available, otherwise zero.

Definition at line 145 of file load.c.

Referenced by `_base_spe_context_create()`.

```

146 {
147     spe_program_handle_t *loader = emulated_loader_program();
148
149     if (!loader)
150         return 0;
151
152     return !_base_spe_verify_spe_elf_image(loader);
153 }
```

### 3.18.2.2 `int _base_spe_program_load (spe_context_ptr_t spectx, spe_program_handle_t *program)`

`_base_spe_program_load` loads an ELF image into a context

**Parameters:**

*spectx* Specifies the SPE context

*program* handle to the ELF image

Definition at line 189 of file load.c.

```

190 {
191     int rc = 0;
192     struct spe_ld_info ld_info;
193
194     spe->base_private->loaded_program = program;
195
196     if (spe->base_private->flags & SPE_ISOLATE) {
197         rc = spe_start_isolated_app(spe, program);
198     } else if (spe->base_private->flags & SPE_ISOLATE_EMULATE) {
199         rc = spe_start_emulated_isolated_app(spe, program, &ld_info);
200     } else {
201         rc = _base_spe_load_spe_elf(program,
202                                     spe->base_private->mem_mmap_base, &ld_info);
203         if (!rc)
204             _base_spe_program_load_complete(spe);
205     }
206
207     if (rc != 0) {
208         DEBUG_PRINTF ("Load SPE ELF failed..\n");
209         return -1;
210     }
211
212     spe->base_private->entry = ld_info.entry;
213     spe->base_private->emulated_entry = ld_info.entry;
214
215     return 0;
216 }
217
218 }
```

### 3.18.2.3 void \_base\_spe\_program\_load\_complete (spe\_context\_ptr\_t *spectx*)

Register the SPE program's start address with the oprofile and gdb, by writing to the object-id file.

Definition at line 37 of file load.c.

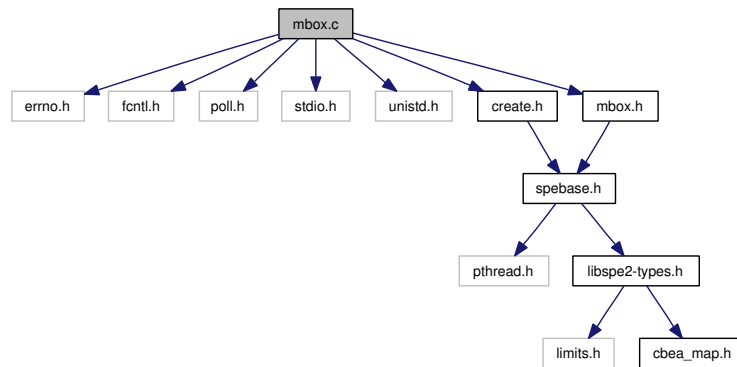
Referenced by `_base_spe_context_run()`, and `_base_spe_program_load()`.

```
38 {
39     int objfd, len;
40     char buf[20];
41     spe_program_handle_t *program;
42
43     program = spectx->base_private->loaded_program;
44
45     if (!program || !program->elf_image) {
46         DEBUG_PRINTF("%s called, but no program loaded\n", __func__);
47         return;
48     }
49
50     objfd = openat(spectx->base_private->fd_spe_dir, "object-id", O_RDWR);
51     if (objfd < 0)
52         return;
53
54     len = sprintf(buf, "%p", program->elf_image);
55     write(objfd, buf, len + 1);
56     close(objfd);
57
58     __spe_context_update_event();
59 }
```

### 3.19 mbox.c File Reference

```
#include <errno.h>
#include <fcntl.h>
#include <poll.h>
#include <stdio.h>
#include <unistd.h>
#include "create.h"
#include "mbox.h"
```

Include dependency graph for mbox.c:



## Functions

- `int _base_spe_out_mbox_read (spe_context_ptr_t spectx, unsigned int mbox_data[ ], int count)`
- `int _base_spe_in_mbox_write (spe_context_ptr_t spectx, unsigned int *mbox_data, int count, int behavior_flag)`
- `int _base_spe_in_mbox_status (spe_context_ptr_t spectx)`
- `int _base_spe_out_mbox_status (spe_context_ptr_t spectx)`
- `int _base_spe_out_intr_mbox_status (spe_context_ptr_t spectx)`
- `int _base_spe_out_intr_mbox_read (spe_context_ptr_t spectx, unsigned int mbox_data[ ], int count, int behavior_flag)`
- `int _base_spe_signal_write (spe_context_ptr_t spectx, unsigned int signal_reg, unsigned int data)`

### 3.19.1 Function Documentation

#### 3.19.1.1 `int _base_spe_in_mbox_status (spe_context_ptr_t spectx)`

The `_base_spe_in_mbox_status` function fetches the status of the SPU inbound mailbox for the SPE thread specified by the `speid` 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:

***spectx*** Specifies the SPE context whose mailbox status is to be read.

**Returns:**

On success, returns the current status of the mailbox, respectively. On failure, -1 is returned.

**See also:**

`spe_read_out_mbox`, `spe_write_in_mbox`, `read` (2)

Definition at line 202 of file `mbox.c`.

```

203 {
204     int rc, ret;
205     volatile struct spe_spu_control_area *cntl_area =
206         spectx->base_private->cntl_mmap_base;
207
208     if (spectx->base_private->flags & SPE_MAP_PS) {
209         ret = (cntl_area->SPU_Mbox_Stat >> 8) & 0xFF;
210     } else {
211         rc = read(_base_spe_open_if_closed(spectx, FD_WBOX_STAT, 0), &ret, 4);
212         if (rc != 4)
213             ret = -1;
214     }
215
216     return ret;
217 }
218 }
```

### 3.19.1.2 `int _base_spe_in_mbox_write(spe_context_ptr_t spectx, unsigned int *mbox_data, int count, int behavior_flag)`

Definition at line 112 of file `mbox.c`.

References `_base_spe_open_if_closed()`, `spe_context::base_private`, `FD_WBOX`, `FD_WBOX_NB`, `spe_context_base_priv::flags`, `SPE_MAP_PS`, `SPE_MBOX_ALL_BLOCKING`, `SPE_MBOX_ANY_BLOCKING`, and `SPE_MBOX_ANY_NONBLOCKING`.

```

116 {
117     int rc;
118     int total;
119     unsigned int *aux;
120     struct pollfd fds;
121
122     if (mbox_data == NULL || count < 1){
123         errno = EINVAL;
124         return -1;
125     }
126
127     switch (behavior_flag) {
128     case SPE_MBOX_ALL_BLOCKING: // write all, even if blocking
129         total = rc = 0;
130         if (spectx->base_private->flags & SPE_MAP_PS) {
131             do {
132                 aux = mbox_data + total;
133                 total += _base_spe_in_mbox_write_ps(spectx, aux, count - total);
134                 if (total < count) { // we could not write everything, wait for space
135                     fds.fd = _base_spe_open_if_closed(spectx, FD_WBOX, 0);
136                     fds.events = POLLOUT;
137                     rc = poll(&fds, 1, -1);
138                     if (rc == -1 )
139                         return -1;
140                 }
141             } while (total < count);
142         } else {
```

```

143         while (total < 4*count) {
144             rc = write(_base_spe_open_if_closed(spectx,FD_WBOX, 0),
145                       (const char *)mbox_data + total, 4*count - total);
146             if (rc == -1) {
147                 break;
148             }
149             total += rc;
150         }
151         total /=4;
152     }
153     break;
154
155     case SPE_MBOX_ANY_BLOCKING: // write at least one, even if blocking
156         total = rc = 0;
157         if (spectx->base_private->flags & SPE_MAP_PS) {
158             do {
159                 total = _base_spe_in_mbox_write_ps(spectx, mbox_data, count);
160                 if (total == 0) { // we could not anything, wait for space
161                     fds.fd = _base_spe_open_if_closed(spectx, FD_WBOX, 0);
162                     fds.events = POLLOUT;
163                     rc = poll(&fds, 1, -1);
164                     if (rc == -1 )
165                         return -1;
166                 }
167             } while (total == 0);
168         } else {
169             rc = write(_base_spe_open_if_closed(spectx,FD_WBOX, 0), mbox_data, 4*count);
170             total = rc/4;
171         }
172         break;
173
174     case SPE_MBOX_ANY_NONBLOCKING: // only write, if non blocking
175         total = rc = 0;
176         // write directly if we map the PS else write via spufs
177         if (spectx->base_private->flags & SPE_MAP_PS) {
178             total = _base_spe_in_mbox_write_ps(spectx, mbox_data, count);
179         } else {
180             rc = write(_base_spe_open_if_closed(spectx,FD_WBOX_NB, 0), mbox_data, 4*count);
181             if (rc == -1 && errno == EAGAIN) {
182                 rc = 0;
183                 errno = 0;
184             }
185             total = rc/4;
186         }
187         break;
188
189     default:
190         errno = EINVAL;
191         return -1;
192 }
193
194 if (rc == -1) {
195     errno = EIO;
196     return -1;
197 }
198
199 return total;
200 }

```

Here is the call graph for this function:



### 3.19.1.3 `int _base_spe_out_intr_mbox_read (spe_context_ptr_t spectx, unsigned int mbox_data[], int count, int behavior_flag)`

The `_base_spe_out_intr_mbox_read` function reads the contents of the SPE outbound interrupting mailbox for the SPE context.

Definition at line 255 of file `mbox.c`.

```

259 {
260     int rc;
261     int total;
262
263     if (mbox_data == NULL || count < 1){
264         errno = EINVAL;
265         return -1;
266     }
267
268     switch (behavior_flag) {
269     case SPE_MBOX_ALL_BLOCKING: // read all, even if blocking
270         total = rc = 0;
271         while (total < 4*count) {
272             rc = read(_base_spe_open_if_closed(spectx, FD_IBOX, 0),
273                     (char *)mbox_data + total, 4*count - total);
274             if (rc == -1) {
275                 break;
276             }
277             total += rc;
278         }
279         break;
280
281     case SPE_MBOX_ANY_BLOCKING: // read at least one, even if blocking
282         total = rc = read(_base_spe_open_if_closed(spectx, FD_IBOX, 0), mbox_data, 4*count);
283         break;
284
285     case SPE_MBOX_ANY_NONBLOCKING: // only read, if non blocking
286         rc = read(_base_spe_open_if_closed(spectx, FD_IBOX_NB, 0), mbox_data, 4*count);
287         if (rc == -1 && errno == EAGAIN) {
288             rc = 0;
289             errno = 0;
290         }
291         total = rc;
292         break;
293
294     default:
295         errno = EINVAL;
296         return -1;
297     }
298
299     if (rc == -1) {
300         errno = EIO;
301         return -1;
302     }
303
304     return rc / 4;
305 }
```

### 3.19.1.4 `int _base_spe_out_intr_mbox_status (spe_context_ptr_t spectx)`

The `_base_spe_out_intr_mbox_status` function fetches the status of the SPU outbound interrupt mailbox for the SPE thread specified by the `speid` 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:**

*spectx* Specifies the SPE context whose mailbox status is to be read.

**Returns:**

On success, returns the current status of the mailbox, respectively. On failure, -1 is returned.

**See also:**

spe\_read\_out\_mbox, spe\_write\_in\_mbox, read (2)

Definition at line 238 of file mbox.c.

```

239 {
240     int rc, ret;
241     volatile struct spe_spu_control_area *cntl_area =
242         spectx->base_private->cntl_mmap_base;
243
244     if (spectx->base_private->flags & SPE_MAP_PS) {
245         ret = (cntl_area->SPU_Mbox_Stat >> 16) & 0xFF;
246     } else {
247         rc = read(_base_spe_open_if_closed(spectx, FD_IBOX_STAT, 0), &ret, 4);
248         if (rc != 4)
249             ret = -1;
250     }
251     return ret;
252 }
253 
```

### 3.19.1.5 int \_base\_spe\_out\_mbox\_read (spe\_context\_ptr\_t *spectx*, unsigned int *mbox\_data*[], int *count*)

The \_base\_spe\_out\_mbox\_read function reads the contents of the SPE outbound interrupting mailbox for the SPE thread speid.

The call will not block until the read request is satisfied, but instead return up to count currently available mailbox entries.

spe\_stat\_out\_intr\_mbox can be called to ensure that data is available prior to reading the outbound interrupting mailbox.

**Parameters:**

*spectx* Specifies the SPE thread whose outbound mailbox is to be read.

*mbox\_data*

*count*

**Return values:**

**>0** the number of 32-bit mailbox messages read

**=0** no data available

**-1** error condition and errno is set

Possible values for errno:

EINVAL speid is invalid

Exxxx what else do we need here??

Definition at line 58 of file mbox.c.



```

61 {
62     int rc;
63
64     if (mbox_data == NULL || count < 1){
65         errno = EINVAL;
66         return -1;
67     }
68
69     if (spectx->base_private->flags & SPE_MAP_PS) {
70         rc = _base_spe_out_mbox_read_ps(spectx, mbox_data, count);
71     } else {
72         rc = read(_base_spe_open_if_closed(spectx, FD_MBOX, 0), mbox_data, count*4);
73         DEBUG_PRINTF("%s read rc: %d\n", __FUNCTION__, rc);
74         if (rc != -1) {
75             rc /= 4;
76         } else {
77             if (errno == EAGAIN ) { // no data ready to be read
78                 errno = 0;
79                 rc = 0;
80             }
81         }
82     }
83     return rc;
84 }

```

#### 3.19.1.6 int \_base\_spe\_out\_mbox\_status (spe\_context\_ptr\_t spectx)

The `_base_spe_out_mbox_status` function fetches the status of the SPU outbound mailbox for the SPE thread specified by the `speid` 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:

***spectx*** Specifies the SPE context whose mailbox status is to be read.

##### Returns:

On success, returns the current status of the mailbox, respectively. On failure, -1 is returned.

##### See also:

`spe_read_out_mbox`, `spe_write_in_mbox`, `read (2)`

Definition at line 220 of file `mbox.c`.

```

221 {
222     int rc, ret;
223     volatile struct spe_spu_control_area *cntl_area =
224         spectx->base_private->cntl_mmap_base;
225
226     if (spectx->base_private->flags & SPE_MAP_PS) {
227         ret = cntl_area->SPU_Mbox_Stat & 0xFF;
228     } else {
229         rc = read(_base_spe_open_if_closed(spectx, FD_MBOX_STAT, 0), &ret, 4);
230         if (rc != 4)
231             ret = -1;
232     }
233
234     return ret;
235
236 }

```

### 3.19.1.7 `int _base_spe_signal_write (spe_context_ptr_t spectx, unsigned int signal_reg, unsigned int data)`

The `_base_spe_signal_write` function writes data to the signal notification register specified by `signal_reg` for the SPE thread specified by the `speid` parameter.

#### Parameters:

***spectx*** Specifies the SPE context whose signal register is to be written to.

***signal\_reg*** Specified 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.

#### Returns:

On success, `spe_write_signal` returns 0. On failure, -1 is returned.

#### See also:

`spe_get_ps_area`, `spe_write_in_mbox`

Definition at line 307 of file `mbox.c`.

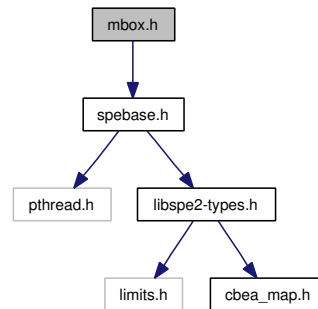
```

310 {
311     int rc;
312
313     if (spectx->base_private->flags & SPE_MAP_PS) {
314         if (signal_reg == SPE_SIG_NOTIFY_REG_1) {
315             spe_sig_notify_1_area_t *sig = spectx->base_private->signal1_mmap_base;
316
317             sig->SPU_Sig_Notify_1 = data;
318         } else if (signal_reg == SPE_SIG_NOTIFY_REG_2) {
319             spe_sig_notify_2_area_t *sig = spectx->base_private->signal2_mmap_base;
320
321             sig->SPU_Sig_Notify_2 = data;
322         } else {
323             errno = EINVAL;
324             return -1;
325         }
326         rc = 0;
327     } else {
328         if (signal_reg == SPE_SIG_NOTIFY_REG_1)
329             rc = write(_base_spe_open_if_closed(spectx, FD_SIG1, 0), &data, 4);
330         else if (signal_reg == SPE_SIG_NOTIFY_REG_2)
331             rc = write(_base_spe_open_if_closed(spectx, FD_SIG2, 0), &data, 4);
332         else {
333             errno = EINVAL;
334             return -1;
335         }
336
337         if (rc == 4)
338             rc = 0;
339
340         if (signal_reg == SPE_SIG_NOTIFY_REG_1)
341             _base_spe_close_if_open(spectx, FD_SIG1);
342         else if (signal_reg == SPE_SIG_NOTIFY_REG_2)
343             _base_spe_close_if_open(spectx, FD_SIG2);
344     }
345
346     return rc;
347 }
```

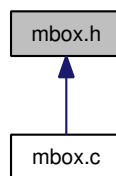
## 3.20 mbox.h File Reference

```
#include "spebase.h"
```

Include dependency graph for mbox.h:



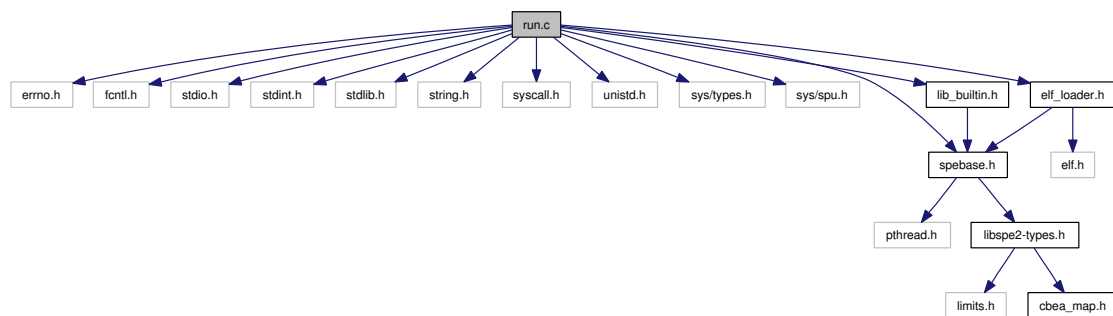
This graph shows which files directly or indirectly include this file:



## 3.21 run.c File Reference

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <syscall.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/spu.h>
#include "elf_loader.h"
#include "lib_builtin.h"
#include "spebase.h"
```

Include dependency graph for run.c:



## Data Structures

- struct [spe\\_context\\_info](#)

## Defines

- #define [GNU\\_SOURCE](#) 1

## Functions

- int [\\_base\\_spe\\_context\\_run](#) ([spe\\_context\\_ptr\\_t](#) spe, unsigned int \*entry, unsigned int runflags, void \*argp, void \*envp, [spe\\_stop\\_info\\_t](#) \*stopinfo)

## Variables

- \_\_thread struct [spe\\_context\\_info](#) \* [\\_\\_spe\\_current\\_active\\_context](#)

### 3.21.1 Define Documentation

#### 3.21.1.1 #define GNU\_SOURCE 1

Definition at line 20 of file run.c.

### 3.21.2 Function Documentation

#### 3.21.2.1 `int _base_spe_context_run (spe_context_ptr_t spe, unsigned int * entry, unsigned int runflags, void * argp, void * envp, spe_stop_info_t * stopinfo)`

`_base_spe_context_run` starts execution of an SPE context with a loaded image

##### Parameters:

*spectx* Specifies the SPE context

*entry* entry point for the SPE program. If set to 0, entry point is determined by the ELF loader.

*runflags* valid values are:

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 do not use built in library functions.

*argp* An (optional) pointer to application specific data, and is passed as the second parameter to the SPE program.

*envp* An (optional) pointer to environment specific data, and is passed as the third parameter to the SPE program.

Definition at line 98 of file run.c.

Referenced by `_event_spe_context_run()`.

```

101 {
102     int retval = 0, run_rc;
103     unsigned int run_status, tmp_entry;
104     spe_stop_info_t stopinfo_buf;
105     struct spe_context_info this_context_info __attribute__((cleanup(cleanupspeinfo)));
106
107     /* If the caller hasn't set a stopinfo buffer, provide a buffer on the
108      * stack instead. */
109     if (!stopinfo)
110         stopinfo = &stopinfo_buf;
111
112
113     /* In emulated isolated mode, the npc will always return as zero.
114      * use our private entry point instead */
115     if (spe->base_private->flags & SPE_ISOLATE_EMULATE)
116         tmp_entry = spe->base_private->emulated_entry;
117
118     else if (*entry == SPE_DEFAULT_ENTRY)
119         tmp_entry = spe->base_private->entry;
120     else
121         tmp_entry = *entry;
122
123     /* If we're starting the SPE binary from its original entry point,
124      * setup the arguments to main() */
125     if (tmp_entry == spe->base_private->entry &&
126         !(spe->base_private->flags &
127           (SPE_ISOLATE | SPE_ISOLATE_EMULATE))) {
128
```

```

129         addr64 argp64, envp64, tid64, ls64;
130         unsigned int regs[128][4];
131
132         /* setup parameters */
133         argp64.u1l = (uint64_t)(unsigned long)argp;
134         envp64.u1l = (uint64_t)(unsigned long)envp;
135         tid64.u1l = (uint64_t)(unsigned long)spe;
136
137         /* make sure the register values are 0 */
138         memset(regs, 0, sizeof(regs));
139
140         /* set sensible values for stack_ptr and stack_size */
141         regs[1][0] = (unsigned int) LS_SIZE - 16;          /* stack_ptr */
142         regs[2][0] = 0;                                     /* stack_size
143
144         if (runflags & SPE_RUN_USER_REGS) {
145             /* When SPE_USER_REGS is set, argp points to an array
146              * of 3x128b registers to be passed directly to the SPE
147              * program.
148              */
149             memcpy(regs[3], argp, sizeof(unsigned int) * 12);
150         } else {
151             regs[3][0] = tid64.ui[0];
152             regs[3][1] = tid64.ui[1];
153
154             regs[4][0] = argp64.ui[0];
155             regs[4][1] = argp64.ui[1];
156
157             regs[5][0] = envp64.ui[0];
158             regs[5][1] = envp64.ui[1];
159         }
160
161         /* Store the LS base address in R6 */
162         ls64.u1l = (uint64_t)(unsigned long)spe->base_private->mem_mmap_base;
163         regs[6][0] = ls64.ui[0];
164         regs[6][1] = ls64.ui[1];
165
166         if (set_regs(spe, regs))
167             return -1;
168     }
169
170     /*Leave a trail of breadcrumbs for the debugger to follow */
171     if (!__spe_current_active_context) {
172         __spe_current_active_context = &this_context_info;
173         if (!__spe_current_active_context)
174             return -1;
175         __spe_current_active_context->prev = NULL;
176     } else {
177         struct spe_context_info *newinfo;
178         newinfo = &this_context_info;
179         if (!newinfo)
180             return -1;
181         newinfo->prev = __spe_current_active_context;
182         __spe_current_active_context = newinfo;
183     }
184     /*remember the ls-addr*/
185     __spe_current_active_context->spe_id = spe->base_private->fd_spe_dir;
186
187 do_run:
188     /*Remember the npc value*/
189     __spe_current_active_context->npc = tmp_entry;
190
191     /* run SPE context */
192     run_rc = spu_run(spe->base_private->fd_spe_dir,
193                     &tmp_entry, &run_status);
194
195     /*Remember the npc value*/

```

```

196     __spe_current_active_context->npc = tmp_entry;
197     __spe_current_active_context->status = run_status;
198
199     DEBUG_PRINTF("spu_run returned run_rc=0x%08x, entry=0x%04x, "
200                 "ext_status=0x%04x.\n", run_rc, tmp_entry, run_status);
201
202     /* set up return values and stopinfo according to spu_run exit
203      * conditions. This is overwritten on error.
204      */
205     stopinfo->spu_status = run_rc;
206
207     if (spe->base_private->flags & SPE_ISOLATE_EMULATE) {
208         /* save the entry point, and pretend that the npc is zero */
209         spe->base_private->emulated_entry = tmp_entry;
210         *entry = 0;
211     } else {
212         *entry = tmp_entry;
213     }
214
215     /* Return with stopinfo set on syscall error paths */
216     if (run_rc == -1) {
217         DEBUG_PRINTF("spu_run returned error %d, errno=%d\n",
218                     run_rc, errno);
219         stopinfo->stop_reason = SPE_RUNTIME_FATAL;
220         stopinfo->result.spe_runtime_fatal = errno;
221         retval = -1;
222
223         /* For isolated contexts, pass EPERM up to the
224          * caller.
225          */
226         if (!(spe->base_private->flags & SPE_ISOLATE
227             && errno == EPERM))
228             errno = EFAULT;
229
230     } else if (run_rc & SPE_SPU_INVALID_INSTR) {
231         DEBUG_PRINTF("SPU has tried to execute an invalid "
232                     "instruction. %d\n", run_rc);
233         stopinfo->stop_reason = SPE_RUNTIME_ERROR;
234         stopinfo->result.spe_runtime_error = SPE_SPU_INVALID_INSTR;
235         errno = EFAULT;
236         retval = -1;
237
238     } else if ((spe->base_private->flags & SPE_EVENTS_ENABLE) && run_status) {
239         /* Report asynchronous error if return val are set and
240          * SPU events are enabled.
241          */
242         stopinfo->stop_reason = SPE_RUNTIME_EXCEPTION;
243         stopinfo->result.spe_runtime_exception = run_status;
244         stopinfo->spu_status = -1;
245         errno = EIO;
246         retval = -1;
247
248     } else if (run_rc & SPE_SPU_STOPPED_BY_STOP) {
249         /* Stop & signals are broken down into three groups
250          * 1. SPE library call
251          * 2. SPE user defined stop & signal
252          * 3. SPE program end.
253          *
254          * These groups are signified by the 14-bit stop code:
255          */
256         int stopcode = (run_rc >> 16) & 0x3fff;
257
258         /* Check if this is a library callback, and callbacks are
259          * allowed (ie, running without SPE_NO_CALLBACKS)
260          */
261         if ((stopcode & 0xff00) == SPE_PROGRAM_LIBRARY_CALL
262             && !(runflags & SPE_NO_CALLBACKS)) {

```

```

263
264         int callback_rc, callback_number = stopcode & 0xff;
265
266         /* execute library callback */
267         DEBUG_PRINTF("SPE library call: %d\n", callback_number);
268         callback_rc = _base_spe_handle_library_callback(spe,
269                                                         callback_number, *entry);
270
271         if (callback_rc) {
272             /* library callback failed; set errno and
273              * return immediately */
274             DEBUG_PRINTF("SPE library call failed: %d\n",
275                         callback_rc);
276             stopinfo->stop_reason = SPE_CALLBACK_ERROR;
277             stopinfo->result.spe_callback_error =
278                 callback_rc;
279             errno = EFAULT;
280             retval = -1;
281         } else {
282             /* successful library callback - restart the SPE
283              * program at the next instruction */
284             tmp_entry += 4;
285             goto do_run;
286         }
287
288     } else if ((stopcode & 0xff00) == SPE_PROGRAM_NORMAL_END) {
289         /* The SPE program has exited by exit(X) */
290         stopinfo->stop_reason = SPE_EXIT;
291         stopinfo->result.spe_exit_code = stopcode & 0xff;
292
293         if (spe->base_private->flags & SPE_ISOLATE) {
294             /* Issue an isolated exit, and re-run the SPE.
295              * We should see a return value without the
296              * 0x80 bit set. */
297             if (!issue_isolated_exit(spe))
298                 goto do_run;
299             retval = -1;
300         }
301
302     } else if ((stopcode & 0xffff0) == SPE_PROGRAM_ISOLATED_STOP) {
303
304         /* 0x2206: isolated app has been loaded by loader;
305          * provide a hook for the debugger to catch this,
306          * and restart
307          */
308         if (stopcode == SPE_PROGRAM_ISO_LOAD_COMPLETE) {
309             _base_spe_program_load_complete(spe);
310             goto do_run;
311         } else {
312             stopinfo->stop_reason = SPE_ISOLATION_ERROR;
313             stopinfo->result.spe_isolation_error =
314                 stopcode & 0xf;
315         }
316
317     } else if (spe->base_private->flags & SPE_ISOLATE &&
318               !(run_rc & 0x80)) {
319         /* We've successfully exited isolated mode */
320         retval = 0;
321
322     } else {
323         /* User defined stop & signal, including
324          * callbacks when disabled */
325         stopinfo->stop_reason = SPE_STOP_AND_SIGNAL;
326         stopinfo->result.spe_signal_code = stopcode;
327         retval = stopcode;
328     }
329

```



```

330     } else if (run_rc & SPE_SPU_HALT) {
331         DEBUG_PRINTF("SPU was stopped by halt. %d\n", run_rc);
332         stopinfo->stop_reason = SPE_RUNTIME_ERROR;
333         stopinfo->result.spe_runtime_error = SPE_SPU_HALT;
334         errno = EFAULT;
335         retval = -1;
336
337     } else if (run_rc & SPE_SPU_WAITING_ON_CHANNEL) {
338         DEBUG_PRINTF("SPU is waiting on channel. %d\n", run_rc);
339         stopinfo->stop_reason = SPE_RUNTIME_EXCEPTION;
340         stopinfo->result.spe_runtime_exception = run_status;
341         stopinfo->spu_status = -1;
342         errno = EIO;
343         retval = -1;
344
345     } else if (run_rc & SPE_SPU_INVALID_CHANNEL) {
346         DEBUG_PRINTF("SPU has tried to access an invalid "
347                     "channel. %d\n", run_rc);
348         stopinfo->stop_reason = SPE_RUNTIME_ERROR;
349         stopinfo->result.spe_runtime_error = SPE_SPU_INVALID_CHANNEL;
350         errno = EFAULT;
351         retval = -1;
352
353     } else {
354         DEBUG_PRINTF("spu_run returned invalid data: 0x%04x\n", run_rc);
355         stopinfo->stop_reason = SPE_RUNTIME_FATAL;
356         stopinfo->result.spe_runtime_fatal = -1;
357         stopinfo->spu_status = -1;
358         errno = EFAULT;
359         retval = -1;
360
361     }
362
363     freespeinfo();
364     return retval;
365 }

```

### 3.21.3 Variable Documentation

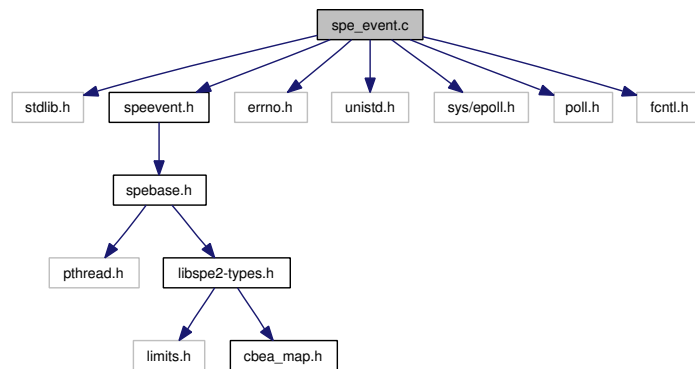
#### 3.21.3.1 `__thread struct spe_context_info* __spe_current_active_context`

Referenced by `_base_spe_context_run()`.

## 3.22 spe\_event.c File Reference

```
#include <stdlib.h>
#include "speevent.h"
#include <errno.h>
#include <unistd.h>
#include <sys/epoll.h>
#include <poll.h>
#include <fcntl.h>
```

Include dependency graph for spe\_event.c:



### Defines

- `#define __SPE_EVENT_ALL`
- `#define __SPE_EPOLL_SIZE 10`
- `#define __SPE_EPOLL_FD_GET(handler) (*(int*)(handler))`
- `#define __SPE_EPOLL_FD_SET(handler, fd) (*(int*)(handler) = (fd))`
- `#define __SPE_EVENT_CONTEXT_PRIV_GET(spe) ( (spe_context_event_priv_ptr_t)(spe) → event_private)`
- `#define __SPE_EVENT_CONTEXT_PRIV_SET(spe, evctx) ( (spe) → event_private = (evctx) )`
- `#define __SPE_EVENTS_ENABLED(spe) ((spe) → base_private → flags & SPE_EVENTS_ENABLE)`

### Functions

- `void _event_spe_context_lock (spe_context_ptr_t spe)`
- `void _event_spe_context_unlock (spe_context_ptr_t spe)`
- `int _event_spe_stop_info_read (spe_context_ptr_t spe, spe_stop_info_t *stopinfo)`
- `spe_event_handler_ptr_t _event_spe_event_handler_create (void)`
- `int _event_spe_event_handler_destroy (spe_event_handler_ptr_t evhandler)`
- `int _event_spe_event_handler_register (spe_event_handler_ptr_t evhandler, spe_event_unit_t *event)`
- `int _event_spe_event_handler_deregister (spe_event_handler_ptr_t evhandler, spe_event_unit_t *event)`

- `int _event_spe_event_wait (spe_event_handler_ptr_t evhandler, spe_event_unit_t *events, int max_events, int timeout)`
- `int _event_spe_context_finalize (spe_context_ptr_t spe)`
- `struct spe_context_event_priv * _event_spe_context_initialize (spe_context_ptr_t spe)`
- `int _event_spe_context_run (spe_context_ptr_t spe, unsigned int *entry, unsigned int runflags, void *argp, void *envp, spe_stop_info_t *stopinfo)`

### 3.22.1 Define Documentation

#### 3.22.1.1 `#define __SPE_EPOLL_FD_GET(handler) (*(int*)(handler))`

Definition at line 37 of file `spe_event.c`.

Referenced by `_event_spe_event_handler_deregister()`, `_event_spe_event_handler_destroy()`, `_event_spe_event_handler_register()`, and `_event_spe_event_wait()`.

#### 3.22.1.2 `#define __SPE_EPOLL_FD_SET(handler, fd) (*(int*)(handler) = (fd))`

Definition at line 38 of file `spe_event.c`.

Referenced by `_event_spe_event_handler_create()`.

#### 3.22.1.3 `#define __SPE_EPOLL_SIZE 10`

Definition at line 35 of file `spe_event.c`.

Referenced by `_event_spe_event_handler_create()`.

#### 3.22.1.4 `#define __SPE_EVENT_ALL`

**Value:**

```
( SPE_EVENT_OUT_INTR_MBOX | SPE_EVENT_IN_MBOX | \
  SPE_EVENT_TAG_GROUP | SPE_EVENT_SPE_STOPPED )
```

Definition at line 31 of file `spe_event.c`.

Referenced by `_event_spe_event_handler_deregister()`, and `_event_spe_event_handler_register()`.

#### 3.22.1.5 `#define __SPE_EVENT_CONTEXT_PRIV_GET(spe) ( (spe_context_event_priv_ptr_t)(spe) → event_private)`

Definition at line 40 of file `spe_event.c`.

Referenced by `_event_spe_context_finalize()`, `_event_spe_context_lock()`, `_event_spe_context_run()`, `_event_spe_context_unlock()`, `_event_spe_event_handler_deregister()`, `_event_spe_event_handler_register()`, and `_event_spe_stop_info_read()`.

#### 3.22.1.6 `#define __SPE_EVENT_CONTEXT_PRIV_SET(spe, evctx) ( (spe) → event_private = (evctx) )`

Definition at line 42 of file `spe_event.c`.

Referenced by `_event_spe_context_finalize()`.

### 3.22.1.7 `#define __SPE_EVENTS_ENABLED(spe) ((spe) → base_private → flags & SPE_EVENTS_ENABLE)`

Definition at line 45 of file `spe_event.c`.

Referenced by `_event_spe_event_handler_deregister()`, and `_event_spe_event_handler_register()`.

## 3.22.2 Function Documentation

### 3.22.2.1 `int _event_spe_context_finalize (spe_context_ptr_t spe)`

Definition at line 416 of file `spe_event.c`.

```

417 {
418     spe_context_event_priv_ptr_t evctx;
419
420     if (!spe) {
421         errno = ESRCH;
422         return -1;
423     }
424
425     evctx = __SPE_EVENT_CONTEXT_PRIV_GET(spe);
426     __SPE_EVENT_CONTEXT_PRIV_SET(spe, NULL);
427
428     close(evctx->stop_event_pipe[0]);
429     close(evctx->stop_event_pipe[1]);
430
431     pthread_mutex_destroy(&evctx->lock);
432     pthread_mutex_destroy(&evctx->stop_event_read_lock);
433
434     free(evctx);
435
436     return 0;
437 }
```

### 3.22.2.2 `struct spe_context_event_priv* _event_spe_context_initialize (spe_context_ptr_t spe)` [read]

Definition at line 439 of file `spe_event.c`.

```

440 {
441     spe_context_event_priv_ptr_t evctx;
442     int rc;
443     int i;
444
445     evctx = calloc(1, sizeof(*evctx));
446     if (!evctx) {
447         return NULL;
448     }
449
450     rc = pipe(evctx->stop_event_pipe);
451     if (rc == -1) {
452         free(evctx);
453         return NULL;
454     }
455     rc = fcntl(evctx->stop_event_pipe[0], F_GETFL);
```

```

456     if (rc != -1) {
457         rc = fcntl(evctx->stop_event_pipe[0], F_SETFL, rc | O_NONBLOCK);
458     }
459     if (rc == -1) {
460         close(evctx->stop_event_pipe[0]);
461         close(evctx->stop_event_pipe[1]);
462         free(evctx);
463         errno = EIO;
464         return NULL;
465     }
466
467     for (i = 0; i < sizeof(evctx->events) / sizeof(evctx->events[0]); i++) {
468         evctx->events[i].spe = spe;
469     }
470
471     pthread_mutex_init(&evctx->lock, NULL);
472     pthread_mutex_init(&evctx->stop_event_read_lock, NULL);
473
474     return evctx;
475 }

```

### 3.22.2.3 void \_event\_spe\_context\_lock (spe\_context\_ptr\_t spe)

Definition at line 49 of file spe\_event.c.

Referenced by \_event\_spe\_event\_handler\_deregister(), \_event\_spe\_event\_handler\_register(), and \_event\_spe\_event\_wait().

```

50 {
51     pthread_mutex_lock(&__SPE_EVENT_CONTEXT_PRIV_GET(spe)->lock);
52 }

```

### 3.22.2.4 int \_event\_spe\_context\_run (spe\_context\_ptr\_t spe, unsigned int \*entry, unsigned int runflags, void \*argp, void \*envp, spe\_stop\_info\_t \*stopinfo)

Definition at line 477 of file spe\_event.c.

```

478 {
479     spe_context_event_priv_ptr_t evctx;
480     spe_stop_info_t stopinfo_buf;
481     int rc;
482
483     if (!stopinfo) {
484         stopinfo = &stopinfo_buf;
485     }
486     rc = _base_spe_context_run(spe, entry, runflags, argp, envp, stopinfo);
487
488     evctx = __SPE_EVENT_CONTEXT_PRIV_GET(spe);
489     if (write(evctx->stop_event_pipe[1], stopinfo, sizeof(*stopinfo)) != sizeof(*stopinfo)) {
490         /* error check. */
491     }
492
493     return rc;
494 }

```

### 3.22.2.5 void \_event\_spe\_context\_unlock (spe\_context\_ptr\_t spe)

Definition at line 54 of file spe\_event.c.

Referenced by `_event_spe_event_handler_deregister()`, `_event_spe_event_handler_register()`, and `_event_spe_event_wait()`.

```
55 {
56     pthread_mutex_unlock(&__SPE_EVENT_CONTEXT_PRIV_GET(spe)->lock);
57 }
```

### 3.22.2.6 `spe_event_handler_ptr_t _event_spe_event_handler_create(void)`

Definition at line 110 of file `spe_event.c`.

```
111 {
112     int epfd;
113     spe_event_handler_t *evhandler;
114
115     evhandler = calloc(1, sizeof(*evhandler));
116     if (!evhandler) {
117         return NULL;
118     }
119
120     epfd = epoll_create(__SPE_EPOLL_SIZE);
121     if (epfd == -1) {
122         free(evhandler);
123         return NULL;
124     }
125
126     __SPE_EPOLL_FD_SET(evhandler, epfd);
127
128     return evhandler;
129 }
```

### 3.22.2.7 `int _event_spe_event_handler_deregister(spe_event_handler_ptr_t evhandler, spe_event_unit_t *event)`

Definition at line 273 of file `spe_event.c`.

```
274 {
275     int epfd;
276     const int ep_op = EPOLL_CTL_DEL;
277     spe_context_event_priv_ptr_t evctx;
278     int fd;
279
280     if (!evhandler) {
281         errno = ESRCH;
282         return -1;
283     }
284     if (!event || !event->spe) {
285         errno = EINVAL;
286         return -1;
287     }
288     if (!__SPE_EVENTS_ENABLED(event->spe)) {
289         errno = ENOTSUP;
290         return -1;
291     }
292
293     epfd = __SPE_EPOLL_FD_GET(evhandler);
294     evctx = __SPE_EVENT_CONTEXT_PRIV_GET(event->spe);
295
296     if (event->events & ~__SPE_EVENT_ALL) {
```

```

297     errno = ENOTSUP;
298     return -1;
299 }
300
301 _event_spe_context_lock(event->spe); /* for spe->event_private->events */
302
303 if (event->events & SPE_EVENT_OUT_INTR_MBOX) {
304     fd = __base_spe_event_source_acquire(event->spe, FD_IBOX);
305     if (fd == -1) {
306         _event_spe_context_unlock(event->spe);
307         return -1;
308     }
309     if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
310         _event_spe_context_unlock(event->spe);
311         return -1;
312     }
313     evctx->events[__SPE_EVENT_OUT_INTR_MBOX].events = 0;
314 }
315
316 if (event->events & SPE_EVENT_IN_MBOX) {
317     fd = __base_spe_event_source_acquire(event->spe, FD_WBOX);
318     if (fd == -1) {
319         _event_spe_context_unlock(event->spe);
320         return -1;
321     }
322     if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
323         _event_spe_context_unlock(event->spe);
324         return -1;
325     }
326     evctx->events[__SPE_EVENT_IN_MBOX].events = 0;
327 }
328
329 if (event->events & SPE_EVENT_TAG_GROUP) {
330     fd = __base_spe_event_source_acquire(event->spe, FD_MFC);
331     if (fd == -1) {
332         _event_spe_context_unlock(event->spe);
333         return -1;
334     }
335     if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
336         _event_spe_context_unlock(event->spe);
337         return -1;
338     }
339     evctx->events[__SPE_EVENT_TAG_GROUP].events = 0;
340 }
341
342 if (event->events & SPE_EVENT_SPE_STOPPED) {
343     fd = evctx->stop_event_pipe[0];
344     if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
345         _event_spe_context_unlock(event->spe);
346         return -1;
347     }
348     evctx->events[__SPE_EVENT_SPE_STOPPED].events = 0;
349 }
350
351 _event_spe_context_unlock(event->spe);
352
353 return 0;
354 }

```

### 3.22.2.8 int \_event\_spe\_event\_handler\_destroy (spe\_event\_handler\_ptr\_t evhandler)

Definition at line 135 of file spe\_event.c.

```
136 {
```

```

137     int epfd;
138
139     if (!evhandler) {
140         errno = ESRCH;
141         return -1;
142     }
143
144     epfd = __SPE_EPOLL_FD_GET(evhandler);
145     close(epfd);
146
147     free(evhandler);
148     return 0;
149 }

```

### 3.22.2.9 `int _event_spe_event_handler_register (spe_event_handler_ptr_t evhandler, spe_event_unit_t * event)`

Definition at line 155 of file `spe_event.c`.

```

156 {
157     int epfd;
158     const int ep_op = EPOLL_CTL_ADD;
159     spe_context_event_priv_ptr_t evctx;
160     spe_event_unit_t *ev_buf;
161     struct epoll_event ep_event;
162     int fd;
163
164     if (!evhandler) {
165         errno = ESRCH;
166         return -1;
167     }
168     if (!event || !event->spe) {
169         errno = EINVAL;
170         return -1;
171     }
172     if (!__SPE_EVENTS_ENABLED(event->spe)) {
173         errno = ENOTSUP;
174         return -1;
175     }
176
177     epfd = __SPE_EPOLL_FD_GET(evhandler);
178     evctx = __SPE_EVENT_CONTEXT_PRIV_GET(event->spe);
179
180     if (event->events & ~__SPE_EVENT_ALL) {
181         errno = ENOTSUP;
182         return -1;
183     }
184
185     _event_spe_context_lock(event->spe); /* for spe->event_private->events */
186
187     if (event->events & SPE_EVENT_OUT_INTR_MBOX) {
188         fd = __base_spe_event_source_acquire(event->spe, FD_IBOX);
189         if (fd == -1) {
190             _event_spe_context_unlock(event->spe);
191             return -1;
192         }
193
194         ev_buf = &evctx->events[__SPE_EVENT_OUT_INTR_MBOX];
195         ev_buf->events = SPE_EVENT_OUT_INTR_MBOX;
196         ev_buf->data = event->data;
197
198         ep_event.events = EPOLLIN;
199         ep_event.data.ptr = ev_buf;
200         if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {

```



```

201     _event_spe_context_unlock(event->spe);
202     return -1;
203 }
204 }
205
206 if (event->events & SPE_EVENT_IN_MBOX) {
207     fd = __base_spe_event_source_acquire(event->spe, FD_WBOX);
208     if (fd == -1) {
209         _event_spe_context_unlock(event->spe);
210         return -1;
211     }
212
213     ev_buf = &evctx->events[__SPE_EVENT_IN_MBOX];
214     ev_buf->events = SPE_EVENT_IN_MBOX;
215     ev_buf->data = event->data;
216
217     ep_event.events = EPOLLOUT;
218     ep_event.data.ptr = ev_buf;
219     if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
220         _event_spe_context_unlock(event->spe);
221         return -1;
222     }
223 }
224
225 if (event->events & SPE_EVENT_TAG_GROUP) {
226     fd = __base_spe_event_source_acquire(event->spe, FD_MFC);
227     if (fd == -1) {
228         _event_spe_context_unlock(event->spe);
229         return -1;
230     }
231
232     if (event->spe->base_private->flags & SPE_MAP_PS) {
233         _event_spe_context_unlock(event->spe);
234         errno = ENOTSUP;
235         return -1;
236     }
237
238     ev_buf = &evctx->events[__SPE_EVENT_TAG_GROUP];
239     ev_buf->events = SPE_EVENT_TAG_GROUP;
240     ev_buf->data = event->data;
241
242     ep_event.events = EPOLLIN;
243     ep_event.data.ptr = ev_buf;
244     if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
245         _event_spe_context_unlock(event->spe);
246         return -1;
247     }
248 }
249
250 if (event->events & SPE_EVENT_SPE_STOPPED) {
251     fd = evctx->stop_event_pipe[0];
252
253     ev_buf = &evctx->events[__SPE_EVENT_SPE_STOPPED];
254     ev_buf->events = SPE_EVENT_SPE_STOPPED;
255     ev_buf->data = event->data;
256
257     ep_event.events = EPOLLIN;
258     ep_event.data.ptr = ev_buf;
259     if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
260         _event_spe_context_unlock(event->spe);
261         return -1;
262     }
263 }
264
265 _event_spe_context_unlock(event->spe);
266
267 return 0;

```

268 }

### 3.22.2.10 `int _event_spe_event_wait (spe_event_handler_ptr_t evhandler, spe_event_unit_t * events, int max_events, int timeout)`

Definition at line 360 of file `spe_event.c`.

```

361 {
362     int epfd;
363     struct epoll_event *ep_events;
364     int rc;
365
366     if (!evhandler) {
367         errno = ESRCH;
368         return -1;
369     }
370     if (!events || max_events <= 0) {
371         errno = EINVAL;
372         return -1;
373     }
374
375     epfd = __SPE_EPOLL_FD_GET(evhandler);
376
377     ep_events = malloc(sizeof(*ep_events) * max_events);
378     if (!ep_events) {
379         return -1;
380     }
381
382     for ( ; ; ) {
383         rc = epoll_wait(epfd, ep_events, max_events, timeout);
384         if (rc == -1) { /* error */
385             if (errno == EINTR) {
386                 if (timeout >= 0) { /* behave as timeout */
387                     rc = 0;
388                     break;
389                 }
390                 /* else retry */
391             }
392             else {
393                 break;
394             }
395         }
396         else if (rc > 0) {
397             int i;
398             for (i = 0; i < rc; i++) {
399                 spe_event_unit_t *ev = (spe_event_unit_t *) (ep_events[i].data.ptr);
400                 _event_spe_context_lock(ev->spe); /* lock ev itself */
401                 events[i] = *ev;
402                 _event_spe_context_unlock(ev->spe);
403             }
404             break;
405         }
406         else { /* timeout */
407             break;
408         }
409     }
410
411     free(ep_events);
412
413     return rc;
414 }

```

### 3.22.2.11 int \_event\_spe\_stop\_info\_read (spe\_context\_ptr\_t spe, spe\_stop\_info\_t \* stopinfo)

Definition at line 59 of file spe\_event.c.

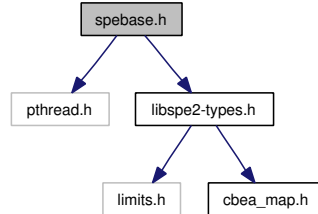
```
60 {
61     spe_context_event_priv_ptr_t evctx;
62     int rc;
63     int fd;
64     size_t total;
65
66     evctx = __SPE_EVENT_CONTEXT_PRIV_GET(spe);
67     fd = evctx->stop_event_pipe[0];
68
69     pthread_mutex_lock(&evctx->stop_event_read_lock); /* for atomic read */
70
71     rc = read(fd, stopinfo, sizeof(*stopinfo));
72     if (rc == -1) {
73         pthread_mutex_unlock(&evctx->stop_event_read_lock);
74         return -1;
75     }
76
77     total = rc;
78     while (total < sizeof(*stopinfo)) { /* this loop will be executed in few cases */
79         struct pollfd fds;
80         fds.fd = fd;
81         fds.events = POLLIN;
82         rc = poll(&fds, 1, -1);
83         if (rc == -1) {
84             if (errno != EINTR) {
85                 break;
86             }
87         }
88         else if (rc == 1) {
89             rc = read(fd, (char *)stopinfo + total, sizeof(*stopinfo) - total);
90             if (rc == -1) {
91                 if (errno != EAGAIN) {
92                     break;
93                 }
94             }
95             else {
96                 total += rc;
97             }
98         }
99     }
100
101     pthread_mutex_unlock(&evctx->stop_event_read_lock);
102
103     return rc == -1 ? -1 : 0;
104 }
```

### 3.23 spebase.h File Reference

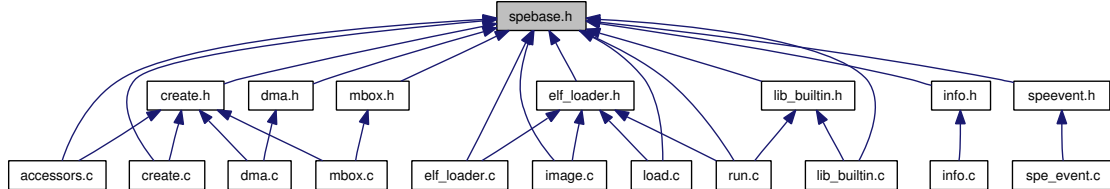
```
#include <pthread.h>
```

```
#include "libspe2-types.h"
```

Include dependency graph for spebase.h:



This graph shows which files directly or indirectly include this file:



#### Data Structures

- struct [spe\\_context\\_base\\_priv](#)
- struct [spe\\_gang\\_context\\_base\\_priv](#)

#### Defines

- `#define __PRINTF(fmt, args...) { fprintf(stderr,fmt , ## args); }`
- `#define DEBUG_PRINTF(fmt, args...)`
- `#define LS_SIZE 0x40000`
- `#define PSMAP_SIZE 0x20000`
- `#define MFC_SIZE 0x1000`
- `#define MSS_SIZE 0x1000`
- `#define CNTL_SIZE 0x1000`
- `#define SIGNAL_SIZE 0x1000`
- `#define MSSYNC_OFFSET 0x00000`
- `#define MFC_OFFSET 0x03000`
- `#define CNTL_OFFSET 0x04000`
- `#define SIGNAL1_OFFSET 0x14000`
- `#define SIGNAL2_OFFSET 0x1c000`
- `#define SPE_EMULATE_PARAM_BUFFER 0x3e000`
- `#define SPE_PROGRAM_NORMAL_END 0x2000`
- `#define SPE_PROGRAM_LIBRARY_CALL 0x2100`
- `#define SPE_PROGRAM_ISOLATED_STOP 0x2200`
- `#define SPE_PROGRAM_ISO_LOAD_COMPLETE 0x2206`

## Enumerations

- enum `fd_name` {  
`FD_MBOX`, `FD_MBOX_STAT`, `FD_IBOX`, `FD_IBOX_NB`,  
`FD_IBOX_STAT`, `FD_WBOX`, `FD_WBOX_NB`, `FD_WBOX_STAT`,  
`FD_SIG1`, `FD_SIG2`, `FD_MFC`, `FD_MSS`,  
`NUM_MBOX_FDS` }

## Functions

- `spe_context_ptr_t _base_spe_context_create` (unsigned int flags, `spe_gang_context_ptr_t` gctx, `spe_context_ptr_t` aff\_spe)
- `spe_gang_context_ptr_t _base_spe_gang_context_create` (unsigned int flags)
- int `_base_spe_program_load` (`spe_context_ptr_t` spectx, `spe_program_handle_t` \*program)
- void `_base_spe_program_load_complete` (`spe_context_ptr_t` spectx)
- int `_base_spe_emulated_loader_present` (void)
- int `_base_spe_context_destroy` (`spe_context_ptr_t` spectx)
- int `_base_spe_gang_context_destroy` (`spe_gang_context_ptr_t` gctx)
- int `_base_spe_context_run` (`spe_context_ptr_t` spe, unsigned int \*entry, unsigned int runflags, void \*argp, void \*envp, `spe_stop_info_t` \*stopinfo)
- int `_base_spe_image_close` (`spe_program_handle_t` \*handle)
- `spe_program_handle_t` \* `_base_spe_image_open` (const char \*filename)
- int `_base_spe_mfcio_put` (`spe_context_ptr_t` spectx, unsigned int ls, void \*ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)
- int `_base_spe_mfcio_putb` (`spe_context_ptr_t` spectx, unsigned int ls, void \*ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)
- int `_base_spe_mfcio_putf` (`spe_context_ptr_t` spectx, unsigned int ls, void \*ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)
- int `_base_spe_mfcio_get` (`spe_context_ptr_t` spectx, unsigned int ls, void \*ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)
- int `_base_spe_mfcio_getb` (`spe_context_ptr_t` spectx, unsigned int ls, void \*ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)
- int `_base_spe_mfcio_getf` (`spe_context_ptr_t` spectx, unsigned int ls, void \*ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)
- int `_base_spe_out_mbox_read` (`spe_context_ptr_t` spectx, unsigned int mbox\_data[ ], int count)
- int `_base_spe_in_mbox_write` (`spe_context_ptr_t` spectx, unsigned int mbox\_data[ ], int count, int behavior\_flag)
- int `_base_spe_in_mbox_status` (`spe_context_ptr_t` spectx)
- int `_base_spe_out_mbox_status` (`spe_context_ptr_t` spectx)
- int `_base_spe_out_intr_mbox_status` (`spe_context_ptr_t` spectx)
- int `_base_spe_out_intr_mbox_read` (`spe_context_ptr_t` spectx, unsigned int mbox\_data[ ], int count, int behavior\_flag)
- int `_base_spe_signal_write` (`spe_context_ptr_t` spectx, unsigned int signal\_reg, unsigned int data)
- int `_base_spe_callback_handler_register` (void \*handler, unsigned int callnum, unsigned int mode)
- int `_base_spe_callback_handler_deregister` (unsigned int callnum)
- void \* `_base_spe_callback_handler_query` (unsigned int callnum)
- int `_base_spe_stop_reason_get` (`spe_context_ptr_t` spectx)
- int `_base_spe_mfcio_tag_status_read` (`spe_context_ptr_t` spectx, unsigned int mask, unsigned int behavior, unsigned int \*tag\_status)
- int `_base_spe_stop_event_source_get` (`spe_context_ptr_t` spectx)

- `int __base_spe_stop_event_target_get (spe_context_ptr_t spectx)`
- `int __base_spe_stop_status_get (spe_context_ptr_t spectx)`
- `int __base_spe_event_source_acquire (struct spe_context *spectx, enum fd_name fdesc)`
- `void __base_spe_event_source_release (struct spe_context *spectx, enum fd_name fdesc)`
- `void * __base_spe_ps_area_get (struct spe_context *spectx, enum ps_area area)`
- `int __base_spe_spe_dir_get (struct spe_context *spectx)`
- `void * __base_spe_ls_area_get (struct spe_context *spectx)`
- `int __base_spe_ls_size_get (spe_context_ptr_t spe)`
- `void __base_spe_context_lock (spe_context_ptr_t spe, enum fd_name fd)`
- `void __base_spe_context_unlock (spe_context_ptr_t spe, enum fd_name fd)`
- `int __base_spe_cpu_info_get (int info_requested, int cpu_node)`
- `void __spe_context_update_event (void)`
- `int __base_spe_mssync_start (spe_context_ptr_t spectx)`
- `int __base_spe_mssync_status (spe_context_ptr_t spectx)`

### 3.23.1 Detailed Description

[spebase.h](#) contains the public API funtions

Definition in file [spebase.h](#).

### 3.23.2 Define Documentation

#### 3.23.2.1 `#define __PRINTF(fmt, args...) { fprintf(stderr,fmt , ## args); }`

Definition at line 34 of file [spebase.h](#).

#### 3.23.2.2 `#define CNTL_OFFSET 0x04000`

Definition at line 124 of file [spebase.h](#).

Referenced by [\\_base\\_spe\\_context\\_create\(\)](#).

#### 3.23.2.3 `#define CNTL_SIZE 0x1000`

Definition at line 119 of file [spebase.h](#).

Referenced by [\\_base\\_spe\\_context\\_create\(\)](#).

#### 3.23.2.4 `#define DEBUG_PRINTF(fmt, args...)`

Definition at line 38 of file [spebase.h](#).

#### 3.23.2.5 `#define LS_SIZE 0x40000`

Definition at line 115 of file [spebase.h](#).

**3.23.2.6 #define MFC\_OFFSET 0x03000**

Definition at line 123 of file spebase.h.

Referenced by `_base_spe_context_create()`.

**3.23.2.7 #define MFC\_SIZE 0x1000**

Definition at line 117 of file spebase.h.

Referenced by `_base_spe_context_create()`.

**3.23.2.8 #define MSS\_SIZE 0x1000**

Definition at line 118 of file spebase.h.

Referenced by `_base_spe_context_create()`.

**3.23.2.9 #define MSSYNC\_OFFSET 0x00000**

Definition at line 122 of file spebase.h.

Referenced by `_base_spe_context_create()`.

**3.23.2.10 #define PSMAP\_SIZE 0x20000**

Definition at line 116 of file spebase.h.

Referenced by `_base_spe_context_create()`.

**3.23.2.11 #define SIGNAL1\_OFFSET 0x14000**

Definition at line 125 of file spebase.h.

Referenced by `_base_spe_context_create()`.

**3.23.2.12 #define SIGNAL2\_OFFSET 0x1c000**

Definition at line 126 of file spebase.h.

Referenced by `_base_spe_context_create()`.

**3.23.2.13 #define SIGNAL\_SIZE 0x1000**

Definition at line 120 of file spebase.h.

Referenced by `_base_spe_context_create()`.

**3.23.2.14 #define SPE\_EMULATE\_PARAM\_BUFFER 0x3e000**

Location of the PPE-assisted library call buffer for emulated isolation contexts.

Definition at line 132 of file spebase.h.

Referenced by `_base_spe_handle_library_callback()`.

#### 3.23.2.15 **#define SPE\_PROGRAM\_ISO\_LOAD\_COMPLETE 0x2206**

Definition at line 143 of file `spebase.h`.

Referenced by `_base_spe_context_run()`.

#### 3.23.2.16 **#define SPE\_PROGRAM\_ISOLATED\_STOP 0x2200**

Isolated exit codes: 0x220x

Definition at line 142 of file `spebase.h`.

Referenced by `_base_spe_context_run()`.

#### 3.23.2.17 **#define SPE\_PROGRAM\_LIBRARY\_CALL 0x2100**

Definition at line 137 of file `spebase.h`.

Referenced by `_base_spe_context_run()`.

#### 3.23.2.18 **#define SPE\_PROGRAM\_NORMAL\_END 0x2000**

Definition at line 136 of file `spebase.h`.

Referenced by `_base_spe_context_run()`.

### 3.23.3 Enumeration Type Documentation

#### 3.23.3.1 **enum fd\_name**

NOTE: `NUM_MBOX_FDS` must always be the last element in the enumeration

Enumerator:

***FD\_MBOX***

***FD\_MBOX\_STAT***

***FD\_IBOX***

***FD\_IBOX\_NB***

***FD\_IBOX\_STAT***

***FD\_WBOX***

***FD\_WBOX\_NB***

***FD\_WBOX\_STAT***

***FD\_SIG1***

***FD\_SIG2***

***FD\_MFC***

***FD\_MSS***

***NUM\_MBOX\_FDS***



Definition at line 42 of file spebase.h.

```

42         {
43         FD_MBOX,
44         FD_MBOX_STAT,
45         FD_IBOX,
46         FD_IBOX_NB,
47         FD_IBOX_STAT,
48         FD_WBOX,
49         FD_WBOX_NB,
50         FD_WBOX_STAT,
51         FD_SIG1,
52         FD_SIG2,
53         FD_MFC,
54         FD_MSS,
55         NUM_MBOX_FDS
56 };

```

### 3.23.4 Function Documentation

#### 3.23.4.1 int \_\_base\_spe\_event\_source\_acquire (struct spe\_context \* *spectx*, enum fd\_name *fdesc*)

\_\_base\_spe\_event\_source\_acquire opens a file descriptor to the specified event source

##### Parameters:

*spectx* Specifies the SPE context

*fdesc* Specifies the event source

#### 3.23.4.2 void \_\_base\_spe\_event\_source\_release (struct spe\_context \* *spectx*, enum fd\_name *fdesc*)

\_\_base\_spe\_event\_source\_release releases the file descriptor to the specified event source

##### Parameters:

*spectx* Specifies the SPE context

*fdesc* Specifies the event source

Definition at line 79 of file accessors.c.

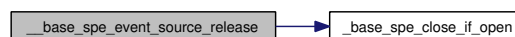
References `_base_spe_close_if_open()`.

```

80 {
81     _base_spe_close_if_open(spe, fdesc);
82 }

```

Here is the call graph for this function:



**3.23.4.3 int \_\_base\_spe\_spe\_dir\_get (struct spe\_context \* *spectx*)**

\_\_base\_spe\_spe\_dir\_get return the file descriptor of the SPE directory in spufs

**Parameters:**

*spectx* Specifies the SPE context

**3.23.4.4 int \_\_base\_spe\_stop\_event\_source\_get (spe\_context\_ptr\_t *spe*)**

\_\_base\_spe\_stop\_event\_source\_get

**Parameters:**

*spectx* Specifies the SPE context

speevent users read from this end

Definition at line 92 of file accessors.c.

References spe\_context::base\_private, and spe\_context\_base\_priv::ev\_pipe.

```

93 {
94     return spe->base_private->ev_pipe[1];
95 }
```

**3.23.4.5 int \_\_base\_spe\_stop\_event\_target\_get (spe\_context\_ptr\_t *spe*)**

\_\_base\_spe\_stop\_event\_target\_get

**Parameters:**

*spectx* Specifies the SPE context

speevent writes to this end

Definition at line 100 of file accessors.c.

References spe\_context::base\_private, and spe\_context\_base\_priv::ev\_pipe.

```

101 {
102     return spe->base_private->ev_pipe[0];
103 }
```

**3.23.4.6 void \_\_spe\_context\_update\_event (void)**

\_\_spe\_context\_update\_event internal function for gdb notification.

Referenced by \_base\_spe\_context\_destroy(), and \_base\_spe\_program\_load\_complete().

**3.23.4.7 int \_base\_spe\_callback\_handler\_deregister (unsigned int *callnum*)**

unregister a handler function for the specified number NOTE: unregistering a handler from call zero and one is ignored.

Definition at line 78 of file lib\_builtin.c.

References MAX\_CALLNUM, and RESERVED.

```

79 {
80     errno = 0;
81     if (callnum > MAX_CALLNUM) {
82         errno = EINVAL;
83         return -1;
84     }
85     if (callnum < RESERVED) {
86         errno = EACCES;
87         return -1;
88     }
89     if (handlers[callnum] == NULL) {
90         errno = ESRCH;
91         return -1;
92     }
93
94     handlers[callnum] = NULL;
95     return 0;
96 }
```

**3.23.4.8 void\* \_base\_spe\_callback\_handler\_query (unsigned int *callnum*)**

query a handler function for the specified number

Definition at line 98 of file lib\_builtin.c.

References MAX\_CALLNUM.

```

99 {
100     errno = 0;
101
102     if (callnum > MAX_CALLNUM) {
103         errno = EINVAL;
104         return NULL;
105     }
106     if (handlers[callnum] == NULL) {
107         errno = ESRCH;
108         return NULL;
109     }
110     return handlers[callnum];
111 }
```

**3.23.4.9 int \_base\_spe\_callback\_handler\_register (void \* *handler*, unsigned int *callnum*, unsigned int *mode*)**

register a handler function for the specified number NOTE: registering a handler to call zero and one is ignored.

Definition at line 40 of file lib\_builtin.c.

References MAX\_CALLNUM, RESERVED, SPE\_CALLBACK\_NEW, and SPE\_CALLBACK\_UPDATE.

```

41 {
42     errno = 0;
43
44     if (callnum > MAX_CALLNUM) {
45         errno = EINVAL;
46         return -1;
47     }
48
49     switch(mode){
50     case SPE_CALLBACK_NEW:
51         if (callnum < RESERVED) {
52             errno = EACCES;
53             return -1;
54         }
55         if (handlers[callnum] != NULL) {
56             errno = EACCES;
57             return -1;
58         }
59         handlers[callnum] = handler;
60         break;
61
62     case SPE_CALLBACK_UPDATE:
63         if (handlers[callnum] == NULL) {
64             errno = ESRCH;
65             return -1;
66         }
67         handlers[callnum] = handler;
68         break;
69     default:
70         errno = EINVAL;
71         return -1;
72         break;
73     }
74     return 0;
75
76 }

```

#### 3.23.4.10 `spe_context_ptr_t _base_spe_context_create (unsigned int flags, spe_gang_context_ptr_t gctx, spe_context_ptr_t aff_spe)`

`_base_spe_context_create` creates a single SPE context, i.e., the corresponding directory is created in SPUFS either as a subdirectory of a gang or individually (maybe this is best considered a gang of one)

##### Parameters:

*flags*

*gctx* specify NULL if not belonging to a gang

*aff\_spe* specify NULL to skip affinity information

Definition at line 183 of file `create.c`.

References `_base_spe_emulated_loader_present()`, `spe_gang_context::base_private`, `spe_context::base_private`, `spe_context_base_priv::cntl_mmap_base`, `CNTL_OFFSET`, `CNTL_SIZE`, `DEBUG_PRINTF`, `spe_context_base_priv::fd_lock`, `spe_context_base_priv::fd_spe_dir`, `spe_context_base_priv::flags`, `spe_gang_context_base_priv::gangname`, `spe_context_base_priv::loaded_program`, `LS_SIZE`, `spe_context_base_priv::mem_mmap_base`, `spe_context_base_priv::mfc_mmap_base`, `MFC_OFFSET`, `MFC_SIZE`, `MSS_SIZE`, `spe_context_base_priv::mssync_mmap_base`, `MSSYNC_OFFSET`, `spe_context_base_priv::psmap_mmap_base`, `PSMAP_SIZE`, `spe_context_base_priv::signal1_mmap_base`, `SIGNAL1_OFFSET`, `spe_context_base_priv::signal2_mmap_base`, `SIGNAL2_OFFSET`, `SIGNAL_SIZE`, `SPE_AFFINITY_MEMORY`, `SPE_CFG_SIGNOTIFY1_OR`, `SPE_CFG_SIGNOTIFY2_OR`,

SPE\_EVENTS\_ENABLE, spe\_context\_base\_priv::spe\_fds\_array, SPE\_ISOLATE, SPE\_ISOLATE\_EMULATE, and SPE\_MAP\_PS.

```

185 {
186     char pathname[256];
187     int i, aff_spe_fd = 0;
188     unsigned int spu_createflags = 0;
189     struct spe_context *spe = NULL;
190     struct spe_context_base_priv *priv;
191
192     /* We need a loader present to run in emulated isolated mode */
193     if (flags & SPE_ISOLATE_EMULATE
194         && !_base_spe_emulated_loader_present()) {
195         errno = EINVAL;
196         return NULL;
197     }
198
199     /* Put some sane defaults into the SPE context */
200     spe = malloc(sizeof(*spe));
201     if (!spe) {
202         DEBUG_PRINTF("ERROR: Could not allocate spe context.\n");
203         return NULL;
204     }
205     memset(spe, 0, sizeof(*spe));
206
207     spe->base_private = malloc(sizeof(*spe->base_private));
208     if (!spe->base_private) {
209         DEBUG_PRINTF("ERROR: Could not allocate "
210                     "spe->base_private context.\n");
211         free(spe);
212         return NULL;
213     }
214
215     /* just a convenience variable */
216     priv = spe->base_private;
217
218     priv->fd_spe_dir = -1;
219     priv->mem_mmap_base = MAP_FAILED;
220     priv->psmap_mmap_base = MAP_FAILED;
221     priv->mssync_mmap_base = MAP_FAILED;
222     priv->mfc_mmap_base = MAP_FAILED;
223     priv->cntl_mmap_base = MAP_FAILED;
224     priv->signal1_mmap_base = MAP_FAILED;
225     priv->signal2_mmap_base = MAP_FAILED;
226     priv->loaded_program = NULL;
227
228     for (i = 0; i < NUM_MBOX_FDS; i++) {
229         priv->spe_fds_array[i] = -1;
230         pthread_mutex_init(&priv->fd_lock[i], NULL);
231     }
232
233     /* initialise spu_createflags */
234     if (flags & SPE_ISOLATE) {
235         flags |= SPE_MAP_PS;
236         spu_createflags |= SPU_CREATE_ISOLATE | SPU_CREATE_NOSCHED;
237     }
238
239     if (flags & SPE_EVENTS_ENABLE)
240         spu_createflags |= SPU_CREATE_EVENTS_ENABLED;
241
242     if (aff_spe)
243         spu_createflags |= SPU_CREATE_AFFINITY_SPU;
244
245     if (flags & SPE_AFFINITY_MEMORY)
246         spu_createflags |= SPU_CREATE_AFFINITY_MEM;
247
248     /* Make the SPUFS directory for the SPE */

```

```

249     if (gctx == NULL)
250         sprintf(pathname, "/spu/spethread-%i-%lu",
251                 getpid(), (unsigned long)spe);
252     else
253         sprintf(pathname, "/spu/%s/spethread-%i-%lu",
254                 gctx->base_private->gangname, getpid(),
255                 (unsigned long)spe);
256
257     if (aff_spe)
258         aff_spe_fd = aff_spe->base_private->fd_spe_dir;
259
260     priv->fd_spe_dir = spu_create(pathname, spu_createflags,
261                                 S_IRUSR | S_IWUSR | S_IXUSR, aff_spe_fd);
262
263     if (priv->fd_spe_dir < 0) {
264         DEBUG_PRINTF("ERROR: Could not create SPE %s\n", pathname);
265         perror("spu_create()");
266         free_spe_context(spe);
267         /* we mask most errors, but leave ENODEV */
268         if (errno != ENODEV)
269             errno = EFAULT;
270         return NULL;
271     }
272
273     priv->flags = flags;
274
275     /* Map the required areas into process memory */
276     priv->mem_mmap_base = mapfileat(priv->fd_spe_dir, "mem", LS_SIZE);
277     if (priv->mem_mmap_base == MAP_FAILED) {
278         DEBUG_PRINTF("ERROR: Could not map SPE memory.\n");
279         free_spe_context(spe);
280         errno = ENOMEM;
281         return NULL;
282     }
283
284     if (flags & SPE_MAP_PS) {
285         /* It's possible to map the entire problem state area with
286          * one mmap - try this first */
287         priv->psmap_mmap_base = mapfileat(priv->fd_spe_dir,
288                                         "psmap", PSMAP_SIZE);
289
290         if (priv->psmap_mmap_base != MAP_FAILED) {
291             priv->mssync_mmap_base =
292                 priv->psmap_mmap_base + MSSYNC_OFFSET;
293             priv->mfc_mmap_base =
294                 priv->psmap_mmap_base + MFC_OFFSET;
295             priv->cntl_mmap_base =
296                 priv->psmap_mmap_base + CNTL_OFFSET;
297             priv->signal1_mmap_base =
298                 priv->psmap_mmap_base + SIGNAL1_OFFSET;
299             priv->signal2_mmap_base =
300                 priv->psmap_mmap_base + SIGNAL2_OFFSET;
301
302         } else {
303             /* map each region separately */
304             priv->mfc_mmap_base =
305                 mapfileat(priv->fd_spe_dir, "mfc", MFC_SIZE);
306             priv->mssync_mmap_base =
307                 mapfileat(priv->fd_spe_dir, "mss", MSS_SIZE);
308             priv->cntl_mmap_base =
309                 mapfileat(priv->fd_spe_dir, "cntl", CNTL_SIZE);
310             priv->signal1_mmap_base =
311                 mapfileat(priv->fd_spe_dir, "signal1",
312                           SIGNAL_SIZE);
313             priv->signal2_mmap_base =
314                 mapfileat(priv->fd_spe_dir, "signal2",
315                           SIGNAL_SIZE);

```

```

316
317         if (priv->mfc_mmap_base == MAP_FAILED ||
318             priv->cntl_mmap_base == MAP_FAILED ||
319             priv->signal1_mmap_base == MAP_FAILED ||
320             priv->signal2_mmap_base == MAP_FAILED) {
321             DEBUG_PRINTF("ERROR: Could not map SPE "
322                          "PS memory.\n");
323             free_spe_context(spe);
324             errno = ENOMEM;
325             return NULL;
326         }
327     }
328 }
329
330 if (flags & SPE_CFG_SIGNOTIFY1_OR) {
331     if (setsignotify(priv->fd_spe_dir, "signal1_type")) {
332         DEBUG_PRINTF("ERROR: Could not open SPE "
333                     "signal1_type file.\n");
334         free_spe_context(spe);
335         errno = EFAULT;
336         return NULL;
337     }
338 }
339
340 if (flags & SPE_CFG_SIGNOTIFY2_OR) {
341     if (setsignotify(priv->fd_spe_dir, "signal2_type")) {
342         DEBUG_PRINTF("ERROR: Could not open SPE "
343                     "signal2_type file.\n");
344         free_spe_context(spe);
345         errno = EFAULT;
346         return NULL;
347     }
348 }
349
350 return spe;
351 }

```

Here is the call graph for this function:



#### 3.23.4.11 int \_base\_spe\_context\_destroy (spe\_context\_ptr\_t *spectx*)

\_base\_spe\_context\_destroy cleans up what is left when an SPE executable has exited. Closes open file handles and unmaps memory areas.

##### Parameters:

*spectx* Specifies the SPE context

Definition at line 406 of file create.c.

References \_\_spe\_context\_update\_event().

```

407 {
408     int ret = free_spe_context(spe);
409
410     __spe_context_update_event();
411
412     return ret;
413 }

```

Here is the call graph for this function:



### 3.23.4.12 void \_base\_spe\_context\_lock (spe\_context\_ptr\_t *spe*, enum fd\_name *fd*)

\_base\_spe\_context\_lock locks members of the SPE context

#### Parameters:

*spectx* Specifies the SPE context

*fd* Specifies the file

Definition at line 91 of file create.c.

References spe\_context::base\_private, and spe\_context\_base\_priv::fd\_lock.

```

92 {
93     pthread_mutex_lock (&spe->base_private->fd_lock[fdesc]);
94 }
```

### 3.23.4.13 int \_base\_spe\_context\_run (spe\_context\_ptr\_t *spe*, unsigned int \* *entry*, unsigned int *runflags*, void \* *argp*, void \* *envp*, spe\_stop\_info\_t \* *stopinfo*)

\_base\_spe\_context\_run starts execution of an SPE context with a loaded image

#### Parameters:

*spectx* Specifies the SPE context

*entry* entry point for the SPE program. If set to 0, entry point is determined by the ELF loader.

*runflags* valid values are:

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 do not use built in library functions.

*argp* An (optional) pointer to application specific data, and is passed as the second parameter to the SPE program.

*envp* An (optional) pointer to environment specific data, and is passed as the third parameter to the SPE program.

Definition at line 98 of file run.c.

References \_\_spe\_current\_active\_context, \_base\_spe\_handle\_library\_callback(), \_base\_spe\_program\_load\_complete(), spe\_context::base\_private, DEBUG\_PRINTF, spe\_context\_base\_priv::emulated\_entry, spe\_context\_base\_priv::entry, spe\_context\_base\_priv::fd\_spe\_dir, spe\_context\_base\_priv::flags, LS\_SIZE, spe\_context\_base\_priv::mem\_mmap\_base, spe\_context\_info::npc, spe\_context\_info::prev, spe\_stop\_info\_t::result, spe\_stop\_info\_t::spe\_callback\_error, SPE\_CALLBACK\_ERROR, SPE\_DEFAULT\_ENTRY, SPE\_EVENTS\_ENABLE, SPE\_EXIT, spe\_stop\_info\_t::spe\_exit\_code, spe\_context\_info::spe\_id, SPE\_ISOLATE, SPE\_ISOLATE\_EMULATE, spe\_stop\_info\_t::spe\_isolation\_error, SPE\_ISOLATION\_ERROR, SPE\_NO\_CALLBACKS, SPE\_PROGRAM\_ISO\_LOAD\_COMPLETE,



SPE\_PROGRAM\_ISOLATED\_STOP, SPE\_PROGRAM\_LIBRARY\_CALL, SPE\_PROGRAM\_NORMAL\_END, SPE\_RUN\_USER\_REGS, spe\_stop\_info\_t::spe\_runtime\_error, SPE\_RUNTIME\_ERROR, spe\_stop\_info\_t::spe\_runtime\_exception, SPE\_RUNTIME\_EXCEPTION, spe\_stop\_info\_t::spe\_runtime\_fatal, SPE\_RUNTIME\_FATAL, spe\_stop\_info\_t::spe\_signal\_code, SPE\_SPU\_HALT, SPE\_SPU\_INVALID\_CHANNEL, SPE\_SPU\_INVALID\_INSTR, SPE\_SPU\_STOPPED\_BY\_STOP, SPE\_SPU\_WAITING\_ON\_CHANNEL, SPE\_STOP\_AND\_SIGNAL, spe\_stop\_info\_t::spu\_status, spe\_context\_info::status, spe\_stop\_info\_t::stop\_reason, addr64::ui, and addr64::ull.

```

101 {
102     int retval = 0, run_rc;
103     unsigned int run_status, tmp_entry;
104     spe_stop_info_t stopinfo_buf;
105     struct spe_context_info this_context_info __attribute__((cleanup(cleanupspeinfo)));
106
107     /* If the caller hasn't set a stopinfo buffer, provide a buffer on the
108      * stack instead. */
109     if (!stopinfo)
110         stopinfo = &stopinfo_buf;
111
112
113     /* In emulated isolated mode, the npc will always return as zero.
114      * use our private entry point instead */
115     if (spe->base_private->flags & SPE_ISOLATE_EMULATE)
116         tmp_entry = spe->base_private->emulated_entry;
117
118     else if (*entry == SPE_DEFAULT_ENTRY)
119         tmp_entry = spe->base_private->entry;
120     else
121         tmp_entry = *entry;
122
123     /* If we're starting the SPE binary from its original entry point,
124      * setup the arguments to main() */
125     if (tmp_entry == spe->base_private->entry &&
126         !(spe->base_private->flags &
127           (SPE_ISOLATE | SPE_ISOLATE_EMULATE))) {
128
129         addr64 argp64, envp64, tid64, ls64;
130         unsigned int regs[128][4];
131
132         /* setup parameters */
133         argp64.ull = (uint64_t)(unsigned long)argp;
134         envp64.ull = (uint64_t)(unsigned long)envp;
135         tid64.ull = (uint64_t)(unsigned long)spe;
136
137         /* make sure the register values are 0 */
138         memset(regs, 0, sizeof(regs));
139
140         /* set sensible values for stack_ptr and stack_size */
141         regs[1][0] = (unsigned int) LS_SIZE - 16;          /* stack_ptr */
142         regs[2][0] = 0;                                     /* stack_size
143
144         if (runflags & SPE_RUN_USER_REGS) {
145             /* When SPE_USER_REGS is set, argp points to an array
146              * of 3x128b registers to be passed directly to the SPE
147              * program.
148              */
149             memcpy(regs[3], argp, sizeof(unsigned int) * 12);
150         } else {
151             regs[3][0] = tid64.ui[0];
152             regs[3][1] = tid64.ui[1];
153
154             regs[4][0] = argp64.ui[0];
155             regs[4][1] = argp64.ui[1];
156
157             regs[5][0] = envp64.ui[0];
158             regs[5][1] = envp64.ui[1];

```

```

159         }
160
161         /* Store the LS base address in R6 */
162         ls64.u1l = (uint64_t)(unsigned long)spe->base_private->mem_mmap_base;
163         regs[6][0] = ls64.ui[0];
164         regs[6][1] = ls64.ui[1];
165
166         if (set_regs(spe, regs))
167             return -1;
168     }
169
170     /*Leave a trail of breadcrumbs for the debugger to follow */
171     if (!__spe_current_active_context) {
172         __spe_current_active_context = &this_context_info;
173         if (!__spe_current_active_context)
174             return -1;
175         __spe_current_active_context->prev = NULL;
176     } else {
177         struct spe_context_info *newinfo;
178         newinfo = &this_context_info;
179         if (!newinfo)
180             return -1;
181         newinfo->prev = __spe_current_active_context;
182         __spe_current_active_context = newinfo;
183     }
184     /*remember the ls-addr*/
185     __spe_current_active_context->spe_id = spe->base_private->fd_spe_dir;
186
187 do_run:
188     /*Remember the npc value*/
189     __spe_current_active_context->npc = tmp_entry;
190
191     /* run SPE context */
192     run_rc = spu_run(spe->base_private->fd_spe_dir,
193                     &tmp_entry, &run_status);
194
195     /*Remember the npc value*/
196     __spe_current_active_context->npc = tmp_entry;
197     __spe_current_active_context->status = run_status;
198
199     DEBUG_PRINTF("spu_run returned run_rc=0x%08x, entry=0x%04x, "
200                 "ext_status=0x%04x.\n", run_rc, tmp_entry, run_status);
201
202     /* set up return values and stopinfo according to spu_run exit
203      * conditions. This is overwritten on error.
204      */
205     stopinfo->spu_status = run_rc;
206
207     if (spe->base_private->flags & SPE_ISOLATE_EMULATE) {
208         /* save the entry point, and pretend that the npc is zero */
209         spe->base_private->emulated_entry = tmp_entry;
210         *entry = 0;
211     } else {
212         *entry = tmp_entry;
213     }
214
215     /* Return with stopinfo set on syscall error paths */
216     if (run_rc == -1) {
217         DEBUG_PRINTF("spu_run returned error %d, errno=%d\n",
218                     run_rc, errno);
219         stopinfo->stop_reason = SPE_RUNTIME_FATAL;
220         stopinfo->result.spe_runtime_fatal = errno;
221         retval = -1;
222
223         /* For isolated contexts, pass EPERM up to the
224          * caller.
225          */

```

```

226         if (!(spe->base_private->flags & SPE_ISOLATE
227             && errno == EPERM))
228             errno = EFAULT;
229
230     } else if (run_rc & SPE_SPU_INVALID_INSTR) {
231         DEBUG_PRINTF("SPU has tried to execute an invalid "
232             "instruction. %d\n", run_rc);
233         stopinfo->stop_reason = SPE_RUNTIME_ERROR;
234         stopinfo->result.spe_runtime_error = SPE_SPU_INVALID_INSTR;
235         errno = EFAULT;
236         retval = -1;
237
238     } else if ((spe->base_private->flags & SPE_EVENTS_ENABLE) && run_status) {
239         /* Report asynchronous error if return val are set and
240          * SPU events are enabled.
241          */
242         stopinfo->stop_reason = SPE_RUNTIME_EXCEPTION;
243         stopinfo->result.spe_runtime_exception = run_status;
244         stopinfo->spu_status = -1;
245         errno = EIO;
246         retval = -1;
247
248     } else if (run_rc & SPE_SPU_STOPPED_BY_STOP) {
249         /* Stop & signals are broken down into three groups
250          * 1. SPE library call
251          * 2. SPE user defined stop & signal
252          * 3. SPE program end.
253          *
254          * These groups are signified by the 14-bit stop code:
255          */
256         int stopcode = (run_rc >> 16) & 0x3fff;
257
258         /* Check if this is a library callback, and callbacks are
259          * allowed (ie, running without SPE_NO_CALLBACKS)
260          */
261         if ((stopcode & 0xff00) == SPE_PROGRAM_LIBRARY_CALL
262             && !(runflags & SPE_NO_CALLBACKS)) {
263
264             int callback_rc, callback_number = stopcode & 0xff;
265
266             /* execute library callback */
267             DEBUG_PRINTF("SPE library call: %d\n", callback_number);
268             callback_rc = _base_spe_handle_library_callback(spe,
269                 callback_number, *entry);
270
271             if (callback_rc) {
272                 /* library callback failed; set errno and
273                  * return immediately */
274                 DEBUG_PRINTF("SPE library call failed: %d\n",
275                     callback_rc);
276                 stopinfo->stop_reason = SPE_CALLBACK_ERROR;
277                 stopinfo->result.spe_callback_error =
278                     callback_rc;
279                 errno = EFAULT;
280                 retval = -1;
281             } else {
282                 /* successful library callback - restart the SPE
283                  * program at the next instruction */
284                 tmp_entry += 4;
285                 goto do_run;
286             }
287
288         } else if ((stopcode & 0xff00) == SPE_PROGRAM_NORMAL_END) {
289             /* The SPE program has exited by exit(X) */
290             stopinfo->stop_reason = SPE_EXIT;
291             stopinfo->result.spe_exit_code = stopcode & 0xff;
292

```

```

293         if (spe->base_private->flags & SPE_ISOLATE) {
294             /* Issue an isolated exit, and re-run the SPE.
295              * We should see a return value without the
296              * 0x80 bit set. */
297             if (!issue_isolated_exit(spe))
298                 goto do_run;
299             retval = -1;
300         }
301
302     } else if ((stopcode & 0xfff0) == SPE_PROGRAM_ISOLATED_STOP) {
303
304         /* 0x2206: isolated app has been loaded by loader;
305          * provide a hook for the debugger to catch this,
306          * and restart
307          */
308         if (stopcode == SPE_PROGRAM_ISO_LOAD_COMPLETE) {
309             _base_spe_program_load_complete(spe);
310             goto do_run;
311         } else {
312             stopinfo->stop_reason = SPE_ISOLATION_ERROR;
313             stopinfo->result.spe_isolation_error =
314                 stopcode & 0xf;
315         }
316
317     } else if (spe->base_private->flags & SPE_ISOLATE &&
318               !(run_rc & 0x80)) {
319         /* We've successfully exited isolated mode */
320         retval = 0;
321
322     } else {
323         /* User defined stop & signal, including
324          * callbacks when disabled */
325         stopinfo->stop_reason = SPE_STOP_AND_SIGNAL;
326         stopinfo->result.spe_signal_code = stopcode;
327         retval = stopcode;
328     }
329
330 } else if (run_rc & SPE_SPU_HALT) {
331     DEBUG_PRINTF("SPU was stopped by halt. %d\n", run_rc);
332     stopinfo->stop_reason = SPE_RUNTIME_ERROR;
333     stopinfo->result.spe_runtime_error = SPE_SPU_HALT;
334     errno = EFAULT;
335     retval = -1;
336
337 } else if (run_rc & SPE_SPU_WAITING_ON_CHANNEL) {
338     DEBUG_PRINTF("SPU is waiting on channel. %d\n", run_rc);
339     stopinfo->stop_reason = SPE_RUNTIME_EXCEPTION;
340     stopinfo->result.spe_runtime_exception = run_status;
341     stopinfo->spu_status = -1;
342     errno = EIO;
343     retval = -1;
344
345 } else if (run_rc & SPE_SPU_INVALID_CHANNEL) {
346     DEBUG_PRINTF("SPU has tried to access an invalid "
347                 "channel. %d\n", run_rc);
348     stopinfo->stop_reason = SPE_RUNTIME_ERROR;
349     stopinfo->result.spe_runtime_error = SPE_SPU_INVALID_CHANNEL;
350     errno = EFAULT;
351     retval = -1;
352
353 } else {
354     DEBUG_PRINTF("spu_run returned invalid data: 0x%04x\n", run_rc);
355     stopinfo->stop_reason = SPE_RUNTIME_FATAL;
356     stopinfo->result.spe_runtime_fatal = -1;
357     stopinfo->spu_status = -1;
358     errno = EFAULT;
359     retval = -1;

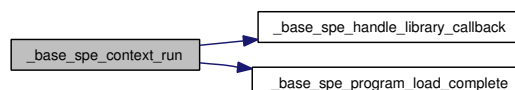
```

```

360
361     }
362
363     freespeinfo();
364     return retval;
365 }

```

Here is the call graph for this function:



#### 3.23.4.14 void \_base\_spe\_context\_unlock (spe\_context\_ptr\_t *spe*, enum fd\_name *fd*)

\_base\_spe\_context\_unlock unlocks members of the SPE context

##### Parameters:

*spectx* Specifies the SPE context

*fd* Specifies the file

Definition at line 96 of file create.c.

References spe\_context::base\_private, and spe\_context\_base\_priv::fd\_lock.

```

97 {
98     pthread_mutex_unlock(&spe->base_private->fd_lock[fdesc]);
99 }

```

#### 3.23.4.15 int \_base\_spe\_cpu\_info\_get (int *info\_requested*, int *cpu\_node*)

\_base\_spe\_info\_get

Definition at line 105 of file info.c.

References \_base\_spe\_count\_physical\_cpus(), \_base\_spe\_count\_physical\_spes(), \_base\_spe\_count\_usable\_spes(), SPE\_COUNT\_PHYSICAL\_CPU\_NODES, SPE\_COUNT\_PHYSICAL\_SPES, and SPE\_COUNT\_USABLE\_SPES.

```

105                                     {
106     int ret = 0;
107     errno = 0;
108
109     switch (info_requested) {
110     case SPE_COUNT_PHYSICAL_CPU_NODES:
111         ret = _base_spe_count_physical_cpus(cpu_node);
112         break;
113     case SPE_COUNT_PHYSICAL_SPES:
114         ret = _base_spe_count_physical_spes(cpu_node);
115         break;
116     case SPE_COUNT_USABLE_SPES:
117         ret = _base_spe_count_usable_spes(cpu_node);
118         break;
119     default:

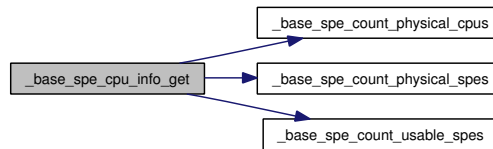
```

```

120             errno = EINVAL;
121             ret = -1;
122         }
123         return ret;
124     }

```

Here is the call graph for this function:



### 3.23.4.16 int \_base\_spe\_emulated\_loader\_present (void)

Check if the emulated loader is present in the filesystem

#### Returns:

Non-zero if the loader is available, otherwise zero.

Definition at line 145 of file load.c.

References `_base_spe_verify_spe_elf_image()`.

```

146 {
147     spe_program_handle_t *loader = emulated_loader_program();
148
149     if (!loader)
150         return 0;
151
152     return !_base_spe_verify_spe_elf_image(loader);
153 }

```

Here is the call graph for this function:



### 3.23.4.17 spe\_gang\_context\_ptr\_t \_base\_spe\_gang\_context\_create (unsigned int flags)

creates the directory in SPUFS that will contain all SPEs that are considered a gang Note: I would like to generalize this to a "group" or "set" Additional attributes maintained at the group level should be used to define scheduling constraints such "temporal" (e.g., scheduled all at the same time, i.e., a gang) "topology" (e.g., "closeness" of SPEs for optimal communication)

Definition at line 364 of file create.c.

References `spe_gang_context::base_private`, `DEBUG_PRINTF`, and `spe_gang_context_base_priv::gangname`.

```

365 {

```

```

366     char pathname[256];
367     struct spe_gang_context_base_priv *pgctx = NULL;
368     struct spe_gang_context *gctx = NULL;
369
370     gctx = malloc(sizeof(*gctx));
371     if (!gctx) {
372         DEBUG_PRINTF("ERROR: Could not allocate spe context.\n");
373         return NULL;
374     }
375     memset(gctx, 0, sizeof(*gctx));
376
377     pgctx = malloc(sizeof(*pgctx));
378     if (!pgctx) {
379         DEBUG_PRINTF("ERROR: Could not allocate spe context.\n");
380         free(gctx);
381         return NULL;
382     }
383     memset(pgctx, 0, sizeof(*pgctx));
384
385     gctx->base_private = pgctx;
386
387     sprintf(gctx->base_private->gangname, "gang-%i-%lu", getpid(),
388             (unsigned long)gctx);
389     sprintf(pathname, "/spu/%s", gctx->base_private->gangname);
390
391     gctx->base_private->fd_gang_dir = spu_create(pathname, SPU_CREATE_GANG,
392         S_IRUSR | S_IWUSR | S_IXUSR);
393
394     if (gctx->base_private->fd_gang_dir < 0) {
395         DEBUG_PRINTF("ERROR: Could not create Gang %s\n", pathname);
396         free_spe_gang_context(gctx);
397         errno = EFAULT;
398         return NULL;
399     }
400
401     gctx->base_private->flags = flags;
402
403     return gctx;
404 }

```

#### 3.23.4.18 int \_base\_spe\_gang\_context\_destroy (spe\_gang\_context\_ptr\_t gctx)

\_base\_spe\_gang\_context\_destroy destroys a gang context and frees associated resources

##### Parameters:

*gctx* Specifies the SPE gang context

Definition at line 415 of file create.c.

```

416 {
417     return free_spe_gang_context(gctx);
418 }

```

#### 3.23.4.19 int \_base\_spe\_image\_close (spe\_program\_handle\_t \* handle)

\_base\_spe\_image\_close unmaps an SPE ELF object that was previously mapped using spe\_open\_image.

##### Parameters:

*handle* handle to open file

**Return values:**

**0** On success, `spe_close_image` returns 0.

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

Possible values for `errno`:

**EINVAL** From `spe_close_image`, this indicates that the file, specified by `filename`, was not previously mapped by a call to `spe_open_image`.

Definition at line 96 of file `image.c`.

References `spe_program_handle_t::elf_image`, `image_handle::map_size`, `image_handle::speh`, and `spe_program_handle_t::toe_shadow`.

```

97 {
98     int ret = 0;
99     struct image_handle *ih;
100
101     if (!handle) {
102         errno = EINVAL;
103         return -1;
104     }
105
106     ih = (struct image_handle *)handle;
107
108     if (!ih->speh.elf_image || !ih->map_size) {
109         errno = EINVAL;
110         return -1;
111     }
112
113     if (ih->speh.toe_shadow)
114         free(ih->speh.toe_shadow);
115
116     ret = munmap(ih->speh.elf_image, ih->map_size);
117     free(handle);
118
119     return ret;
120 }
```

#### 3.23.4.20 `spe_program_handle_t*_base_spe_image_open (const char *filename)`

`_base_spe_image_open` maps an SPE ELF executable indicated by `filename` into system memory and returns the mapped address appropriate for use by the `spe_create_thread` API. It is often more convenient/appropriate to use the loading methodologies where SPE ELF objects are converted to PPE static or shared libraries with symbols which point to the SPE ELF objects after these special libraries are loaded. These libraries are then linked with the associated PPE code to provide a direct symbol reference to the SPE ELF object. The symbols in this scheme are equivalent to the address returned from the `spe_open_image` function. SPE ELF objects loaded using this function are not shared with other processes, but SPE ELF objects loaded using the other scheme, mentioned above, can be shared if so desired.

**Parameters:**

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

**Returns:**

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 appropriately.

Possible values for `errno` include:

**EACCES** The calling process does not have permission to access the specified file.



**EFAULT** The filename parameter points to an address that was not contained in the calling process's address space.

A number of other errno values could be returned by the open(2), fstat(2), mmap(2), munmap(2), or close(2) system calls which may be utilized by the spe\_open\_image or spe\_close\_image functions.

**See also:**

spe\_create\_thread

Definition at line 37 of file image.c.

References `_base_spe_toe_ear()`, `_base_spe_verify_spe_elf_image()`, `spe_program_handle_t::elf_image`, `spe_program_handle_t::handle_size`, `image_handle::map_size`, `image_handle::speh`, and `spe_program_handle_t::toe_shadow`.

```

38 {
39     /* allocate an extra integer in the spe handle to keep the mapped size information */
40     struct image_handle *ret;
41     int binfd = -1, f_stat;
42     struct stat statbuf;
43     size_t ps = getpagesize ();
44
45     ret = malloc(sizeof(struct image_handle));
46     if (!ret)
47         return NULL;
48
49     ret->speh.handle_size = sizeof(spe_program_handle_t);
50     ret->speh.toe_shadow = NULL;
51
52     binfd = open(filename, O_RDONLY);
53     if (binfd < 0)
54         goto ret_err;
55
56     f_stat = fstat(binfd, &statbuf);
57     if (f_stat < 0)
58         goto ret_err;
59
60     /* Sanity: is it executable ?
61     */
62     if(!(statbuf.st_mode & (S_IXUSR | S_IXGRP | S_IXOTH))) {
63         errno=EACCES;
64         goto ret_err;
65     }
66
67     /* now store the size at the extra allocated space */
68     ret->map_size = (statbuf.st_size + ps - 1) & ~(ps - 1);
69
70     ret->speh.elf_image = mmap(NULL, ret->map_size,
71                               PROT_WRITE | PROT_READ,
72                               MAP_PRIVATE, binfd, 0);
73     if (ret->speh.elf_image == MAP_FAILED)
74         goto ret_err;
75
76     /*Verify that this is a valid SPE ELF object*/
77     if(!_base_spe_verify_spe_elf_image((spe_program_handle_t *)ret))
78         goto ret_err;
79
80     if (_base_spe_toe_ear(&ret->speh))
81         goto ret_err;
82
83     /* ok */
84     close(binfd);
85     return (spe_program_handle_t *)ret;
86

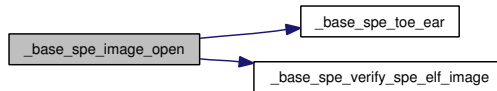
```

```

87      /* err & cleanup */
88 ret_err:
89      if (binfd >= 0)
90          close(binfd);
91
92      free(ret);
93      return NULL;
94 }

```

Here is the call graph for this function:



### 3.23.4.21 int \_base\_spe\_in\_mbox\_status (spe\_context\_ptr\_t spectx)

The `_base_spe_in_mbox_status` function fetches the status of the SPU inbound mailbox for the SPE thread specified by the `speid` 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:

*spectx* Specifies the SPE context whose mailbox status is to be read.

#### Returns:

On success, returns the current status of the mailbox, respectively. On failure, -1 is returned.

#### See also:

`spe_read_out_mbox`, `spe_write_in_mbox`, `read` (2)

Definition at line 202 of file `mbox.c`.

References `_base_spe_open_if_closed()`, `spe_context::base_private`, `spe_context_base_priv::cntl_mmap_base`, `FD_WBOX_STAT`, `spe_context_base_priv::flags`, and `SPE_MAP_PS`.

```

203 {
204     int rc, ret;
205     volatile struct spe_spu_control_area *cntl_area =
206         spectx->base_private->cntl_mmap_base;
207
208     if (spectx->base_private->flags & SPE_MAP_PS) {
209         ret = (cntl_area->SPU_Mbox_Stat >> 8) & 0xFF;
210     } else {
211         rc = read(_base_spe_open_if_closed(spectx, FD_WBOX_STAT, 0), &ret, 4);
212         if (rc != 4)
213             ret = -1;
214     }
215
216     return ret;
217 }
218 }

```

Here is the call graph for this function:



### 3.23.4.22 `int _base_spe_in_mbox_write (spe_context_ptr_t spectx, unsigned int mbox_data[], int count, int behavior_flag)`

The `_base_spe_in_mbox_write` function writes `mbox_data` to the SPE inbound mailbox for the SPE thread `speid`.

If the behavior flag indicates `ALL_BLOCKING` the call will try to write exactly `count` mailbox entries and block until the write request is satisfied, i.e., exactly `count` mailbox entries have been written. If the behavior flag indicates `ANY_BLOCKING` the call will try to write up to `count` mailbox entries, and block until the write request is satisfied, i.e., at least 1 mailbox entry has been written. If the behavior flag indicates `ANY_NON_BLOCKING` the call will not block until the write request is satisfied, but instead write whatever is immediately possible and return the number of mailbox entries written. `spe_stat_in_mbox` can be called to ensure that data can be written prior to calling the function.

#### Parameters:

*spectx* Specifies the SPE thread whose outbound mailbox is to be read.

*mbox\_data*

*count*

*behavior\_flag* `ALL_BLOCKING`

`ANY_BLOCKING`

`ANY_NON_BLOCKING`

#### Return values:

`>=0` the number of 32-bit mailbox messages written

`-1` error condition and `errno` is set

Possible values for `errno`:

`EINVAL` *spectx* is invalid

Exxxx what else do we need here??

### 3.23.4.23 `void* _base_spe_ls_area_get (struct spe_context * spectx)`

`_base_spe_ls_area_get` returns a pointer to the start of the memory mapped local store area

#### Parameters:

*spectx* Specifies the SPE context

### 3.23.4.24 `int _base_spe_ls_size_get (spe_context_ptr_t spe)`

`_base_spe_ls_size_get` returns the size of the local store area

#### Parameters:

*spectx* Specifies the SPE context

Definition at line 105 of file `accessors.c`.

References `LS_SIZE`.

```
106 {
107     return LS_SIZE;
108 }
```

### 3.23.4.25 **int \_base\_spe\_mfcio\_get (spe\_context\_ptr\_t *spectx*, unsigned int *ls*, void \* *ea*, unsigned int *size*, unsigned int *tag*, unsigned int *tid*, unsigned int *rid*)**

The `_base_spe_mfcio_get` function places a get DMA command on the proxy command queue of the SPE thread specified by `speid`. The get command transfers `size` bytes of data starting at the effective address specified by `ea` to the local store address specified by `ls`. The DMA is identified by the tag id specified by `tag` and performed according to the transfer class and replacement class specified by `tid` and `rid` respectively.

#### Parameters:

- spectx* Specifies the SPE context
- ls* 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.
- tid* Specifies the transfer class identifier of the DMA command.
- rid* Specifies the replacement class identifier of the DMA command.

#### Returns:

On success, return 0. On failure, -1 is returned.

Definition at line 160 of file `dma.c`.

References `MFC_CMD_GET`.

```
167 {
168     return spe_do_mfc_get(spectx, ls, ea, size, tag, tid, rid, MFC_CMD_GET);
169 }
```

### 3.23.4.26 **int \_base\_spe\_mfcio\_getb (spe\_context\_ptr\_t *spectx*, unsigned int *ls*, void \* *ea*, unsigned int *size*, unsigned int *tag*, unsigned int *tid*, unsigned int *rid*)**

The `_base_spe_mfcio_getb` function is identical to `_base_spe_mfcio_get` except that it places a `getb` (get with barrier) DMA command on the proxy command queue. The barrier form ensures that this command and all 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.

#### Parameters:

- spectx* Specifies the SPE context
- ls* 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.
- tid* Specifies the transfer class identifier of the DMA command.
- rid* Specifies the replacement class identifier of the DMA command.

#### Returns:

On success, return 0. On failure, -1 is returned.

Definition at line 171 of file dma.c.

References MFC\_CMD\_GETB.

```
178 {
179     return spe_do_mfc_get(spectx, ls, ea, size, tag, rid, rid, MFC_CMD_GETB);
180 }
```

#### 3.23.4.27 **int \_base\_spe\_mfcio\_getf (spe\_context\_ptr\_t spectx, unsigned int ls, void \* ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)**

The `_base_spe_mfcio_getf` function is identical to `_base_spe_mfcio_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.

##### Parameters:

- spectx* Specifies the SPE context
- ls* 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.
- tid* Specifies the transfer class identifier of the DMA command.
- rid* Specifies the replacement class identifier of the DMA command.

##### Returns:

On success, return 0. On failure, -1 is returned.

Definition at line 182 of file dma.c.

References MFC\_CMD\_GETF.

```
189 {
190     return spe_do_mfc_get(spectx, ls, ea, size, tag, tid, rid, MFC_CMD_GETF);
191 }
```

#### 3.23.4.28 **int \_base\_spe\_mfcio\_put (spe\_context\_ptr\_t spectx, unsigned int ls, void \* ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)**

The `_base_spe_mfcio_put` function places a `put` DMA command on the proxy command queue of the SPE thread specified by `speid`. The `put` command transfers `size` bytes of data starting at the local store address specified by `ls` 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.

##### Parameters:

- spectx* Specifies the SPE context
- ls* 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.
- tid* Specifies the transfer class identifier of the DMA command.
- rid* Specifies the replacement class identifier of the DMA command.

**Returns:**

On success, return 0. On failure, -1 is returned.

Definition at line 126 of file dma.c.

References MFC\_CMD\_PUT.

```

133 {
134     return spe_do_mfc_put(spectx, ls, ea, size, tag, tid, rid, MFC_CMD_PUT);
135 }
```

#### 3.23.4.29 **int \_base\_spe\_mfcio\_putb (spe\_context\_ptr\_t spectx, unsigned int ls, void \* ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)**

The `_base_spe_mfcio_putb` function is identical to `_base_spe_mfcio_put` except that it places a putb (put with barrier) DMA command on the proxy command queue. The barrier form ensures that this command and all 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.

**Parameters:**

- spectx* Specifies the SPE context
- ls* 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.
- tid* Specifies the transfer class identifier of the DMA command.
- rid* Specifies the replacement class identifier of the DMA command.

**Returns:**

On success, return 0. On failure, -1 is returned.

Definition at line 137 of file dma.c.

References MFC\_CMD\_PUTB.

```

144 {
145     return spe_do_mfc_put(spectx, ls, ea, size, tag, tid, rid, MFC_CMD_PUTB);
146 }
```

#### 3.23.4.30 **int \_base\_spe\_mfcio\_putf (spe\_context\_ptr\_t spectx, unsigned int ls, void \* ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)**

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

**Parameters:**

- spectx* Specifies the SPE context
- ls* 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.
- tid* Specifies the transfer class identifier of the DMA command.
- rid* Specifies the replacement class identifier of the DMA command.

**Returns:**

On success, return 0. On failure, -1 is returned.

Definition at line 148 of file dma.c.

References MFC\_CMD\_PUTF.

```

155 {
156     return spe_do_mfc_put(spectx, ls, ea, size, tag, tid, rid, MFC_CMD_PUTF);
157 }
```

### 3.23.4.31 `int _base_spe_mfcio_tag_status_read (spe_context_ptr_t spectx, unsigned int mask, unsigned int behavior, unsigned int * tag_status)`

`_base_spe_mfcio_tag_status_read`

No Idea

Definition at line 307 of file dma.c.

References `spe_context_base_priv::active_tagmask`, `spe_context::base_private`, `spe_context_base_priv::flags`, `SPE_MAP_PS`, `SPE_TAG_ALL`, `SPE_TAG_ANY`, and `SPE_TAG_IMMEDIATE`.

```

308 {
309     if ( mask != 0 ) {
310         if (!(spectx->base_private->flags & SPE_MAP_PS))
311             mask = 0;
312     } else {
313         if ((spectx->base_private->flags & SPE_MAP_PS))
314             mask = spectx->base_private->active_tagmask;
315     }
316
317     if (!tag_status) {
318         errno = EINVAL;
319         return -1;
320     }
321
322     switch (behavior) {
323     case SPE_TAG_ALL:
324         return spe_mfcio_tag_status_read_all(spectx, mask, tag_status);
325     case SPE_TAG_ANY:
326         return spe_mfcio_tag_status_read_any(spectx, mask, tag_status);
327     case SPE_TAG_IMMEDIATE:
328         return spe_mfcio_tag_status_read_immediate(spectx, mask, tag_status);
329     default:
330         errno = EINVAL;
331         return -1;
332     }
333 }
```

### 3.23.4.32 `int _base_spe_mssync_start (spe_context_ptr_t spectx)`

`_base_spe_mssync_start` starts Multisource Synchronisation

#### Parameters:

*spectx* Specifies the SPE context

Definition at line 335 of file dma.c.

References `_base_spe_open_if_closed()`, `spe_context::base_private`, `FD_MSS`, `spe_context_base_priv::flags`, `spe_context_base_priv::mssync_mmap_base`, and `SPE_MAP_PS`.

```

336 {
337     int ret, fd;
338     unsigned int data = 1; /* Any value can be written here */
339
340     volatile struct spe_mssync_area *mss_area =
341         spectx->base_private->mssync_mmap_base;
342
343     if (spectx->base_private->flags & SPE_MAP_PS) {
344         mss_area->MFC_MSSync = data;
345         return 0;
346     } else {
347         fd = _base_spe_open_if_closed(spectx, FD_MSS, 0);
348         if (fd != -1) {
349             ret = write(fd, &data, sizeof (data));
350             if ((ret < 0) && (errno != EIO)) {
351                 perror("spe_mssync_start: internal error");
352             }
353             return ret < 0 ? -1 : 0;
354         } else
355             return -1;
356     }
357 }
```

Here is the call graph for this function:



### 3.23.4.33 `int _base_spe_mssync_status (spe_context_ptr_t spectx)`

`_base_spe_mssync_status` retrieves status of Multisource Synchronisation

#### Parameters:

*spectx* Specifies the SPE context

Definition at line 359 of file dma.c.

References `_base_spe_open_if_closed()`, `spe_context::base_private`, `FD_MSS`, `spe_context_base_priv::flags`, `spe_context_base_priv::mssync_mmap_base`, and `SPE_MAP_PS`.

```

360 {
361     int ret, fd;
362     unsigned int data;
363
364     volatile struct spe_mssync_area *mss_area =
```



```

365         spectx->base_private->mssync_mmap_base;
366
367     if (spectx->base_private->flags & SPE_MAP_PS) {
368         return mss_area->MFC_MSSync;
369     } else {
370         fd = _base_spe_open_if_closed(spectx, FD_MSS, 0);
371         if (fd != -1) {
372             ret = read(fd, &data, sizeof (data));
373             if ((ret < 0) && (errno != EIO)) {
374                 perror("spe_mssync_start: internal error");
375             }
376             return ret < 0 ? -1 : data;
377         } else
378             return -1;
379     }
380 }

```

Here is the call graph for this function:



#### 3.23.4.34 int \_base\_spe\_out\_intr\_mbox\_read (spe\_context\_ptr\_t spectx, unsigned int mbox\_data[], int count, int behavior\_flag)

The `_base_spe_out_intr_mbox_read` function reads the contents of the SPE outbound interrupting mailbox for the SPE context.

Definition at line 255 of file `mbox.c`.

References `_base_spe_open_if_closed()`, `FD_IBOX`, `FD_IBOX_NB`, `SPE_MBOX_ALL_BLOCKING`, `SPE_MBOX_ANY_BLOCKING`, and `SPE_MBOX_ANY_NONBLOCKING`.

```

259 {
260     int rc;
261     int total;
262
263     if (mbox_data == NULL || count < 1) {
264         errno = EINVAL;
265         return -1;
266     }
267
268     switch (behavior_flag) {
269     case SPE_MBOX_ALL_BLOCKING: // read all, even if blocking
270         total = rc = 0;
271         while (total < 4*count) {
272             rc = read(_base_spe_open_if_closed(spectx, FD_IBOX, 0),
273                     (char *)mbox_data + total, 4*count - total);
274             if (rc == -1) {
275                 break;
276             }
277             total += rc;
278         }
279         break;
280
281     case SPE_MBOX_ANY_BLOCKING: // read at least one, even if blocking
282         total = rc = read(_base_spe_open_if_closed(spectx, FD_IBOX, 0), mbox_data, 4*count);
283         break;
284
285     case SPE_MBOX_ANY_NONBLOCKING: // only read, if non blocking
286         rc = read(_base_spe_open_if_closed(spectx, FD_IBOX_NB, 0), mbox_data, 4*count);

```

```

287         if (rc == -1 && errno == EAGAIN) {
288             rc = 0;
289             errno = 0;
290         }
291         total = rc;
292         break;
293     default:
294         errno = EINVAL;
295         return -1;
296     }
297 }
298
299 if (rc == -1) {
300     errno = EIO;
301     return -1;
302 }
303
304 return rc / 4;
305 }

```

Here is the call graph for this function:



### 3.23.4.35 int \_base\_spe\_out\_intr\_mbox\_status (spe\_context\_ptr\_t *spectx*)

The `_base_spe_out_intr_mbox_status` function fetches the status of the SPU outbound interrupt mailbox for the SPE thread specified by the `speid` 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:

*spectx* Specifies the SPE context whose mailbox status is to be read.

#### Returns:

On success, returns the current status of the mailbox, respectively. On failure, -1 is returned.

#### See also:

`spe_read_out_mbox`, `spe_write_in_mbox`, `read (2)`

Definition at line 238 of file `mbox.c`.

References `_base_spe_open_if_closed()`, `spe_context::base_private`, `spe_context_base_priv::cntl_mmap_base`, `FD_IBOX_STAT`, `spe_context_base_priv::flags`, and `SPE_MAP_PS`.

```

239 {
240     int rc, ret;
241     volatile struct spe_spu_control_area *cntl_area =
242         spectx->base_private->cntl_mmap_base;
243
244     if (spectx->base_private->flags & SPE_MAP_PS) {
245         ret = (cntl_area->SPU_Mbox_Stat >> 16) & 0xFF;
246     } else {
247         rc = read(_base_spe_open_if_closed(spectx, FD_IBOX_STAT, 0), &ret, 4);
248         if (rc != 4)
249             ret = -1;
250     }
251     return ret;
252 }
253 }

```

Here is the call graph for this function:



### 3.23.4.36 int \_base\_spe\_out\_mbox\_read (spe\_context\_ptr\_t *spectx*, unsigned int *mbox\_data*[], int *count*)

The `_base_spe_out_mbox_read` function reads the contents of the SPE outbound interrupting mailbox for the SPE thread `speid`.

The call will not block until the read request is satisfied, but instead return up to `count` currently available mailbox entries.

`spe_stat_out_intr_mbox` can be called to ensure that data is available prior to reading the outbound interrupting mailbox.

#### Parameters:

*spectx* Specifies the SPE thread whose outbound mailbox is to be read.

*mbox\_data*

*count*

#### Return values:

**>0** the number of 32-bit mailbox messages read

**=0** no data available

**-1** error condition and `errno` is set

Possible values for `errno`:

`EINVAL` `speid` is invalid

Exxxx what else do we need here??

Definition at line 58 of file `mbox.c`.

References `_base_spe_open_if_closed()`, `spe_context::base_private`, `DEBUG_PRINTF`, `FD_MBOX`, `spe_context_base_priv::flags`, and `SPE_MAP_PS`.

```

61 {
62     int rc;
63
64     if (mbox_data == NULL || count < 1){
65         errno = EINVAL;
66         return -1;
67     }
68
69     if (spectx->base_private->flags & SPE_MAP_PS) {
70         rc = _base_spe_out_mbox_read_ps(spectx, mbox_data, count);
71     } else {
72         rc = read(_base_spe_open_if_closed(spectx, FD_MBOX, 0), mbox_data, count*4);
73         DEBUG_PRINTF("%s read rc: %d\n", __FUNCTION__, rc);
74         if (rc != -1) {
75             rc /= 4;
76         } else {
77             if (errno == EAGAIN ) { // no data ready to be read
78                 errno = 0;
79                 rc = 0;
80             }

```

```

81         }
82     }
83     return rc;
84 }

```

Here is the call graph for this function:



### 3.23.4.37 int \_base\_spe\_out\_mbox\_status (spe\_context\_ptr\_t *spectx*)

The `_base_spe_out_mbox_status` function fetches the status of the SPU outbound mailbox for the SPE thread specified by the `speid` 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:

*spectx* Specifies the SPE context whose mailbox status is to be read.

#### Returns:

On success, returns the current status of the mailbox, respectively. On failure, -1 is returned.

#### See also:

`spe_read_out_mbox`, `spe_write_in_mbox`, `read` (2)

Definition at line 220 of file `mbox.c`.

References `_base_spe_open_if_closed()`, `spe_context::base_private`, `spe_context_base_priv::cntl_mmap_base`, `FD_MBOX_STAT`, `spe_context_base_priv::flags`, and `SPE_MAP_PS`.

```

221 {
222     int rc, ret;
223     volatile struct spe_spu_control_area *cntl_area =
224         spectx->base_private->cntl_mmap_base;
225
226     if (spectx->base_private->flags & SPE_MAP_PS) {
227         ret = cntl_area->SPU_Mbox_Stat & 0xFF;
228     } else {
229         rc = read(_base_spe_open_if_closed(spectx, FD_MBOX_STAT, 0), &ret, 4);
230         if (rc != 4)
231             ret = -1;
232     }
233
234     return ret;
235 }
236 }

```

Here is the call graph for this function:



### 3.23.4.38 `int _base_spe_program_load (spe_context_ptr_t spectx, spe_program_handle_t * program)`

`_base_spe_program_load` loads an ELF image into a context

#### Parameters:

*spectx* Specifies the SPE context

*program* handle to the ELF image

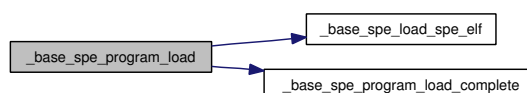
Definition at line 189 of file load.c.

References `_base_spe_load_spe_elf()`, `_base_spe_program_load_complete()`, `spe_context::base_private`, `DEBUG_PRINTF`, `spe_context_base_priv::emulated_entry`, `spe_ld_info::entry`, `spe_context_base_priv::entry`, `spe_context_base_priv::flags`, `spe_context_base_priv::loaded_program`, `spe_context_base_priv::mem_mmap_base`, `SPE_ISOLATE`, and `SPE_ISOLATE_EMULATE`.

```

190 {
191     int rc = 0;
192     struct spe_ld_info ld_info;
193
194     spe->base_private->loaded_program = program;
195
196     if (spe->base_private->flags & SPE_ISOLATE) {
197         rc = spe_start_isolated_app(spe, program);
198
199     } else if (spe->base_private->flags & SPE_ISOLATE_EMULATE) {
200         rc = spe_start_emulated_isolated_app(spe, program, &ld_info);
201
202     } else {
203         rc = _base_spe_load_spe_elf(program,
204                                     spe->base_private->mem_mmap_base, &ld_info);
205         if (!rc)
206             _base_spe_program_load_complete(spe);
207     }
208
209     if (rc != 0) {
210         DEBUG_PRINTF ("Load SPE ELF failed..\n");
211         return -1;
212     }
213
214     spe->base_private->entry = ld_info.entry;
215     spe->base_private->emulated_entry = ld_info.entry;
216
217     return 0;
218 }
```

Here is the call graph for this function:



### 3.23.4.39 `void _base_spe_program_load_complete (spe_context_ptr_t spectx)`

Signal that the program load has completed. For normal apps, this is called directly in the load path. For (emulated) isolated apps, the load is asynchronous, so this needs to be called when we know that the load has completed

**Precondition:**

`spe->base_priv->loaded_program` is a valid SPE program

**Parameters:**

*spectx* The spe context that has been loaded.

Register the SPE program's start address with the oprofile and gdb, by writing to the object-id file.

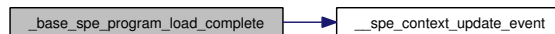
Definition at line 37 of file load.c.

References `__spe_context_update_event()`, `spe_context::base_private`, `DEBUG_PRINTF`, `spe_program_handle_t::elf_image`, `spe_context_base_priv::fd_spe_dir`, and `spe_context_base_priv::loaded_program`.

```

38 {
39     int objfd, len;
40     char buf[20];
41     spe_program_handle_t *program;
42
43     program = spectx->base_private->loaded_program;
44
45     if (!program || !program->elf_image) {
46         DEBUG_PRINTF("%s called, but no program loaded\n", __func__);
47         return;
48     }
49
50     objfd = openat(spectx->base_private->fd_spe_dir, "object-id", O_RDWR);
51     if (objfd < 0)
52         return;
53
54     len = sprintf(buf, "%p", program->elf_image);
55     write(objfd, buf, len + 1);
56     close(objfd);
57
58     __spe_context_update_event();
59 }
```

Here is the call graph for this function:



#### 3.23.4.40 void\* `_base_spe_ps_area_get` (struct `spe_context` \* *spectx*, enum `ps_area` *area*)

`_base_spe_ps_area_get` returns a pointer to the start of memory mapped problem state area

**Parameters:**

*spectx* Specifies the SPE context

*area* specifies the area to map

#### 3.23.4.41 int `_base_spe_signal_write` (spe\_context\_ptr\_t *spectx*, unsigned int *signal\_reg*, unsigned int *data*)

The `_base_spe_signal_write` function writes data to the signal notification register specified by `signal_reg` for the SPE thread specified by the `speid` parameter.

**Parameters:**

**spectx** Specifies the SPE context whose signal register is to be written to.

**signal\_reg** Specified 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.

**Returns:**

On success, spe\_write\_signal returns 0. On failure, -1 is returned.

**See also:**

spe\_get\_ps\_area, spe\_write\_in\_mbox

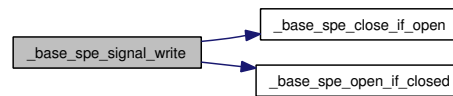
Definition at line 307 of file mbox.c.

References `_base_spe_close_if_open()`, `_base_spe_open_if_closed()`, `spe_context::base_private`, `FD_SIG1`, `FD_SIG2`, `spe_context_base_priv::flags`, `spe_context_base_priv::signal1_mmap_base`, `spe_context_base_priv::signal2_mmap_base`, `SPE_MAP_PS`, `SPE_SIG_NOTIFY_REG_1`, `SPE_SIG_NOTIFY_REG_2`, `spe_sig_notify_1_area_t::SPU_Sig_Notify_1`, and `spe_sig_notify_2_area_t::SPU_Sig_Notify_2`.

```

310 {
311     int rc;
312
313     if (spectx->base_private->flags & SPE_MAP_PS) {
314         if (signal_reg == SPE_SIG_NOTIFY_REG_1) {
315             spe_sig_notify_1_area_t *sig = spectx->base_private->signal1_mmap_base;
316
317             sig->SPU_Sig_Notify_1 = data;
318         } else if (signal_reg == SPE_SIG_NOTIFY_REG_2) {
319             spe_sig_notify_2_area_t *sig = spectx->base_private->signal2_mmap_base;
320
321             sig->SPU_Sig_Notify_2 = data;
322         } else {
323             errno = EINVAL;
324             return -1;
325         }
326         rc = 0;
327     } else {
328         if (signal_reg == SPE_SIG_NOTIFY_REG_1)
329             rc = write(_base_spe_open_if_closed(spectx, FD_SIG1, 0), &data, 4);
330         else if (signal_reg == SPE_SIG_NOTIFY_REG_2)
331             rc = write(_base_spe_open_if_closed(spectx, FD_SIG2, 0), &data, 4);
332         else {
333             errno = EINVAL;
334             return -1;
335         }
336
337         if (rc == 4)
338             rc = 0;
339
340         if (signal_reg == SPE_SIG_NOTIFY_REG_1)
341             _base_spe_close_if_open(spectx, FD_SIG1);
342         else if (signal_reg == SPE_SIG_NOTIFY_REG_2)
343             _base_spe_close_if_open(spectx, FD_SIG2);
344     }
345
346     return rc;
347 }
```

Here is the call graph for this function:



#### 3.23.4.42 int \_base\_spe\_stop\_reason\_get (spe\_context\_ptr\_t *spectx*)

\_base\_spe\_stop\_reason\_get

##### Parameters:

*spectx* one thread for which to check why it was stopped

##### Return values:

- 0* success - eventid and eventdata set appropriately
- 1* spe has not stopped after checking last, so no data was written to event
- 1* an error has happened, event was not touched, errno gets set
  - Possible vales for errno:
  - EINVAL speid is invalid
  - Exxxx what else do we need here??

#### 3.23.4.43 int \_base\_spe\_stop\_status\_get (spe\_context\_ptr\_t *spectx*)

\_base\_spe\_stop\_status\_get

##### Parameters:

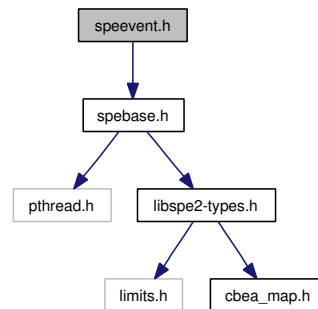
*spectx* Specifies the SPE context



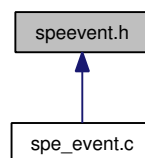
## 3.24 speevent.h File Reference

```
#include "spebase.h"
```

Include dependency graph for speevent.h:



This graph shows which files directly or indirectly include this file:



### Data Structures

- struct [spe\\_context\\_event\\_priv\\_t](#)

### Typedefs

- typedef struct [spe\\_context\\_event\\_priv\\_t](#) \* [spe\\_context\\_event\\_priv\\_ptr\\_t](#)

### Enumerations

- enum [\\_\\_spe\\_event\\_types](#) {  
[\\_\\_SPE\\_EVENT\\_OUT\\_INTR\\_MBOX](#), [\\_\\_SPE\\_EVENT\\_IN\\_MBOX](#), [\\_\\_SPE\\_EVENT\\_TAG\\_-](#)  
[GROUP](#), [\\_\\_SPE\\_EVENT\\_SPE\\_STOPPED](#),  
[\\_\\_NUM\\_SPE\\_EVENT\\_TYPES](#) }

### Functions

- int [\\_event\\_spe\\_stop\\_info\\_read](#) ([spe\\_context\\_ptr\\_t](#) spe, [spe\\_stop\\_info\\_t](#) \*stopinfo)
- [spe\\_event\\_handler\\_ptr\\_t](#) [\\_event\\_spe\\_event\\_handler\\_create](#) (void)
- int [\\_event\\_spe\\_event\\_handler\\_destroy](#) ([spe\\_event\\_handler\\_ptr\\_t](#) evhandler)
- int [\\_event\\_spe\\_event\\_handler\\_register](#) ([spe\\_event\\_handler\\_ptr\\_t](#) evhandler, [spe\\_event\\_unit\\_-](#)  
[t](#) \*event)

- `int _event_spe_event_handler_deregister (spe_event_handler_ptr_t evhandler, spe_event_unit_t *event)`
- `int _event_spe_event_wait (spe_event_handler_ptr_t evhandler, spe_event_unit_t *events, int max_events, int timeout)`
- `int _event_spe_context_finalize (spe_context_ptr_t spe)`
- `struct spe_context_event_priv * _event_spe_context_initialize (spe_context_ptr_t spe)`
- `int _event_spe_context_run (spe_context_ptr_t spe, unsigned int *entry, unsigned int runflags, void *argp, void *envp, spe_stop_info_t *stopinfo)`
- `void _event_spe_context_lock (spe_context_ptr_t spe)`
- `void _event_spe_context_unlock (spe_context_ptr_t spe)`

### 3.24.1 Typedef Documentation

#### 3.24.1.1 `typedef struct spe_context_event_priv_t * spe_context_event_priv_ptr_t`

### 3.24.2 Enumeration Type Documentation

#### 3.24.2.1 `enum __spe_event_types`

Enumerator:

```
__SPE_EVENT_OUT_INTR_MBOX
__SPE_EVENT_IN_MBOX
__SPE_EVENT_TAG_GROUP
__SPE_EVENT_SPE_STOPPED
__NUM_SPE_EVENT_TYPES
```

Definition at line 28 of file `speevent.h`.

```
28 {
29  __SPE_EVENT_OUT_INTR_MBOX, __SPE_EVENT_IN_MBOX,
30  __SPE_EVENT_TAG_GROUP, __SPE_EVENT_SPE_STOPPED,
31  __NUM_SPE_EVENT_TYPES
32 };
```

### 3.24.3 Function Documentation

#### 3.24.3.1 `int _event_spe_context_finalize (spe_context_ptr_t spe)`

Definition at line 416 of file `spe_event.c`.

References `__SPE_EVENT_CONTEXT_PRIV_GET`, `__SPE_EVENT_CONTEXT_PRIV_SET`, `spe_context_event_priv_t::lock`, `spe_context_event_priv_t::stop_event_pipe`, and `spe_context_event_priv_t::stop_event_read_lock`.

```
417 {
418  spe_context_event_priv_ptr_t evctx;
419
420  if (!spe) {
421      errno = ESRCH;
422      return -1;
423  }
424
425  evctx = __SPE_EVENT_CONTEXT_PRIV_GET (spe);
```

```

426  __SPE_EVENT_CONTEXT_PRIV_SET(spe, NULL);
427
428  close(evctx->stop_event_pipe[0]);
429  close(evctx->stop_event_pipe[1]);
430
431  pthread_mutex_destroy(&evctx->lock);
432  pthread_mutex_destroy(&evctx->stop_event_read_lock);
433
434  free(evctx);
435
436  return 0;
437 }

```

### 3.24.3.2 struct spe\_context\_event\_priv\* \_event\_spe\_context\_initialize (spe\_context\_ptr\_t spe) [read]

Definition at line 439 of file spe\_event.c.

References `spe_context_event_priv_t::events`, `spe_context_event_priv_t::lock`, `spe_event_unit_t::spe`, `spe_context_event_priv_t::stop_event_pipe`, and `spe_context_event_priv_t::stop_event_read_lock`.

```

440 {
441  spe_context_event_priv_ptr_t evctx;
442  int rc;
443  int i;
444
445  evctx = calloc(1, sizeof(*evctx));
446  if (!evctx) {
447      return NULL;
448  }
449
450  rc = pipe(evctx->stop_event_pipe);
451  if (rc == -1) {
452      free(evctx);
453      return NULL;
454  }
455  rc = fcntl(evctx->stop_event_pipe[0], F_GETFL);
456  if (rc != -1) {
457      rc = fcntl(evctx->stop_event_pipe[0], F_SETFL, rc | O_NONBLOCK);
458  }
459  if (rc == -1) {
460      close(evctx->stop_event_pipe[0]);
461      close(evctx->stop_event_pipe[1]);
462      free(evctx);
463      errno = EIO;
464      return NULL;
465  }
466
467  for (i = 0; i < sizeof(evctx->events) / sizeof(evctx->events[0]); i++) {
468      evctx->events[i].spe = spe;
469  }
470
471  pthread_mutex_init(&evctx->lock, NULL);
472  pthread_mutex_init(&evctx->stop_event_read_lock, NULL);
473
474  return evctx;
475 }

```

### 3.24.3.3 void \_event\_spe\_context\_lock (spe\_context\_ptr\_t spe)

Definition at line 49 of file spe\_event.c.

References `__SPE_EVENT_CONTEXT_PRIV_GET`.

```
50 {
51     pthread_mutex_lock(&__SPE_EVENT_CONTEXT_PRIV_GET(spe)->lock);
52 }
```

#### 3.24.3.4 `int _event_spe_context_run (spe_context_ptr_t spe, unsigned int * entry, unsigned int runflags, void * argp, void * envp, spe_stop_info_t * stopinfo)`

Definition at line 477 of file `spe_event.c`.

References `__SPE_EVENT_CONTEXT_PRIV_GET`, `_base_spe_context_run()`, and `spe_context_event_priv_t::stop_event_pipe`.

```
478 {
479     spe_context_event_priv_ptr_t evctx;
480     spe_stop_info_t stopinfo_buf;
481     int rc;
482
483     if (!stopinfo) {
484         stopinfo = &stopinfo_buf;
485     }
486     rc = _base_spe_context_run(spe, entry, runflags, argp, envp, stopinfo);
487
488     evctx = __SPE_EVENT_CONTEXT_PRIV_GET(spe);
489     if (write(evctx->stop_event_pipe[1], stopinfo, sizeof(*stopinfo)) != sizeof(*stopinfo)) {
490         /* error check. */
491     }
492
493     return rc;
494 }
```

Here is the call graph for this function:



#### 3.24.3.5 `void _event_spe_context_unlock (spe_context_ptr_t spe)`

Definition at line 54 of file `spe_event.c`.

References `__SPE_EVENT_CONTEXT_PRIV_GET`.

```
55 {
56     pthread_mutex_unlock(&__SPE_EVENT_CONTEXT_PRIV_GET(spe)->lock);
57 }
```

#### 3.24.3.6 `spe_event_handler_ptr_t _event_spe_event_handler_create (void)`

Definition at line 110 of file `spe_event.c`.

References `__SPE_EPOLL_FD_SET`, and `__SPE_EPOLL_SIZE`.

```

111 {
112     int epfd;
113     spe_event_handler_t *evhandler;
114
115     evhandler = calloc(1, sizeof(*evhandler));
116     if (!evhandler) {
117         return NULL;
118     }
119
120     epfd = epoll_create(__SPE_EPOLL_SIZE);
121     if (epfd == -1) {
122         free(evhandler);
123         return NULL;
124     }
125
126     __SPE_EPOLL_FD_SET(evhandler, epfd);
127
128     return evhandler;
129 }

```

### 3.24.3.7 `int _event_spe_event_handler_deregister (spe_event_handler_ptr_t evhandler, spe_event_unit_t *event)`

Definition at line 273 of file `spe_event.c`.

References `__base_spe_event_source_acquire()`, `__SPE_EPOLL_FD_GET`, `__SPE_EVENT_ALL`, `__SPE_EVENT_CONTEXT_PRIV_GET`, `__SPE_EVENT_IN_MBOX`, `__SPE_EVENT_OUT_INTR_MBOX`, `__SPE_EVENT_SPE_STOPPED`, `__SPE_EVENT_TAG_GROUP`, `__SPE_EVENTS_ENABLED`, `_event_spe_context_lock()`, `_event_spe_context_unlock()`, `spe_context_event_priv_t::events`, `spe_event_unit_t::events`, `FD_IBOX`, `FD_MFC`, `FD_WBOX`, `spe_event_unit_t::spe`, `SPE_EVENT_IN_MBOX`, `SPE_EVENT_OUT_INTR_MBOX`, `SPE_EVENT_SPE_STOPPED`, `SPE_EVENT_TAG_GROUP`, and `spe_context_event_priv_t::stop_event_pipe`.

```

274 {
275     int epfd;
276     const int ep_op = EPOLL_CTL_DEL;
277     spe_context_event_priv_ptr_t evctx;
278     int fd;
279
280     if (!evhandler) {
281         errno = ESRCH;
282         return -1;
283     }
284     if (!event || !event->spe) {
285         errno = EINVAL;
286         return -1;
287     }
288     if (!__SPE_EVENTS_ENABLED(event->spe)) {
289         errno = ENOTSUP;
290         return -1;
291     }
292
293     epfd = __SPE_EPOLL_FD_GET(evhandler);
294     evctx = __SPE_EVENT_CONTEXT_PRIV_GET(event->spe);
295
296     if (event->events & ~__SPE_EVENT_ALL) {
297         errno = ENOTSUP;
298         return -1;
299     }
300
301     _event_spe_context_lock(event->spe); /* for spe->event_private->events */
302
303     if (event->events & SPE_EVENT_OUT_INTR_MBOX) {

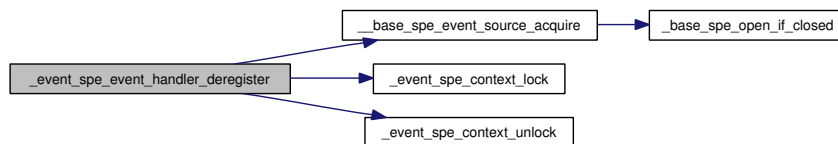
```

```

304     fd = __base_spe_event_source_acquire(event->spe, FD_IBOX);
305     if (fd == -1) {
306         _event_spe_context_unlock(event->spe);
307         return -1;
308     }
309     if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
310         _event_spe_context_unlock(event->spe);
311         return -1;
312     }
313     evctx->events[__SPE_EVENT_OUT_INTR_MBOX].events = 0;
314 }
315
316 if (event->events & SPE_EVENT_IN_MBOX) {
317     fd = __base_spe_event_source_acquire(event->spe, FD_WBOX);
318     if (fd == -1) {
319         _event_spe_context_unlock(event->spe);
320         return -1;
321     }
322     if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
323         _event_spe_context_unlock(event->spe);
324         return -1;
325     }
326     evctx->events[__SPE_EVENT_IN_MBOX].events = 0;
327 }
328
329 if (event->events & SPE_EVENT_TAG_GROUP) {
330     fd = __base_spe_event_source_acquire(event->spe, FD_MFC);
331     if (fd == -1) {
332         _event_spe_context_unlock(event->spe);
333         return -1;
334     }
335     if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
336         _event_spe_context_unlock(event->spe);
337         return -1;
338     }
339     evctx->events[__SPE_EVENT_TAG_GROUP].events = 0;
340 }
341
342 if (event->events & SPE_EVENT_SPE_STOPPED) {
343     fd = evctx->stop_event_pipe[0];
344     if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
345         _event_spe_context_unlock(event->spe);
346         return -1;
347     }
348     evctx->events[__SPE_EVENT_SPE_STOPPED].events = 0;
349 }
350
351 _event_spe_context_unlock(event->spe);
352
353 return 0;
354 }

```

Here is the call graph for this function:



**3.24.3.8 int \_event\_spe\_event\_handler\_destroy (spe\_event\_handler\_ptr\_t *evhandler*)**

Definition at line 135 of file spe\_event.c.

References `__SPE_EPOLL_FD_GET`.

```

136 {
137     int epfd;
138
139     if (!evhandler) {
140         errno = ESRCH;
141         return -1;
142     }
143
144     epfd = __SPE_EPOLL_FD_GET(evhandler);
145     close(epfd);
146
147     free(evhandler);
148     return 0;
149 }
```

**3.24.3.9 int \_event\_spe\_event\_handler\_register (spe\_event\_handler\_ptr\_t *evhandler*, spe\_event\_unit\_t \* *event*)**

Definition at line 155 of file spe\_event.c.

References `__base_spe_event_source_acquire()`, `__SPE_EPOLL_FD_GET`, `__SPE_EVENT_ALL`, `__SPE_EVENT_CONTEXT_PRIV_GET`, `__SPE_EVENT_IN_MBOX`, `__SPE_EVENT_OUT_INTR_MBOX`, `__SPE_EVENT_SPE_STOPPED`, `__SPE_EVENT_TAG_GROUP`, `__SPE_EVENTS_ENABLED`, `_event_spe_context_lock()`, `_event_spe_context_unlock()`, `spe_context::base_private`, `spe_event_unit_t::data`, `spe_context_event_priv_t::events`, `spe_event_unit_t::events`, `FD_IBOX`, `FD_MFC`, `FD_WBOX`, `spe_context_base_priv::flags`, `spe_event_data_t::ptr`, `spe_event_unit_t::spe`, `SPE_EVENT_IN_MBOX`, `SPE_EVENT_OUT_INTR_MBOX`, `SPE_EVENT_SPE_STOPPED`, `SPE_EVENT_TAG_GROUP`, `SPE_MAP_PS`, and `spe_context_event_priv_t::stop_event_pipe`.

```

156 {
157     int epfd;
158     const int ep_op = EPOLL_CTL_ADD;
159     spe_context_event_priv_ptr_t evctx;
160     spe_event_unit_t *ev_buf;
161     struct epoll_event ep_event;
162     int fd;
163
164     if (!evhandler) {
165         errno = ESRCH;
166         return -1;
167     }
168     if (!event || !event->spe) {
169         errno = EINVAL;
170         return -1;
171     }
172     if (!__SPE_EVENTS_ENABLED(event->spe)) {
173         errno = ENOTSUP;
174         return -1;
175     }
176
177     epfd = __SPE_EPOLL_FD_GET(evhandler);
178     evctx = __SPE_EVENT_CONTEXT_PRIV_GET(event->spe);
179
180     if (event->events & ~__SPE_EVENT_ALL) {
181         errno = ENOTSUP;

```

```

182     return -1;
183 }
184
185 _event_spe_context_lock(event->spe); /* for spe->event_private->events */
186
187 if (event->events & SPE_EVENT_OUT_INTR_MBOX) {
188     fd = __base_spe_event_source_acquire(event->spe, FD_IBOX);
189     if (fd == -1) {
190         _event_spe_context_unlock(event->spe);
191         return -1;
192     }
193
194     ev_buf = &evctx->events[__SPE_EVENT_OUT_INTR_MBOX];
195     ev_buf->events = SPE_EVENT_OUT_INTR_MBOX;
196     ev_buf->data = event->data;
197
198     ep_event.events = EPOLLIN;
199     ep_event.data.ptr = ev_buf;
200     if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
201         _event_spe_context_unlock(event->spe);
202         return -1;
203     }
204 }
205
206 if (event->events & SPE_EVENT_IN_MBOX) {
207     fd = __base_spe_event_source_acquire(event->spe, FD_WBOX);
208     if (fd == -1) {
209         _event_spe_context_unlock(event->spe);
210         return -1;
211     }
212
213     ev_buf = &evctx->events[__SPE_EVENT_IN_MBOX];
214     ev_buf->events = SPE_EVENT_IN_MBOX;
215     ev_buf->data = event->data;
216
217     ep_event.events = EPOLLOUT;
218     ep_event.data.ptr = ev_buf;
219     if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
220         _event_spe_context_unlock(event->spe);
221         return -1;
222     }
223 }
224
225 if (event->events & SPE_EVENT_TAG_GROUP) {
226     fd = __base_spe_event_source_acquire(event->spe, FD_MFC);
227     if (fd == -1) {
228         _event_spe_context_unlock(event->spe);
229         return -1;
230     }
231
232     if (event->spe->base_private->flags & SPE_MAP_PS) {
233         _event_spe_context_unlock(event->spe);
234         errno = ENOTSUP;
235         return -1;
236     }
237
238     ev_buf = &evctx->events[__SPE_EVENT_TAG_GROUP];
239     ev_buf->events = SPE_EVENT_TAG_GROUP;
240     ev_buf->data = event->data;
241
242     ep_event.events = EPOLLIN;
243     ep_event.data.ptr = ev_buf;
244     if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
245         _event_spe_context_unlock(event->spe);
246         return -1;
247     }
248 }

```

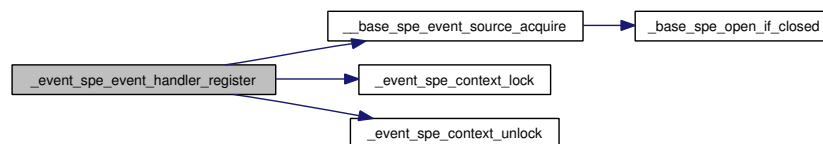


```

249
250     if (event->events & SPE_EVENT_SPE_STOPPED) {
251         fd = evctx->stop_event_pipe[0];
252
253         ev_buf = &evctx->events[__SPE_EVENT_SPE_STOPPED];
254         ev_buf->events = SPE_EVENT_SPE_STOPPED;
255         ev_buf->data = event->data;
256
257         ep_event.events = EPOLLIN;
258         ep_event.data.ptr = ev_buf;
259         if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
260             _event_spe_context_unlock(event->spe);
261             return -1;
262         }
263     }
264
265     _event_spe_context_unlock(event->spe);
266
267     return 0;
268 }

```

Here is the call graph for this function:



#### 3.24.3.10 int \_event\_spe\_event\_wait (spe\_event\_handler\_ptr\_t *evhandler*, spe\_event\_unit\_t \* *events*, int *max\_events*, int *timeout*)

Definition at line 360 of file spe\_event.c.

References `__SPE_EPOLL_FD_GET`, `_event_spe_context_lock()`, `_event_spe_context_unlock()`, and `spe_event_unit_t::spe`.

```

361 {
362     int epfd;
363     struct epoll_event *ep_events;
364     int rc;
365
366     if (!evhandler) {
367         errno = ESRCH;
368         return -1;
369     }
370     if (!events || max_events <= 0) {
371         errno = EINVAL;
372         return -1;
373     }
374
375     epfd = __SPE_EPOLL_FD_GET(evhandler);
376
377     ep_events = malloc(sizeof(*ep_events) * max_events);
378     if (!ep_events) {
379         return -1;
380     }
381
382     for ( ; ; ) {
383         rc = epoll_wait(epfd, ep_events, max_events, timeout);

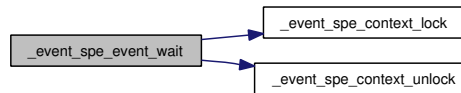
```

```

384     if (rc == -1) { /* error */
385         if (errno == EINTR) {
386             if (timeout >= 0) { /* behave as timeout */
387                 rc = 0;
388                 break;
389             }
390             /* else retry */
391         }
392         else {
393             break;
394         }
395     }
396     else if (rc > 0) {
397         int i;
398         for (i = 0; i < rc; i++) {
399             spe_event_unit_t *ev = (spe_event_unit_t *) (ep_events[i].data.ptr);
400             _event_spe_context_lock(ev->spe); /* lock ev itself */
401             events[i] = *ev;
402             _event_spe_context_unlock(ev->spe);
403         }
404         break;
405     }
406     else { /* timeout */
407         break;
408     }
409 }
410
411 free(ep_events);
412
413 return rc;
414 }

```

Here is the call graph for this function:



### 3.24.3.11 int \_event\_spe\_stop\_info\_read (spe\_context\_ptr\_t spe, spe\_stop\_info\_t \* stopinfo)

Definition at line 59 of file spe\_event.c.

References `__SPE_EVENT_CONTEXT_PRIV_GET`, `spe_context_event_priv_t::stop_event_pipe`, and `spe_context_event_priv_t::stop_event_read_lock`.

```

60 {
61     spe_context_event_priv_ptr_t evctx;
62     int rc;
63     int fd;
64     size_t total;
65
66     evctx = __SPE_EVENT_CONTEXT_PRIV_GET(spe);
67     fd = evctx->stop_event_pipe[0];
68
69     pthread_mutex_lock(&evctx->stop_event_read_lock); /* for atomic read */
70
71     rc = read(fd, stopinfo, sizeof(*stopinfo));
72     if (rc == -1) {
73         pthread_mutex_unlock(&evctx->stop_event_read_lock);
74         return -1;
75     }

```

```
76
77     total = rc;
78     while (total < sizeof(*stopinfo)) { /* this loop will be executed in few cases */
79         struct pollfd fds;
80         fds.fd = fd;
81         fds.events = POLLIN;
82         rc = poll(&fds, 1, -1);
83         if (rc == -1) {
84             if (errno != EINTR) {
85                 break;
86             }
87         }
88         else if (rc == 1) {
89             rc = read(fd, (char *)stopinfo + total, sizeof(*stopinfo) - total);
90             if (rc == -1) {
91                 if (errno != EAGAIN) {
92                     break;
93                 }
94             }
95             else {
96                 total += rc;
97             }
98         }
99     }
100
101     pthread_mutex_unlock(&evctx->stop_event_read_lock);
102
103     return rc == -1 ? -1 : 0;
104 }
```

# Index

- \_\_NUM\_SPE\_EVENT\_TYPES
  - speevent.h, 166
- \_\_PRINTF
  - elf\_loader.c, 61
  - handler\_utils.h, 70
  - spebase.h, 130
- \_\_SPE\_EPOLL\_FD\_GET
  - spe\_event.c, 119
- \_\_SPE\_EPOLL\_FD\_SET
  - spe\_event.c, 119
- \_\_SPE\_EPOLL\_SIZE
  - spe\_event.c, 119
- \_\_SPE\_EVENTS\_ENABLED
  - spe\_event.c, 120
- \_\_SPE\_EVENT\_ALL
  - spe\_event.c, 119
- \_\_SPE\_EVENT\_CONTEXT\_PRIV\_GET
  - spe\_event.c, 119
- \_\_SPE\_EVENT\_CONTEXT\_PRIV\_SET
  - spe\_event.c, 119
- \_\_SPE\_EVENT\_IN\_MBOX
  - speevent.h, 166
- \_\_SPE\_EVENT\_OUT\_INTR\_MBOX
  - speevent.h, 166
- \_\_SPE\_EVENT\_SPE\_STOPPED
  - speevent.h, 166
- \_\_SPE\_EVENT\_TAG\_GROUP
  - speevent.h, 166
- \_\_attribute\_\_
  - accessors.c, 38
- \_\_base\_spe\_event\_source\_acquire
  - accessors.c, 38
  - spebase.h, 133
- \_\_base\_spe\_event\_source\_release
  - accessors.c, 38
  - spebase.h, 133
- \_\_base\_spe\_spe\_dir\_get
  - accessors.c, 38
  - spebase.h, 133
- \_\_base\_spe\_stop\_event\_source\_get
  - accessors.c, 38
  - spebase.h, 134
- \_\_base\_spe\_stop\_event\_target\_get
  - accessors.c, 39
  - spebase.h, 134
- \_\_reserved\_ptr
  - spe\_stop\_info\_t, 36
- \_\_reserved\_u64
  - spe\_stop\_info\_t, 36
- \_\_spe\_context\_update\_event
  - spebase.h, 134
- \_\_spe\_current\_active\_context
  - run.c, 117
- \_\_spe\_event\_types
  - speevent.h, 166
- \_base\_spe\_callback\_handler\_deregister
  - lib\_builtin.c, 83
  - spebase.h, 134
- \_base\_spe\_callback\_handler\_query
  - lib\_builtin.c, 83
  - spebase.h, 135
- \_base\_spe\_callback\_handler\_register
  - lib\_builtin.c, 83
  - spebase.h, 135
- \_base\_spe\_close\_if\_open
  - create.c, 43
  - create.h, 49
- \_base\_spe\_context\_create
  - create.c, 43
  - spebase.h, 136
- \_base\_spe\_context\_destroy
  - create.c, 46
  - spebase.h, 139
- \_base\_spe\_context\_lock
  - create.c, 46
  - spebase.h, 140
- \_base\_spe\_context\_run
  - run.c, 113
  - spebase.h, 140
- \_base\_spe\_context\_unlock
  - create.c, 46
  - spebase.h, 145
- \_base\_spe\_count\_physical\_cpus
  - info.c, 76
  - info.h, 79
- \_base\_spe\_count\_physical\_spes
  - info.c, 77
  - info.h, 80
- \_base\_spe\_count\_usable\_spes
  - info.c, 77

- info.h, 81
- \_base\_spe\_cpu\_info\_get
  - info.c, 77
  - spebase.h, 145
- \_base\_spe\_emulated\_loader\_present
  - load.c, 101
  - spebase.h, 146
- \_base\_spe\_gang\_context\_create
  - create.c, 47
  - spebase.h, 146
- \_base\_spe\_gang\_context\_destroy
  - create.c, 47
  - spebase.h, 147
- \_base\_spe\_handle\_library\_callback
  - lib\_builtin.c, 84
  - lib\_builtin.h, 87
- \_base\_spe\_image\_close
  - image.c, 73
  - spebase.h, 147
- \_base\_spe\_image\_open
  - image.c, 74
  - spebase.h, 148
- \_base\_spe\_in\_mbox\_status
  - mbox.c, 104
  - spebase.h, 150
- \_base\_spe\_in\_mbox\_write
  - mbox.c, 105
  - spebase.h, 150
- \_base\_spe\_load\_spe\_elf
  - elf\_loader.c, 61
  - elf\_loader.h, 66
- \_base\_spe\_ls\_area\_get
  - accessors.c, 39
  - spebase.h, 151
- \_base\_spe\_ls\_size\_get
  - accessors.c, 39
  - spebase.h, 151
- \_base\_spe\_mfcio\_get
  - dma.c, 53
  - spebase.h, 151
- \_base\_spe\_mfcio\_getb
  - dma.c, 53
  - spebase.h, 152
- \_base\_spe\_mfcio\_getf
  - dma.c, 53
  - spebase.h, 153
- \_base\_spe\_mfcio\_put
  - dma.c, 54
  - spebase.h, 153
- \_base\_spe\_mfcio\_putb
  - dma.c, 54
  - spebase.h, 154
- \_base\_spe\_mfcio\_putf
  - dma.c, 55
  - spebase.h, 154
- \_base\_spe\_mfcio\_tag\_status\_read
  - dma.c, 55
  - spebase.h, 155
- \_base\_spe\_mssync\_start
  - dma.c, 56
  - spebase.h, 155
- \_base\_spe\_mssync\_status
  - dma.c, 57
  - spebase.h, 156
- \_base\_spe\_open\_if\_closed
  - create.c, 48
  - create.h, 50
- \_base\_spe\_out\_intr\_mbox\_read
  - mbox.c, 106
  - spebase.h, 157
- \_base\_spe\_out\_intr\_mbox\_status
  - mbox.c, 107
  - spebase.h, 158
- \_base\_spe\_out\_mbox\_read
  - mbox.c, 108
  - spebase.h, 159
- \_base\_spe\_out\_mbox\_status
  - mbox.c, 109
  - spebase.h, 160
- \_base\_spe\_parse\_isolated\_elf
  - elf\_loader.c, 62
  - elf\_loader.h, 67
- \_base\_spe\_program\_load
  - load.c, 102
  - spebase.h, 160
- \_base\_spe\_program\_load\_complete
  - load.c, 102
  - spebase.h, 161
- \_base\_spe\_ps\_area\_get
  - accessors.c, 39
  - spebase.h, 162
- \_base\_spe\_signal\_write
  - mbox.c, 109
  - spebase.h, 162
- \_base\_spe\_stop\_reason\_get
  - spebase.h, 164
- \_base\_spe\_stop\_status\_get
  - spebase.h, 164
- \_base\_spe\_toe\_ear
  - elf\_loader.c, 63
  - elf\_loader.h, 68
- \_base\_spe\_verify\_spe\_elf\_image
  - elf\_loader.c, 64
  - elf\_loader.h, 69
- \_event\_spe\_context\_finalize
  - spe\_event.c, 120
  - speevent.h, 166
- \_event\_spe\_context\_initialize

- spe\_event.c, 120
- speevent.h, 167
- \_event\_spe\_context\_lock
  - spe\_event.c, 121
  - speevent.h, 167
- \_event\_spe\_context\_run
  - spe\_event.c, 121
  - speevent.h, 168
- \_event\_spe\_context\_unlock
  - spe\_event.c, 121
  - speevent.h, 168
- \_event\_spe\_event\_handler\_create
  - spe\_event.c, 122
  - speevent.h, 168
- \_event\_spe\_event\_handler\_deregister
  - spe\_event.c, 122
  - speevent.h, 169
- \_event\_spe\_event\_handler\_destroy
  - spe\_event.c, 123
  - speevent.h, 170
- \_event\_spe\_event\_handler\_register
  - spe\_event.c, 124
  - speevent.h, 171
- \_event\_spe\_event\_wait
  - spe\_event.c, 126
  - speevent.h, 173
- \_event\_spe\_stop\_info\_read
  - spe\_event.c, 126
  - speevent.h, 174
- accessors.c, 37
  - \_\_attribute\_\_, 38
  - \_\_base\_spe\_event\_source\_acquire, 38
  - \_\_base\_spe\_event\_source\_release, 38
  - \_\_base\_spe\_spe\_dir\_get, 38
  - \_\_base\_spe\_stop\_event\_source\_get, 38
  - \_\_base\_spe\_stop\_event\_target\_get, 39
  - \_\_base\_spe\_ls\_area\_get, 39
  - \_\_base\_spe\_ls\_size\_get, 39
  - \_\_base\_spe\_ps\_area\_get, 39
- active\_tagmask
  - spe\_context\_base\_priv, 15
- addr64, 5
  - ui, 5
  - ull, 5
- base\_private
  - spe\_context, 11
  - spe\_gang\_context, 22
- cbea\_map.h, 41
- class
  - mfc\_command\_parameter\_area, 8
- cmd
  - mfc\_command\_parameter\_area, 8
- cntl\_mmap\_base
  - spe\_context\_base\_priv, 14
- CNTL\_OFFSET
  - spebase.h, 130
- CNTL\_SIZE
  - spebase.h, 130
- create.c, 42
  - \_base\_spe\_close\_if\_open, 43
  - \_base\_spe\_context\_create, 43
  - \_base\_spe\_context\_destroy, 46
  - \_base\_spe\_context\_lock, 46
  - \_base\_spe\_context\_unlock, 46
  - \_base\_spe\_gang\_context\_create, 47
  - \_base\_spe\_gang\_context\_destroy, 47
  - \_base\_spe\_open\_if\_closed, 48
- create.h, 49
  - \_base\_spe\_close\_if\_open, 49
  - \_base\_spe\_open\_if\_closed, 50
- data
  - spe\_event\_unit\_t, 21
- DEBUG\_PRINTF
  - elf\_loader.c, 61
  - handler\_utils.h, 70
  - spebase.h, 130
- DECL\_0\_ARGS
  - handler\_utils.h, 70
- DECL\_1\_ARGS
  - handler\_utils.h, 70
- DECL\_2\_ARGS
  - handler\_utils.h, 70
- DECL\_3\_ARGS
  - handler\_utils.h, 71
- DECL\_4\_ARGS
  - handler\_utils.h, 71
- DECL\_5\_ARGS
  - handler\_utils.h, 71
- DECL\_6\_ARGS
  - handler\_utils.h, 71
- DECL\_RET
  - handler\_utils.h, 71
- design.txt, 51
- dma.c, 52
  - \_base\_spe\_mfcio\_get, 53
  - \_base\_spe\_mfcio\_getb, 53
  - \_base\_spe\_mfcio\_getf, 53
  - \_base\_spe\_mfcio\_put, 54
  - \_base\_spe\_mfcio\_putb, 54
  - \_base\_spe\_mfcio\_putf, 55
  - \_base\_spe\_mfcio\_tag\_status\_read, 55
  - \_base\_spe\_mssync\_start, 56
  - \_base\_spe\_mssync\_status, 57
- dma.h, 58

- mfc\_cmd, 58
- MFC\_CMD\_GET, 58
- MFC\_CMD\_GETB, 58
- MFC\_CMD\_GETF, 59
- MFC\_CMD\_PUT, 58
- MFC\_CMD\_PUTB, 58
- MFC\_CMD\_PUTF, 58
- ea
  - mfc\_command\_parameter\_area, 8
- elf\_image
  - spe\_program\_handle\_t, 29
- elf\_loader.c, 60
  - \_\_PRINTF, 61
  - \_base\_spe\_load\_spe\_elf, 61
  - \_base\_spe\_parse\_isolated\_elf, 62
  - \_base\_spe\_toe\_ear, 63
  - \_base\_spe\_verify\_spe\_elf\_image, 64
  - DEBUG\_PRINTF, 61
  - TAG, 61
- elf\_loader.h, 65
  - \_base\_spe\_load\_spe\_elf, 66
  - \_base\_spe\_parse\_isolated\_elf, 67
  - \_base\_spe\_toe\_ear, 68
  - \_base\_spe\_verify\_spe\_elf\_image, 69
  - LS\_SIZE, 66
  - SPE\_LDR\_PARAMS\_start, 66
  - SPE\_LDR\_PROG\_start, 66
- emulated\_entry
  - spe\_context\_base\_priv, 15
- entry
  - spe\_context\_base\_priv, 15
  - spe\_ld\_info, 24
- ev\_pipe
  - spe\_context\_base\_priv, 14
- event\_private
  - spe\_context, 11
  - spe\_gang\_context, 22
- events
  - spe\_context\_event\_priv\_t, 17
  - spe\_event\_unit\_t, 21
- fd\_attr, 6
  - mode, 6
  - name, 6
- fd\_gang\_dir
  - spe\_gang\_context\_base\_priv, 23
- fd\_grp\_dir
  - spe\_context\_base\_priv, 13
- FD\_IBOX
  - spebase.h, 132
- FD\_IBOX\_NB
  - spebase.h, 132
- FD\_IBOX\_STAT
  - spebase.h, 132
- fd\_lock
  - spe\_context\_base\_priv, 13
- FD\_MBOX
  - spebase.h, 132
- FD\_MBOX\_STAT
  - spebase.h, 132
- FD\_MFC
  - spebase.h, 132
- FD\_MSS
  - spebase.h, 132
- fd\_name
  - spebase.h, 132
- FD\_SIG1
  - spebase.h, 132
- FD\_SIG2
  - spebase.h, 132
- fd\_spe\_dir
  - spe\_context\_base\_priv, 13
- FD\_WBOX
  - spebase.h, 132
- FD\_WBOX\_NB
  - spebase.h, 132
- FD\_WBOX\_STAT
  - spebase.h, 132
- flags
  - spe\_context\_base\_priv, 13
  - spe\_gang\_context\_base\_priv, 23
- gangname
  - spe\_gang\_context\_base\_priv, 23
- GET\_LS\_PTR
  - handler\_utils.h, 71
- GET\_LS\_PTR\_NULL
  - handler\_utils.h, 72
- GNU\_SOURCE
  - run.c, 113
- handle
  - spe\_context, 11
- handle\_size
  - spe\_program\_handle\_t, 29
- HANDLER\_IDX
  - lib\_builtin.c, 82
- handler\_utils.h, 70
  - \_\_PRINTF, 70
  - DEBUG\_PRINTF, 70
  - DECL\_0\_ARGS, 70
  - DECL\_1\_ARGS, 70
  - DECL\_2\_ARGS, 70
  - DECL\_3\_ARGS, 71
  - DECL\_4\_ARGS, 71
  - DECL\_5\_ARGS, 71
  - DECL\_6\_ARGS, 71

- DECL\_RET, 71
- GET\_LS\_PTR, 71
- GET\_LS\_PTR\_NULL, 72
- LS\_ADDR\_MASK, 72
- LS\_ARG\_ADDR, 72
- LS\_SIZE, 72
- PUT\_LS\_RC, 72
- image.c, 73
  - \_base\_spe\_image\_close, 73
  - \_base\_spe\_image\_open, 74
- image\_handle, 7
  - map\_size, 7
  - speh, 7
- info.c, 76
  - \_base\_spe\_count\_physical\_cpus, 76
  - \_base\_spe\_count\_physical\_spes, 77
  - \_base\_spe\_count\_usable\_spes, 77
  - \_base\_spe\_cpu\_info\_get, 77
- info.h, 79
  - \_base\_spe\_count\_physical\_cpus, 79
  - \_base\_spe\_count\_physical\_spes, 80
  - \_base\_spe\_count\_usable\_spes, 81
  - THREADS\_PER\_BE, 79
- lib\_builtin.c, 82
  - \_base\_spe\_callback\_handler\_deregister, 83
  - \_base\_spe\_callback\_handler\_query, 83
  - \_base\_spe\_callback\_handler\_register, 83
  - \_base\_spe\_handle\_library\_callback, 84
  - HANDLER\_IDX, 82
- lib\_builtin.h, 86
  - \_base\_spe\_handle\_library\_callback, 87
  - MAX\_CALLNUM, 86
  - RESERVED, 86
- libspe2-types.h, 88
  - ps\_area, 96
  - SIGSPE, 90
  - SPE\_AFFINITY\_MEMORY, 90
  - SPE\_CALLBACK\_ERROR, 90
  - SPE\_CALLBACK\_NEW, 90
  - SPE\_CALLBACK\_UPDATE, 90
  - SPE\_CFG\_SIGNOTIFY1\_OR, 90
  - SPE\_CFG\_SIGNOTIFY2\_OR, 90
  - spe\_context\_ptr\_t, 95
  - SPE\_CONTROL\_AREA, 96
  - SPE\_COUNT\_PHYSICAL\_CPU\_NODES, 90
  - SPE\_COUNT\_PHYSICAL\_SPES, 90
  - SPE\_COUNT\_USABLE\_SPES, 91
  - SPE\_DEFAULT\_ENTRY, 91
  - SPE\_DMA\_ALIGNMENT, 91
  - SPE\_DMA\_SEGMENTATION, 91
  - SPE\_DMA\_STORAGE, 91
  - SPE\_EVENT\_ALL\_EVENTS, 91
  - spe\_event\_handler\_ptr\_t, 95
  - spe\_event\_handler\_t, 95
  - SPE\_EVENT\_IN\_MBOX, 91
  - SPE\_EVENT\_OUT\_INTR\_MBOX, 91
  - SPE\_EVENT\_SPE\_STOPPED, 92
  - SPE\_EVENT\_TAG\_GROUP, 92
  - SPE\_EVENTS\_ENABLE, 92
  - SPE\_EXIT, 92
  - spe\_gang\_context\_ptr\_t, 95
  - SPE\_ISOLATE, 92
  - SPE\_ISOLATE\_EMULATE, 92
  - SPE\_ISOLATION\_ERROR, 92
  - SPE\_MAP\_PS, 92
  - SPE\_MBOX\_ALL\_BLOCKING, 93
  - SPE\_MBOX\_ANY\_BLOCKING, 93
  - SPE\_MBOX\_ANY\_NONBLOCKING, 93
  - SPE\_MFC\_COMMAND\_AREA, 96
  - SPE\_MSSYNC\_AREA, 96
  - SPE\_NO\_CALLBACKS, 93
  - SPE\_RUN\_USER\_REGS, 93
  - SPE\_RUNTIME\_ERROR, 93
  - SPE\_RUNTIME\_EXCEPTION, 93
  - SPE\_RUNTIME\_FATAL, 93
  - SPE\_SIG\_NOTIFY\_1\_AREA, 96
  - SPE\_SIG\_NOTIFY\_2\_AREA, 96
  - SPE\_SIG\_NOTIFY\_REG\_1, 94
  - SPE\_SIG\_NOTIFY\_REG\_2, 94
  - SPE\_SPU\_HALT, 94
  - SPE\_SPU\_INVALID\_CHANNEL, 94
  - SPE\_SPU\_INVALID\_INSTR, 94
  - SPE\_SPU\_SINGLE\_STEP, 94
  - SPE\_SPU\_STOPPED\_BY\_STOP, 94
  - SPE\_SPU\_WAITING\_ON\_CHANNEL, 94
  - SPE\_STOP\_AND\_SIGNAL, 95
  - SPE\_TAG\_ALL, 95
  - SPE\_TAG\_ANY, 95
  - SPE\_TAG\_IMMEDIATE, 95
- libspe2.h, 97
  - spe\_callback\_handler\_deregister, 100
  - spe\_callback\_handler\_query, 100
  - spe\_callback\_handler\_register, 100
  - spe\_context\_create, 100
  - spe\_context\_create\_affinity, 100
  - spe\_context\_destroy, 100
  - spe\_context\_run, 100
  - spe\_cpu\_info\_get, 100
  - spe\_event\_handler\_create, 100
  - spe\_event\_handler\_deregister, 100
  - spe\_event\_handler\_destroy, 100
  - spe\_event\_handler\_register, 100
  - spe\_event\_wait, 100
  - spe\_gang\_context\_create, 100
  - spe\_gang\_context\_destroy, 100



- spe\_image\_close, 100
- spe\_image\_open, 100
- spe\_in\_mbox\_status, 100
- spe\_in\_mbox\_write, 100
- spe\_ls\_area\_get, 100
- spe\_ls\_size\_get, 100
- spe\_mfcio\_get, 100
- spe\_mfcio\_getb, 100
- spe\_mfcio\_getf, 100
- spe\_mfcio\_put, 100
- spe\_mfcio\_putb, 100
- spe\_mfcio\_putf, 100
- spe\_mfcio\_tag\_status\_read, 100
- spe\_mssync\_start, 100
- spe\_mssync\_status, 100
- spe\_out\_intr\_mbox\_read, 100
- spe\_out\_intr\_mbox\_status, 100
- spe\_out\_mbox\_read, 100
- spe\_out\_mbox\_status, 100
- spe\_program\_load, 100
- spe\_ps\_area\_get, 100
- spe\_signal\_write, 100
- spe\_stop\_info\_read, 100
- load.c, 101
  - \_base\_spe\_emulated\_loader\_present, 101
  - \_base\_spe\_program\_load, 102
  - \_base\_spe\_program\_load\_complete, 102
  - SPE\_EMULATED\_LOADER\_FILE, 101
- loaded\_program
  - spe\_context\_base\_priv, 15
- lock
  - spe\_context\_event\_priv\_t, 17
- LS\_ADDR\_MASK
  - handler\_utils.h, 72
- LS\_ARG\_ADDR
  - handler\_utils.h, 72
- LS\_SIZE
  - elf\_loader.h, 66
  - handler\_utils.h, 72
  - spebase.h, 130
- lsa
  - mfc\_command\_parameter\_area, 8
- map\_size
  - image\_handle, 7
- MAX\_CALLNUM
  - lib\_builtin.h, 86
- mbox.c, 104
  - \_base\_spe\_in\_mbox\_status, 104
  - \_base\_spe\_in\_mbox\_write, 105
  - \_base\_spe\_out\_intr\_mbox\_read, 106
  - \_base\_spe\_out\_intr\_mbox\_status, 107
  - \_base\_spe\_out\_mbox\_read, 108
  - \_base\_spe\_out\_mbox\_status, 109
  - \_base\_spe\_signal\_write, 109
- mbox.h, 111
- mem\_mmap\_base
  - spe\_context\_base\_priv, 14
- MFC\_ClassID\_CMD
  - spe\_mfc\_command\_area\_t, 26
- mfc\_cmd
  - dma.h, 58
- MFC\_CMD\_GET
  - dma.h, 58
- MFC\_CMD\_GETB
  - dma.h, 58
- MFC\_CMD\_GETF
  - dma.h, 59
- MFC\_CMD\_PUT
  - dma.h, 58
- MFC\_CMD\_PUTB
  - dma.h, 58
- MFC\_CMD\_PUTF
  - dma.h, 58
- MFC\_CMDStatus
  - spe\_mfc\_command\_area\_t, 26
- mfc\_command\_parameter\_area, 8
  - class, 8
  - cmd, 8
  - ea, 8
  - lsa, 8
  - pad, 8
  - size, 8
  - tag, 8
- MFC\_EAH
  - spe\_mfc\_command\_area\_t, 25
- MFC\_EAL
  - spe\_mfc\_command\_area\_t, 25
- MFC\_LSA
  - spe\_mfc\_command\_area\_t, 25
- mfc\_mmap\_base
  - spe\_context\_base\_priv, 14
- MFC\_MSSync
  - spe\_mssync\_area\_t, 28
- MFC\_OFFSET
  - spebase.h, 130
- MFC\_QStatus
  - spe\_mfc\_command\_area\_t, 26
- MFC\_SIZE
  - spebase.h, 131
- MFC\_Size\_Tag
  - spe\_mfc\_command\_area\_t, 25
- mode
  - fd\_attr, 6
- MSS\_SIZE
  - spebase.h, 131
- mssync\_mmap\_base
  - spe\_context\_base\_priv, 14

- MSSYNC\_OFFSET
  - spebase.h, [131](#)
- name
  - fd\_attr, [6](#)
- npc
  - spe\_context\_info, [18](#)
- NUM\_MBOX\_FDS
  - spebase.h, [132](#)
- pad
  - mfc\_command\_parameter\_area, [8](#)
- prev
  - spe\_context\_info, [18](#)
- Prxy\_QueryMask
  - spe\_mfc\_command\_area\_t, [26](#)
- Prxy\_QueryType
  - spe\_mfc\_command\_area\_t, [26](#)
- Prxy\_TagStatus
  - spe\_mfc\_command\_area\_t, [26](#)
- ps\_area
  - libspe2-types.h, [96](#)
- psmap\_mmap\_base
  - spe\_context\_base\_priv, [14](#)
- PSMAP\_SIZE
  - spebase.h, [131](#)
- ptr
  - spe\_event\_data\_t, [19](#)
- PUT\_LS\_RC
  - handler\_utils.h, [72](#)
- RESERVED
  - lib\_builtin.h, [86](#)
- reserved\_0\_3
  - spe\_mfc\_command\_area\_t, [25](#)
  - spe\_spu\_control\_area\_t, [33](#)
- reserved\_0\_B
  - spe\_sig\_notify\_1\_area\_t, [31](#)
  - spe\_sig\_notify\_2\_area\_t, [32](#)
- reserved\_108\_203
  - spe\_mfc\_command\_area\_t, [26](#)
- reserved\_10\_13
  - spe\_spu\_control\_area\_t, [33](#)
- reserved\_18\_103
  - spe\_mfc\_command\_area\_t, [26](#)
- reserved\_18\_1B
  - spe\_spu\_control\_area\_t, [34](#)
- reserved\_208\_21B
  - spe\_mfc\_command\_area\_t, [26](#)
- reserved\_20\_23
  - spe\_spu\_control\_area\_t, [34](#)
- reserved\_220\_22B
  - spe\_mfc\_command\_area\_t, [26](#)
- reserved\_28\_33
  - spe\_spu\_control\_area\_t, [34](#)
- reserved\_8\_B
  - spe\_spu\_control\_area\_t, [33](#)
- result
  - spe\_stop\_info\_t, [36](#)
- run.c, [112](#)
  - \_\_spe\_current\_active\_context, [117](#)
  - \_base\_spe\_context\_run, [113](#)
  - GNU\_SOURCE, [113](#)
- signal1\_mmap\_base
  - spe\_context\_base\_priv, [14](#)
- SIGNAL1\_OFFSET
  - spebase.h, [131](#)
- signal2\_mmap\_base
  - spe\_context\_base\_priv, [14](#)
- SIGNAL2\_OFFSET
  - spebase.h, [131](#)
- SIGNAL\_SIZE
  - spebase.h, [131](#)
- SIGSPE
  - libspe2-types.h, [90](#)
- size
  - mfc\_command\_parameter\_area, [8](#)
- slot
  - spe\_reg128, [30](#)
- spe
  - spe\_event\_unit\_t, [21](#)
- SPE\_AFFINITY\_MEMORY
  - libspe2-types.h, [90](#)
- SPE\_CALLBACK\_ERROR
  - libspe2-types.h, [90](#)
- spe\_callback\_error
  - spe\_stop\_info\_t, [36](#)
- spe\_callback\_handler\_deregister
  - libspe2.h, [100](#)
- spe\_callback\_handler\_query
  - libspe2.h, [100](#)
- spe\_callback\_handler\_register
  - libspe2.h, [100](#)
- SPE\_CALLBACK\_NEW
  - libspe2-types.h, [90](#)
- SPE\_CALLBACK\_UPDATE
  - libspe2-types.h, [90](#)
- SPE\_CFG\_SIGNOTIFY1\_OR
  - libspe2-types.h, [90](#)
- SPE\_CFG\_SIGNOTIFY2\_OR
  - libspe2-types.h, [90](#)
- spe\_context, [10](#)
  - base\_private, [11](#)
  - event\_private, [11](#)
  - handle, [11](#)
- spe\_context\_base\_priv, [12](#)
  - active\_tagmask, [15](#)

- cntl\_mmap\_base, [14](#)
- emulated\_entry, [15](#)
- entry, [15](#)
- ev\_pipe, [14](#)
- fd\_grp\_dir, [13](#)
- fd\_lock, [13](#)
- fd\_spe\_dir, [13](#)
- flags, [13](#)
- loaded\_program, [15](#)
- mem\_mmap\_base, [14](#)
- mfc\_mmap\_base, [14](#)
- mssync\_mmap\_base, [14](#)
- pmap\_mmap\_base, [14](#)
- signal1\_mmap\_base, [14](#)
- signal2\_mmap\_base, [14](#)
- spe\_fds\_array, [13](#)
- spe\_fds\_refcount, [13](#)
- spe\_context\_create
  - libspe2.h, [100](#)
- spe\_context\_create\_affinity
  - libspe2.h, [100](#)
- spe\_context\_destroy
  - libspe2.h, [100](#)
- spe\_context\_event\_priv\_ptr\_t
  - speevent.h, [166](#)
- spe\_context\_event\_priv\_t, [16](#)
  - events, [17](#)
  - lock, [17](#)
  - stop\_event\_pipe, [17](#)
  - stop\_event\_read\_lock, [17](#)
- spe\_context\_info, [18](#)
  - npc, [18](#)
  - prev, [18](#)
  - spe\_id, [18](#)
  - status, [18](#)
- spe\_context\_ptr\_t
  - libspe2-types.h, [95](#)
- spe\_context\_run
  - libspe2.h, [100](#)
- SPE\_CONTROL\_AREA
  - libspe2-types.h, [96](#)
- SPE\_COUNT\_PHYSICAL\_CPU\_NODES
  - libspe2-types.h, [90](#)
- SPE\_COUNT\_PHYSICAL\_SPES
  - libspe2-types.h, [90](#)
- SPE\_COUNT\_USABLE\_SPES
  - libspe2-types.h, [91](#)
- spe\_cpu\_info\_get
  - libspe2.h, [100](#)
- SPE\_DEFAULT\_ENTRY
  - libspe2-types.h, [91](#)
- SPE\_DMA\_ALIGNMENT
  - libspe2-types.h, [91](#)
- SPE\_DMA\_SEGMENTATION
  - libspe2-types.h, [91](#)
- SPE\_DMA\_STORAGE
  - libspe2-types.h, [91](#)
- SPE\_EMULATE\_PARAM\_BUFFER
  - spebase.h, [131](#)
- SPE\_EMULATED\_LOADER\_FILE
  - load.c, [101](#)
- spe\_event.c, [118](#)
  - \_\_SPE\_EPOLL\_FD\_GET, [119](#)
  - \_\_SPE\_EPOLL\_FD\_SET, [119](#)
  - \_\_SPE\_EPOLL\_SIZE, [119](#)
  - \_\_SPE\_EVENTS\_ENABLED, [120](#)
  - \_\_SPE\_EVENT\_ALL, [119](#)
  - \_\_SPE\_EVENT\_CONTEXT\_PRIV\_GET, [119](#)
  - \_\_SPE\_EVENT\_CONTEXT\_PRIV\_SET, [119](#)
  - \_event\_spe\_context\_finalize, [120](#)
  - \_event\_spe\_context\_initialize, [120](#)
  - \_event\_spe\_context\_lock, [121](#)
  - \_event\_spe\_context\_run, [121](#)
  - \_event\_spe\_context\_unlock, [121](#)
  - \_event\_spe\_event\_handler\_create, [122](#)
  - \_event\_spe\_event\_handler\_deregister, [122](#)
  - \_event\_spe\_event\_handler\_destroy, [123](#)
  - \_event\_spe\_event\_handler\_register, [124](#)
  - \_event\_spe\_event\_wait, [126](#)
  - \_event\_spe\_stop\_info\_read, [126](#)
- SPE\_EVENT\_ALL\_EVENTS
  - libspe2-types.h, [91](#)
- spe\_event\_data\_t, [19](#)
  - ptr, [19](#)
  - u32, [19](#)
  - u64, [19](#)
- spe\_event\_handler\_create
  - libspe2.h, [100](#)
- spe\_event\_handler\_deregister
  - libspe2.h, [100](#)
- spe\_event\_handler\_destroy
  - libspe2.h, [100](#)
- spe\_event\_handler\_ptr\_t
  - libspe2-types.h, [95](#)
- spe\_event\_handler\_register
  - libspe2.h, [100](#)
- spe\_event\_handler\_t
  - libspe2-types.h, [95](#)
- SPE\_EVENT\_IN\_MBOX
  - libspe2-types.h, [91](#)
- SPE\_EVENT\_OUT\_INTR\_MBOX
  - libspe2-types.h, [91](#)
- SPE\_EVENT\_SPE\_STOPPED
  - libspe2-types.h, [92](#)
- SPE\_EVENT\_TAG\_GROUP
  - libspe2-types.h, [92](#)
- spe\_event\_unit\_t, [20](#)

- data, 21
- events, 21
- spe, 21
- spe\_event\_wait
  - libspe2.h, 100
- SPE\_EVENTS\_ENABLE
  - libspe2-types.h, 92
- SPE\_EXIT
  - libspe2-types.h, 92
- spe\_exit\_code
  - spe\_stop\_info\_t, 35
- spe\_fds\_array
  - spe\_context\_base\_priv, 13
- spe\_fds\_refcount
  - spe\_context\_base\_priv, 13
- spe\_gang\_context, 22
  - base\_private, 22
  - event\_private, 22
- spe\_gang\_context\_base\_priv, 23
  - fd\_gang\_dir, 23
  - flags, 23
  - gangname, 23
- spe\_gang\_context\_create
  - libspe2.h, 100
- spe\_gang\_context\_destroy
  - libspe2.h, 100
- spe\_gang\_context\_ptr\_t
  - libspe2-types.h, 95
- spe\_id
  - spe\_context\_info, 18
- spe\_image\_close
  - libspe2.h, 100
- spe\_image\_open
  - libspe2.h, 100
- spe\_in\_mbox\_status
  - libspe2.h, 100
- spe\_in\_mbox\_write
  - libspe2.h, 100
- SPE\_ISOLATE
  - libspe2-types.h, 92
- SPE\_ISOLATE\_EMULATE
  - libspe2-types.h, 92
- SPE\_ISOLATION\_ERROR
  - libspe2-types.h, 92
- spe\_isolation\_error
  - spe\_stop\_info\_t, 36
- spe\_ld\_info, 24
  - entry, 24
- SPE\_LDR\_PARAMS\_start
  - elf\_loader.h, 66
- SPE\_LDR\_PROG\_start
  - elf\_loader.h, 66
- spe\_ls\_area\_get
  - libspe2.h, 100
- spe\_ls\_size\_get
  - libspe2.h, 100
- SPE\_MAP\_PS
  - libspe2-types.h, 92
- SPE\_MBOX\_ALL\_BLOCKING
  - libspe2-types.h, 93
- SPE\_MBOX\_ANY\_BLOCKING
  - libspe2-types.h, 93
- SPE\_MBOX\_ANY\_NONBLOCKING
  - libspe2-types.h, 93
- SPE\_MFC\_COMMAND\_AREA
  - libspe2-types.h, 96
- spe\_mfc\_command\_area\_t, 25
  - MFC\_ClassID\_CMD, 26
  - MFC\_CMDStatus, 26
  - MFC\_EAH, 25
  - MFC\_EAL, 25
  - MFC\_LSA, 25
  - MFC\_QStatus, 26
  - MFC\_Size\_Tag, 25
  - Prxy\_QueryMask, 26
  - Prxy\_QueryType, 26
  - Prxy\_TagStatus, 26
  - reserved\_0\_3, 25
  - reserved\_108\_203, 26
  - reserved\_18\_103, 26
  - reserved\_208\_21B, 26
  - reserved\_220\_22B, 26
- spe\_mfcio\_get
  - libspe2.h, 100
- spe\_mfcio\_getb
  - libspe2.h, 100
- spe\_mfcio\_getf
  - libspe2.h, 100
- spe\_mfcio\_put
  - libspe2.h, 100
- spe\_mfcio\_putb
  - libspe2.h, 100
- spe\_mfcio\_putf
  - libspe2.h, 100
- spe\_mfcio\_tag\_status\_read
  - libspe2.h, 100
- SPE\_MSSYNC\_AREA
  - libspe2-types.h, 96
- spe\_mssync\_area\_t, 28
  - MFC\_MSSync, 28
- spe\_mssync\_start
  - libspe2.h, 100
- spe\_mssync\_status
  - libspe2.h, 100
- SPE\_NO\_CALLBACKS
  - libspe2-types.h, 93
- spe\_out\_intr\_mbox\_read
  - libspe2.h, 100

- spe\_out\_intr\_mbox\_status
  - libspe2.h, 100
- spe\_out\_mbox\_read
  - libspe2.h, 100
- spe\_out\_mbox\_status
  - libspe2.h, 100
- spe\_program\_handle\_t, 29
  - elf\_image, 29
  - handle\_size, 29
  - toe\_shadow, 29
- SPE\_PROGRAM\_ISO\_LOAD\_COMPLETE
  - spebase.h, 132
- SPE\_PROGRAM\_ISOLATED\_STOP
  - spebase.h, 132
- SPE\_PROGRAM\_LIBRARY\_CALL
  - spebase.h, 132
- spe\_program\_load
  - libspe2.h, 100
- SPE\_PROGRAM\_NORMAL\_END
  - spebase.h, 132
- spe\_ps\_area\_get
  - libspe2.h, 100
- spe\_reg128, 30
  - slot, 30
- SPE\_RUN\_USER\_REGS
  - libspe2-types.h, 93
- SPE\_RUNTIME\_ERROR
  - libspe2-types.h, 93
- spe\_runtime\_error
  - spe\_stop\_info\_t, 35
- SPE\_RUNTIME\_EXCEPTION
  - libspe2-types.h, 93
- spe\_runtime\_exception
  - spe\_stop\_info\_t, 35
- SPE\_RUNTIME\_FATAL
  - libspe2-types.h, 93
- spe\_runtime\_fatal
  - spe\_stop\_info\_t, 36
- SPE\_SIG\_NOTIFY\_1\_AREA
  - libspe2-types.h, 96
- spe\_sig\_notify\_1\_area\_t, 31
  - reserved\_0\_B, 31
  - SPU\_Sig\_Notify\_1, 31
- SPE\_SIG\_NOTIFY\_2\_AREA
  - libspe2-types.h, 96
- spe\_sig\_notify\_2\_area\_t, 32
  - reserved\_0\_B, 32
  - SPU\_Sig\_Notify\_2, 32
- SPE\_SIG\_NOTIFY\_REG\_1
  - libspe2-types.h, 94
- SPE\_SIG\_NOTIFY\_REG\_2
  - libspe2-types.h, 94
- spe\_signal\_code
  - spe\_stop\_info\_t, 35
- spe\_signal\_write
  - libspe2.h, 100
- spe\_spu\_control\_area\_t, 33
  - reserved\_0\_3, 33
  - reserved\_10\_13, 33
  - reserved\_18\_1B, 34
  - reserved\_20\_23, 34
  - reserved\_28\_33, 34
  - reserved\_8\_B, 33
  - SPU\_In\_Mbox, 33
  - SPU\_Mbox\_Stat, 33
  - SPU\_NPC, 34
  - SPU\_Out\_Mbox, 33
  - SPU\_RunCntl, 34
  - SPU\_Status, 34
- SPE\_SPU\_HALT
  - libspe2-types.h, 94
- SPE\_SPU\_INVALID\_CHANNEL
  - libspe2-types.h, 94
- SPE\_SPU\_INVALID\_INSTR
  - libspe2-types.h, 94
- SPE\_SPU\_SINGLE\_STEP
  - libspe2-types.h, 94
- SPE\_SPU\_STOPPED\_BY\_STOP
  - libspe2-types.h, 94
- SPE\_SPU\_WAITING\_ON\_CHANNEL
  - libspe2-types.h, 94
- SPE\_STOP\_AND\_SIGNAL
  - libspe2-types.h, 95
- spe\_stop\_info\_read
  - libspe2.h, 100
- spe\_stop\_info\_t, 35
  - \_\_reserved\_ptr, 36
  - \_\_reserved\_u64, 36
  - result, 36
  - spe\_callback\_error, 36
  - spe\_exit\_code, 35
  - spe\_isolation\_error, 36
  - spe\_runtime\_error, 35
  - spe\_runtime\_exception, 35
  - spe\_runtime\_fatal, 36
  - spe\_signal\_code, 35
  - spu\_status, 36
  - stop\_reason, 35
- SPE\_TAG\_ALL
  - libspe2-types.h, 95
- SPE\_TAG\_ANY
  - libspe2-types.h, 95
- SPE\_TAG\_IMMEDIATE
  - libspe2-types.h, 95
- spebase.h, 128
  - \_\_PRINTF, 130
  - \_\_base\_spe\_event\_source\_acquire, 133
  - \_\_base\_spe\_event\_source\_release, 133

- [\\_\\_base\\_spe\\_spe\\_dir\\_get, 133](#)
- [\\_\\_base\\_spe\\_stop\\_event\\_source\\_get, 134](#)
- [\\_\\_base\\_spe\\_stop\\_event\\_target\\_get, 134](#)
- [\\_\\_spe\\_context\\_update\\_event, 134](#)
- [\\_base\\_spe\\_callback\\_handler\\_deregister, 134](#)
- [\\_base\\_spe\\_callback\\_handler\\_query, 135](#)
- [\\_base\\_spe\\_callback\\_handler\\_register, 135](#)
- [\\_base\\_spe\\_context\\_create, 136](#)
- [\\_base\\_spe\\_context\\_destroy, 139](#)
- [\\_base\\_spe\\_context\\_lock, 140](#)
- [\\_base\\_spe\\_context\\_run, 140](#)
- [\\_base\\_spe\\_context\\_unlock, 145](#)
- [\\_base\\_spe\\_cpu\\_info\\_get, 145](#)
- [\\_base\\_spe\\_emulated\\_loader\\_present, 146](#)
- [\\_base\\_spe\\_gang\\_context\\_create, 146](#)
- [\\_base\\_spe\\_gang\\_context\\_destroy, 147](#)
- [\\_base\\_spe\\_image\\_close, 147](#)
- [\\_base\\_spe\\_image\\_open, 148](#)
- [\\_base\\_spe\\_in\\_mbox\\_status, 150](#)
- [\\_base\\_spe\\_in\\_mbox\\_write, 150](#)
- [\\_base\\_spe\\_ls\\_area\\_get, 151](#)
- [\\_base\\_spe\\_ls\\_size\\_get, 151](#)
- [\\_base\\_spe\\_mfcio\\_get, 151](#)
- [\\_base\\_spe\\_mfcio\\_getb, 152](#)
- [\\_base\\_spe\\_mfcio\\_getf, 153](#)
- [\\_base\\_spe\\_mfcio\\_put, 153](#)
- [\\_base\\_spe\\_mfcio\\_putb, 154](#)
- [\\_base\\_spe\\_mfcio\\_putf, 154](#)
- [\\_base\\_spe\\_mfcio\\_tag\\_status\\_read, 155](#)
- [\\_base\\_spe\\_mssync\\_start, 155](#)
- [\\_base\\_spe\\_mssync\\_status, 156](#)
- [\\_base\\_spe\\_out\\_intr\\_mbox\\_read, 157](#)
- [\\_base\\_spe\\_out\\_intr\\_mbox\\_status, 158](#)
- [\\_base\\_spe\\_out\\_mbox\\_read, 159](#)
- [\\_base\\_spe\\_out\\_mbox\\_status, 160](#)
- [\\_base\\_spe\\_program\\_load, 160](#)
- [\\_base\\_spe\\_program\\_load\\_complete, 161](#)
- [\\_base\\_spe\\_ps\\_area\\_get, 162](#)
- [\\_base\\_spe\\_signal\\_write, 162](#)
- [\\_base\\_spe\\_stop\\_reason\\_get, 164](#)
- [\\_base\\_spe\\_stop\\_status\\_get, 164](#)
- [CNTL\\_OFFSET, 130](#)
- [CNTL\\_SIZE, 130](#)
- [DEBUG\\_PRINTF, 130](#)
- [FD\\_IBOX, 132](#)
- [FD\\_IBOX\\_NB, 132](#)
- [FD\\_IBOX\\_STAT, 132](#)
- [FD\\_MBOX, 132](#)
- [FD\\_MBOX\\_STAT, 132](#)
- [FD\\_MFC, 132](#)
- [FD\\_MSS, 132](#)
- [fd\\_name, 132](#)
- [FD\\_SIG1, 132](#)
- [FD\\_SIG2, 132](#)
- [FD\\_WBOX, 132](#)
- [FD\\_WBOX\\_NB, 132](#)
- [FD\\_WBOX\\_STAT, 132](#)
- [LS\\_SIZE, 130](#)
- [MFC\\_OFFSET, 130](#)
- [MFC\\_SIZE, 131](#)
- [MSS\\_SIZE, 131](#)
- [MSSYNC\\_OFFSET, 131](#)
- [NUM\\_MBOX\\_FDS, 132](#)
- [PSMAP\\_SIZE, 131](#)
- [SIGNAL1\\_OFFSET, 131](#)
- [SIGNAL2\\_OFFSET, 131](#)
- [SIGNAL\\_SIZE, 131](#)
- [SPE\\_EMULATE\\_PARAM\\_BUFFER, 131](#)
- [SPE\\_PROGRAM\\_ISO\\_LOAD\\_COMPLETE, 132](#)
- [SPE\\_PROGRAM\\_ISOLATED\\_STOP, 132](#)
- [SPE\\_PROGRAM\\_LIBRARY\\_CALL, 132](#)
- [SPE\\_PROGRAM\\_NORMAL\\_END, 132](#)
- [speevent.h, 165](#)
  - [\\_\\_NUM\\_SPE\\_EVENT\\_TYPES, 166](#)
  - [\\_\\_SPE\\_EVENT\\_IN\\_MBOX, 166](#)
  - [\\_\\_SPE\\_EVENT\\_OUT\\_INTR\\_MBOX, 166](#)
  - [\\_\\_SPE\\_EVENT\\_SPE\\_STOPPED, 166](#)
  - [\\_\\_SPE\\_EVENT\\_TAG\\_GROUP, 166](#)
  - [\\_\\_spe\\_event\\_types, 166](#)
  - [\\_event\\_spe\\_context\\_finalize, 166](#)
  - [\\_event\\_spe\\_context\\_initialize, 167](#)
  - [\\_event\\_spe\\_context\\_lock, 167](#)
  - [\\_event\\_spe\\_context\\_run, 168](#)
  - [\\_event\\_spe\\_context\\_unlock, 168](#)
  - [\\_event\\_spe\\_event\\_handler\\_create, 168](#)
  - [\\_event\\_spe\\_event\\_handler\\_deregister, 169](#)
  - [\\_event\\_spe\\_event\\_handler\\_destroy, 170](#)
  - [\\_event\\_spe\\_event\\_handler\\_register, 171](#)
  - [\\_event\\_spe\\_event\\_wait, 173](#)
  - [\\_event\\_spe\\_stop\\_info\\_read, 174](#)
  - [spe\\_context\\_event\\_priv\\_ptr\\_t, 166](#)
- [speh](#)
  - [image\\_handle, 7](#)
- [SPU\\_In\\_Mbox](#)
  - [spe\\_spu\\_control\\_area\\_t, 33](#)
- [SPU\\_Mbox\\_Stat](#)
  - [spe\\_spu\\_control\\_area\\_t, 33](#)
- [SPU\\_NPC](#)
  - [spe\\_spu\\_control\\_area\\_t, 34](#)
- [SPU\\_Out\\_Mbox](#)
  - [spe\\_spu\\_control\\_area\\_t, 33](#)
- [SPU\\_RunCntl](#)
  - [spe\\_spu\\_control\\_area\\_t, 34](#)
- [SPU\\_Sig\\_Notify\\_1](#)
  - [spe\\_sig\\_notify\\_1\\_area\\_t, 31](#)
- [SPU\\_Sig\\_Notify\\_2](#)
  - [spe\\_sig\\_notify\\_2\\_area\\_t, 32](#)

---

SPU\_Status  
    spe\_spu\_control\_area\_t, [34](#)  
spu\_status  
    spe\_stop\_info\_t, [36](#)  
status  
    spe\_context\_info, [18](#)  
stop\_event\_pipe  
    spe\_context\_event\_priv\_t, [17](#)  
stop\_event\_read\_lock  
    spe\_context\_event\_priv\_t, [17](#)  
stop\_reason  
    spe\_stop\_info\_t, [35](#)  
  
TAG  
    elf\_loader.c, [61](#)  
tag  
    mfc\_command\_parameter\_area, [8](#)  
THREADS\_PER\_BE  
    info.h, [79](#)  
toe\_shadow  
    spe\_program\_handle\_t, [29](#)  
  
u32  
    spe\_event\_data\_t, [19](#)  
u64  
    spe\_event\_data\_t, [19](#)  
ui  
    addr64, [5](#)  
ull  
    addr64, [5](#)