SPE Runtime Management Library

Version 2.0

September 29, 2007

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

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

June 15, 2006

Contents

Chapter 1

Overview

The libspe2 functionality is split into 4 libraries:

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

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

1.1 Terminology

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

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

1.2 Usage Scenarios

1.2.1 Single-threaded sample

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

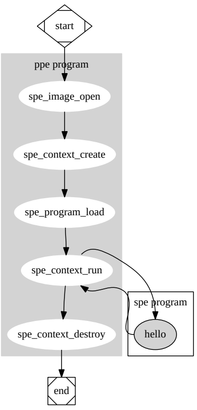


Figure 1.1: Simple program

Here is the same sample with some error checking:

1.2.2 Multi-threaded sample

This illustrates a threaded sample using the pthread library:

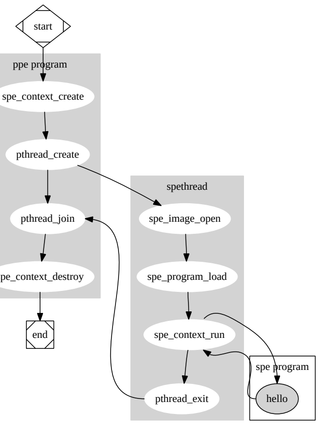


Figure 1.2: Simple pthread program

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

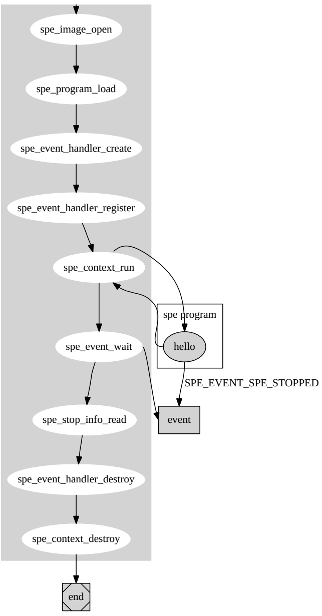


Figure 1.3: Simple event program

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.

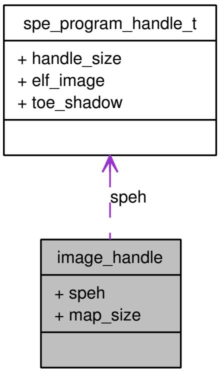
Referenced by _base_spe_open_if_closed().

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.

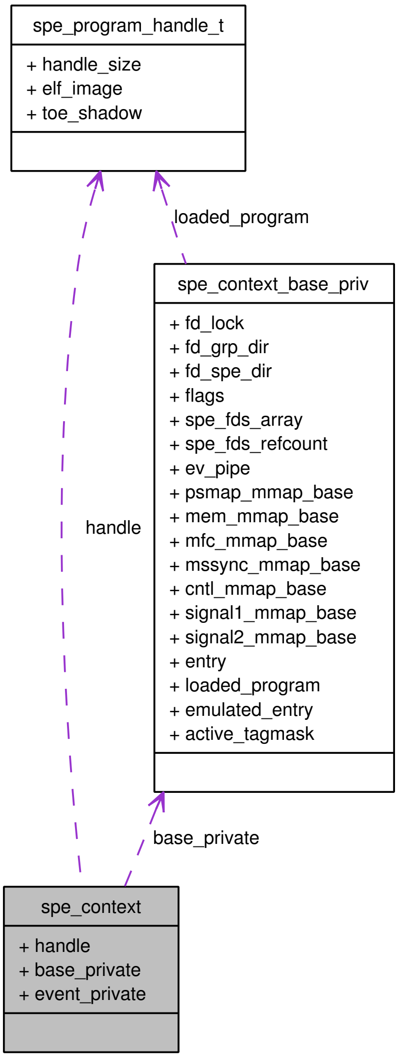
The documentation for this struct was generated from the following file:

- 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.

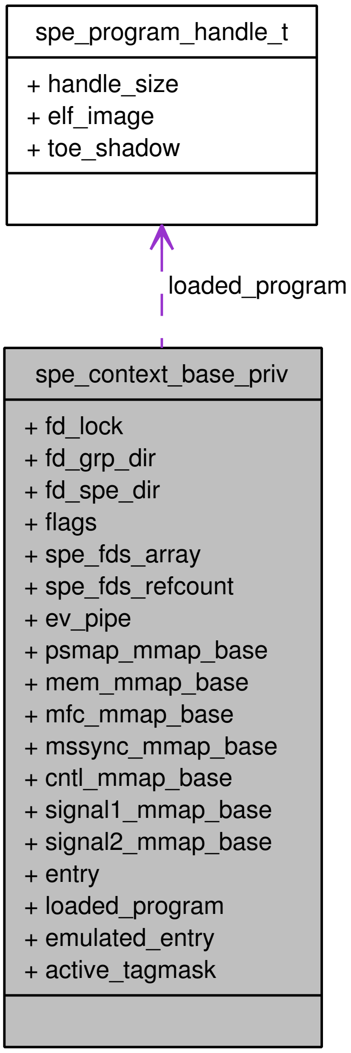
The documentation for this struct was generated from the following file:

- 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∗ psmap_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().

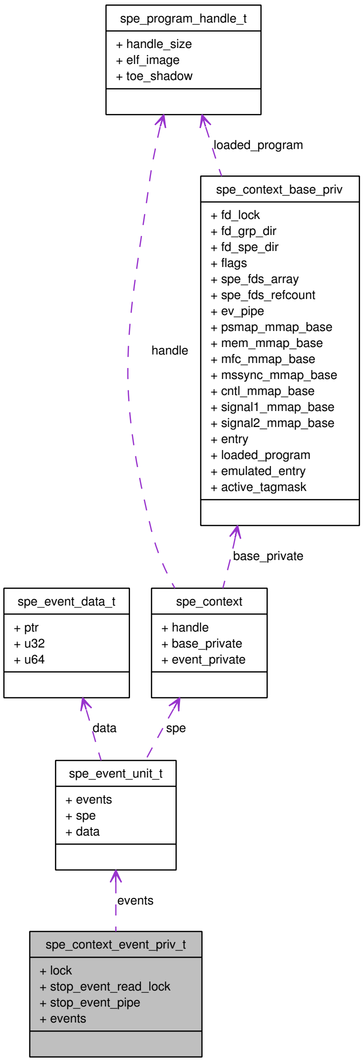
The documentation for this struct was generated from the following file:

- 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.

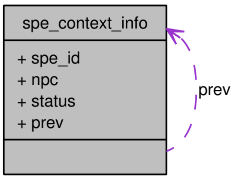
Referenced by _event_spe_context_initialize(), _event_spe_event_handler_deregister(), and _event_spe_-event_handler_register().

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.

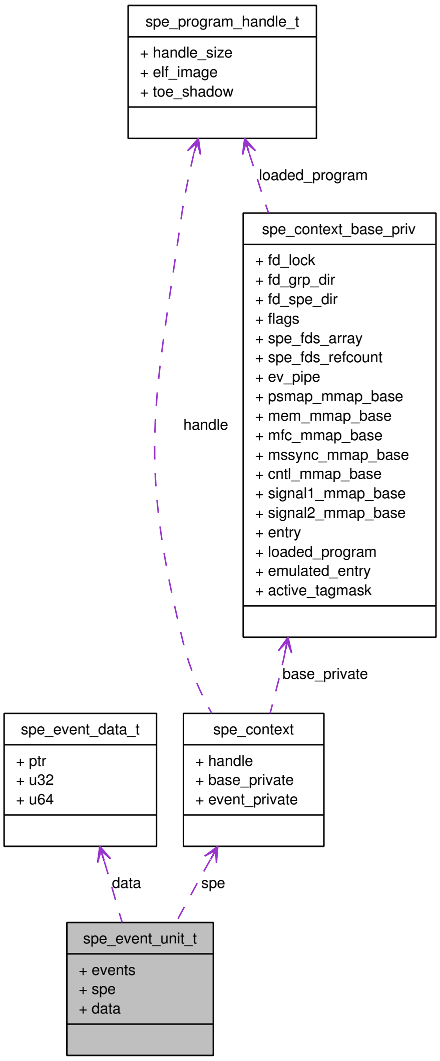
The documentation for this union was generated from the following file:

- 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().

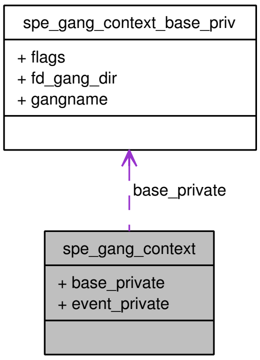
The documentation for this struct was generated from the following file:

- 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 persistant 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
  } result
- 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

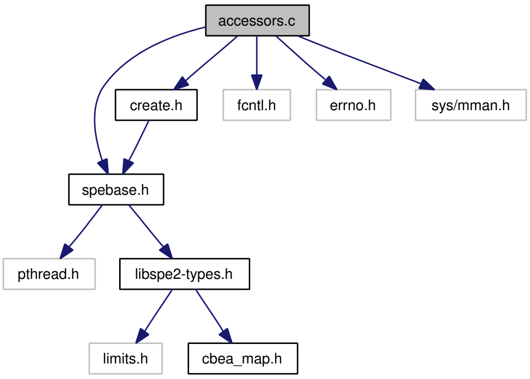
Chapter 3

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.

```
{
        return;
}
```

3.1.1.2 int __base_spe_event_source_acquire (spe_context_ptr_t *spe*, enum fd_name *fdesc*)

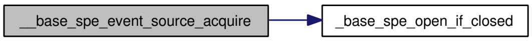
Definition at line 74 of file accessors.c.

References _base_spe_open_if_closed().

Referenced by _event_spe_event_handler_deregister(), and _event_spe_event_handler_register().

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

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.

```
{
        _base_spe_close_if_open(spe, fdesc);
}
```

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.

```
{
        return spe->base_private->fd_spe_dir;
}
```

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*)

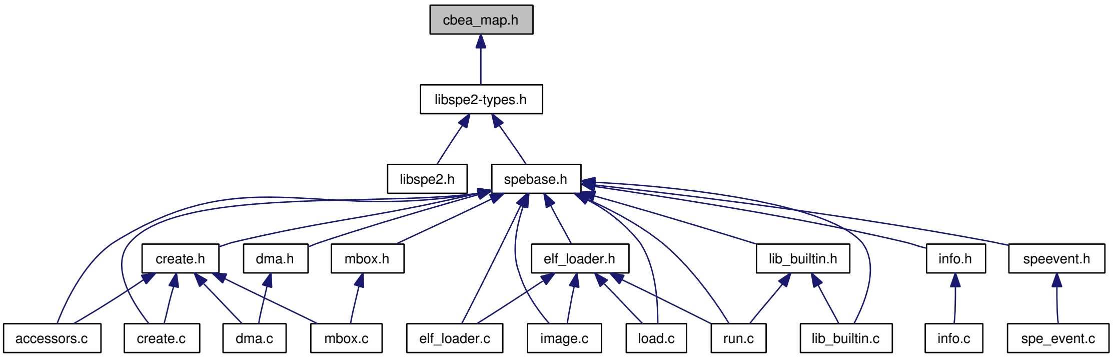
Definition at line 30 of file accessors.c.

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.

```
{
        void *ptr;

        switch (area) {
                case SPE_MSSYNC_AREA:
                        ptr = spe->base_private->mssync_mmap_base;
                        break;
                case SPE_MFC_COMMAND_AREA:
                        ptr = spe->base_private->mfc_mmap_base;
                        break;
                case SPE_CONTROL_AREA:
                        ptr = spe->base_private->cntl_mmap_base;
                        break;
                case SPE_SIG_NOTIFY_1_AREA:
                        ptr = spe->base_private->signal1_mmap_base;
                        break;
                case SPE_SIG_NOTIFY_2_AREA:
                        ptr = spe->base_private->signal2_mmap_base;
                        break;
                default:
                        errno = EINVAL;
                        return NULL;
                        break;
        }

        if (ptr == MAP_FAILED) {
                errno = EACCES;
                return NULL;
        }

        return ptr;
}
```

3.2 cbea_map.h File Reference

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



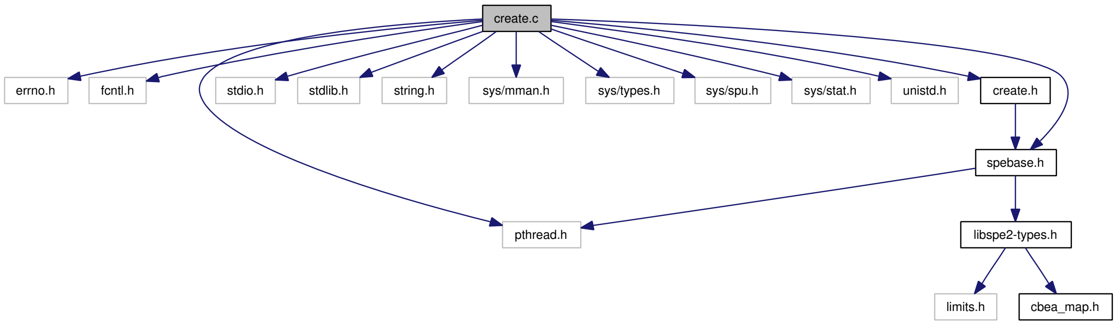
Data Structures

- 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().

```
{
        _base_spe_context_lock(spe, fdesc);

        if (spe->base_private->spe_fds_array[(int)fdesc] != -1 &&
                spe->base_private->spe_fds_refcount[(int)fdesc] == 1) {

                spe->base_private->spe_fds_refcount[(int)fdesc]--;
                close(spe->base_private->spe_fds_array[(int)fdesc]);

                spe->base_private->spe_fds_array[(int)fdesc] = -1;
        } else if (spe->base_private->spe_fds_refcount[(int)fdesc] > 0) {
                spe->base_private->spe_fds_refcount[(int)fdesc]--;
        }

        _base_spe_context_unlock(spe, fdesc);
}
```

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.

```
{
        char pathname[256];
        int i, aff_spe_fd = 0;
        unsigned int spu_createflags = 0;
        struct spe_context *spe = NULL;
        struct spe_context_base_priv *priv;

        /* We need a loader present to run in emulated isolated mode */
        if (flags & SPE_ISOLATE_EMULATE
                        && !_base_spe_emulated_loader_present()) {
                errno = EINVAL;
                return NULL;
        }

        /* Put some sane defaults into the SPE context */
        spe = malloc(sizeof(*spe));
        if (!spe) {
                DEBUG_PRINTF("ERROR: Could not allocate spe context.\n");
                return NULL;
        }
        memset(spe, 0, sizeof(*spe));

        spe->base_private = malloc(sizeof(*spe->base_private));
        if (!spe->base_private) {
                DEBUG_PRINTF("ERROR: Could not allocate "
                                "spe->base_private context.\n");
                free(spe);
                return NULL;
        }

        /* just a convenience variable */
        priv = spe->base_private;

        priv->fd_spe_dir = -1;
        priv->mem_mmap_base = MAP_FAILED;
        priv->psmap_mmap_base = MAP_FAILED;
        priv->mssync_mmap_base = MAP_FAILED;
        priv->mfc_mmap_base = MAP_FAILED;
        priv->cntl_mmap_base = MAP_FAILED;
        priv->signal1_mmap_base = MAP_FAILED;
        priv->signal2_mmap_base = MAP_FAILED;
        priv->loaded_program = NULL;

        for (i = 0; i < NUM_MBOX_FDS; i++) {
                priv->spe_fds_array[i] = -1;
                pthread_mutex_init(&priv->fd_lock[i], NULL);
        }

        /* initialise spu_createflags */
        if (flags & SPE_ISOLATE) {
                flags |= SPE_MAP_PS;
                spu_createflags |= SPU_CREATE_ISOLATE | SPU_CREATE_NOSCHED;
        }

        if (flags & SPE_EVENTS_ENABLE)
                spu_createflags |= SPU_CREATE_EVENTS_ENABLED;

        if (aff_spe)
                spu_createflags |= SPU_CREATE_AFFINITY_SPU;

        if (flags & SPE_AFFINITY_MEMORY)
                spu_createflags |= SPU_CREATE_AFFINITY_MEM;

        /* Make the SPUFS directory for the SPE */
        if (gctx == NULL)
                sprintf(pathname, "/spu/spethread-%i-%lu",
                        getpid(), (unsigned long)spe);
        else
                sprintf(pathname, "/spu/%s/spethread-%i-%lu",
                        gctx->base_private->gangname, getpid(),
                        (unsigned long)spe);

        if (aff_spe)
                aff_spe_fd = aff_spe->base_private->fd_spe_dir;

        priv->fd_spe_dir = spu_create(pathname, spu_createflags,
                        S_IRUSR | S_IWUSR | S_IXUSR, aff_spe_fd);

        if (priv->fd_spe_dir < 0) {
                DEBUG_PRINTF("ERROR: Could not create SPE %s\n", pathname);
                perror("spu_create()");
                free_spe_context(spe);
                /* we mask most errors, but leave ENODEV */
                if (errno != ENODEV)
                        errno = EFAULT;
                return NULL;
        }

        priv->flags = flags;

        /* Map the required areas into process memory */
        priv->mem_mmap_base = mapfileat(priv->fd_spe_dir, "mem", LS_SIZE);
        if (priv->mem_mmap_base == MAP_FAILED) {
                DEBUG_PRINTF("ERROR: Could not map SPE memory.\n");
                free_spe_context(spe);
                errno = ENOMEM;
                return NULL;
        }

        if (flags & SPE_MAP_PS) {
                /* It's possible to map the entire problem state area with
                 * one mmap - try this first */
                priv->psmap_mmap_base =  mapfileat(priv->fd_spe_dir,
                                "psmap", PSMAP_SIZE);

                if (priv->psmap_mmap_base != MAP_FAILED) {
                        priv->mssync_mmap_base =
                                priv->psmap_mmap_base + MSSYNC_OFFSET;
                        priv->mfc_mmap_base =
                                priv->psmap_mmap_base + MFC_OFFSET;
                        priv->cntl_mmap_base =
                                priv->psmap_mmap_base + CNTL_OFFSET;
                        priv->signal1_mmap_base =
                                priv->psmap_mmap_base + SIGNAL1_OFFSET;
                        priv->signal2_mmap_base =
                                priv->psmap_mmap_base + SIGNAL2_OFFSET;

                } else {
                        /* map each region separately */
                        priv->mfc_mmap_base =
                                mapfileat(priv->fd_spe_dir, "mfc", MFC_SIZE);
                        priv->mssync_mmap_base =
                                mapfileat(priv->fd_spe_dir, "mss", MSS_SIZE);
                        priv->cntl_mmap_base =
                                mapfileat(priv->fd_spe_dir, "cntl", CNTL_SIZE);
                        priv->signal1_mmap_base =
                                mapfileat(priv->fd_spe_dir, "signal1",
                                                SIGNAL_SIZE);
                        priv->signal2_mmap_base =
                                mapfileat(priv->fd_spe_dir, "signal2",
                                                SIGNAL_SIZE);

                        if (priv->mfc_mmap_base == MAP_FAILED ||
                                        priv->cntl_mmap_base == MAP_FAILED ||
                                        priv->signal1_mmap_base == MAP_FAILED ||
                                        priv->signal2_mmap_base == MAP_FAILED) {
                                DEBUG_PRINTF("ERROR: Could not map SPE "
                                                "PS memory.\n");
                                free_spe_context(spe);
                                errno = ENOMEM;
                                return NULL;
                        }
                }
        }

        if (flags & SPE_CFG_SIGNOTIFY1_OR) {
                if (setsignotify(priv->fd_spe_dir, "signal1_type")) {
                        DEBUG_PRINTF("ERROR: Could not open SPE "
                                        "signal1_type file.\n");
                        free_spe_context(spe);
                        errno = EFAULT;
                        return NULL;
                }
        }

```

```
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().

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

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.

```
{
        char pathname[256];
        struct spe_gang_context_base_priv *pgctx = NULL;
        struct spe_gang_context *gctx = NULL;

        gctx = malloc(sizeof(*gctx));
        if (!gctx) {
                DEBUG_PRINTF("ERROR: Could not allocate spe context.\n");
                return NULL;
        }
        memset(gctx, 0, sizeof(*gctx));

        pgctx = malloc(sizeof(*pgctx));
        if (!pgctx) {
                DEBUG_PRINTF("ERROR: Could not allocate spe context.\n");
                free(gctx);
                return NULL;
        }
        memset(pgctx, 0, sizeof(*pgctx));

        gctx->base_private = pgctx;

        sprintf(gctx->base_private->gangname, "gang-%i-%lu", getpid(),
                        (unsigned long)gctx);
        sprintf(pathname, "/spu/%s", gctx->base_private->gangname);

        gctx->base_private->fd_gang_dir = spu_create(pathname, SPU_CREATE_GANG,
                                S_IRUSR | S_IWUSR | S_IXUSR);

        if (gctx->base_private->fd_gang_dir < 0) {
                DEBUG_PRINTF("ERROR: Could not create Gang %s\n", pathname);
                free_spe_gang_context(gctx);
                errno = EFAULT;
                return NULL;
        }

        gctx->base_private->flags = flags;

        return gctx;
}
```

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.

```
{
        return free_spe_gang_context(gctx);
}
```

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.

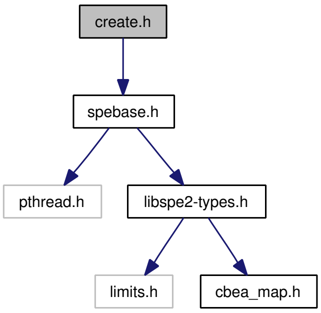
Referenced by __base_spe_event_source_acquire(), _base_spe_in_mbox_status(), _base_spe_in_mbox_write(), _base_spe_mssync_start(), _base_spe_mssync_status(), _base_spe_out_intr_mbox_read(), _base_spe_out_intr_mbox_status(), _base_spe_out_mbox_read(), _base_spe_out_mbox_status(), and _base_spe_signal_write().

```
{
        if (!locked)
                _base_spe_context_lock(spe, fdesc);

        /* already open? */
        if (spe->base_private->spe_fds_array[fdesc] != -1) {
                spe->base_private->spe_fds_refcount[fdesc]++;
        } else {
                spe->base_private->spe_fds_array[fdesc] =
                        openat(spe->base_private->fd_spe_dir,
                                        spe_fd_attr[fdesc].name,
                                        spe_fd_attr[fdesc].mode);

                if (spe->base_private->spe_fds_array[(int)fdesc] > 0)
                        spe->base_private->spe_fds_refcount[(int)fdesc]++;
        }

        if (!locked)
                _base_spe_context_unlock(spe, fdesc);

        return spe->base_private->spe_fds_array[(int)fdesc];
}
```

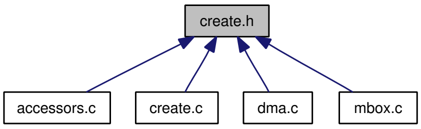
3.4 create.h File Reference

```
#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

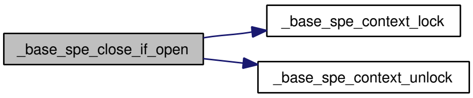
3.4.1.1 void _base_spe_close_if_open (struct spe_context ∗ *spe*, enum fd_name *fdesc*)

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.

```
{
        _base_spe_context_lock(spe, fdesc);

        if (spe->base_private->spe_fds_array[(int)fdesc] != -1 &&
                spe->base_private->spe_fds_refcount[(int)fdesc] == 1) {

                spe->base_private->spe_fds_refcount[(int)fdesc]--;
                close(spe->base_private->spe_fds_array[(int)fdesc]);

                spe->base_private->spe_fds_array[(int)fdesc] = -1;
        } else if (spe->base_private->spe_fds_refcount[(int)fdesc] > 0) {
                spe->base_private->spe_fds_refcount[(int)fdesc]--;
        }

        _base_spe_context_unlock(spe, fdesc);
}
```

Here is the call graph for this function:



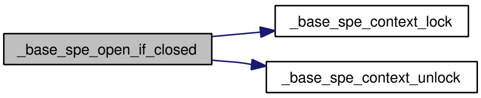
3.4.1.2 int _base_spe_open_if_closed (struct spe_context ∗ *spe*, enum fd_name *fdesc*, int *locked*)

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.

```
{
        if (!locked)
                _base_spe_context_lock(spe, fdesc);

        /* already open? */
        if (spe->base_private->spe_fds_array[fdesc] != -1) {
                spe->base_private->spe_fds_refcount[fdesc]++;
        } else {
                spe->base_private->spe_fds_array[fdesc] =
                        openat(spe->base_private->fd_spe_dir,
                                        spe_fd_attr[fdesc].name,
                                        spe_fd_attr[fdesc].mode);

                if (spe->base_private->spe_fds_array[(int)fdesc] > 0)
                        spe->base_private->spe_fds_refcount[(int)fdesc]++;
        }

        if (!locked)
                _base_spe_context_unlock(spe, fdesc);

        return spe->base_private->spe_fds_array[(int)fdesc];
}
```

Here is the call graph for this function:

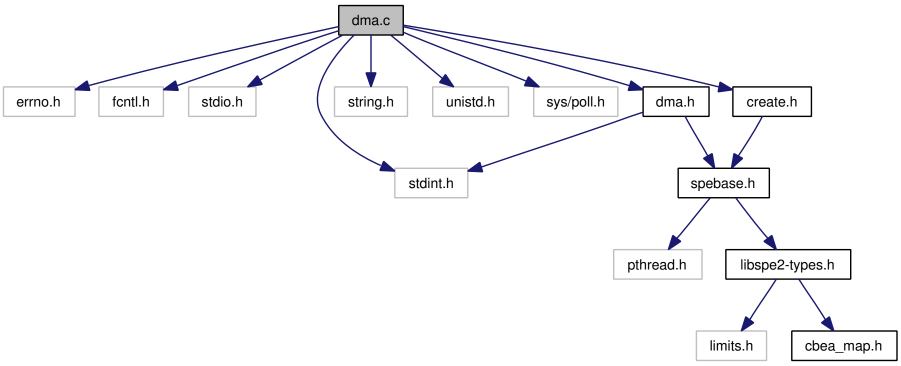


3.5 design.txt File Reference

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.

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

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.

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

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.

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

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.

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

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 i ssued 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.

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

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.

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

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.

```
{
        if ( mask != 0 ) {
                if (!(spectx->base_private->flags & SPE_MAP_PS))
                        mask = 0;
        } else {
                if ((spectx->base_private->flags & SPE_MAP_PS))
                        mask = spectx->base_private->active_tagmask;
        }

        if (!tag_status) {
                errno = EINVAL;
                return -1;
        }

        switch (behavior) {
        case SPE_TAG_ALL:
                return spe_mfcio_tag_status_read_all(spectx, mask, tag_status);
        case SPE_TAG_ANY:
                return spe_mfcio_tag_status_read_any(spectx, mask, tag_status);
        case SPE_TAG_IMMEDIATE:
                return spe_mfcio_tag_status_read_immediate(spectx, mask, tag_status);
        default:
                errno = EINVAL;
                return -1;
        }
}
```

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.

```
{
        int ret, fd;
        unsigned int data = 1; /* Any value can be written here */

        volatile struct spe_mssync_area *mss_area =
                spectx->base_private->mssync_mmap_base;

        if (spectx->base_private->flags & SPE_MAP_PS) {
                mss_area->MFC_MSSync = data;
                return 0;
        } else {
                fd = _base_spe_open_if_closed(spectx, FD_MSS, 0);
                if (fd != -1) {
                        ret = write(fd, &data, sizeof (data));
                        if ((ret < 0) && (errno != EIO)) {
                                perror("spe_mssync_start: internal error");
                        }
                        return ret < 0 ? -1 : 0;
                } else
                        return -1;
        }
}
```

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

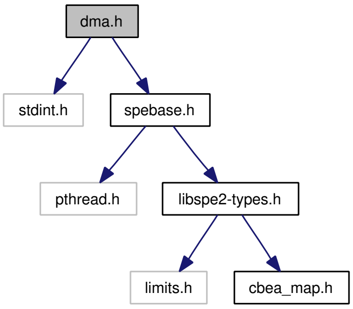
Definition at line 359 of file dma.c.

```
{
        int ret, fd;
        unsigned int data;

        volatile struct spe_mssync_area *mss_area =
                spectx->base_private->mssync_mmap_base;

        if (spectx->base_private->flags & SPE_MAP_PS) {
                return  mss_area->MFC_MSSync;
        } else {
                fd = _base_spe_open_if_closed(spectx, FD_MSS, 0);
                if (fd != -1) {
                        ret = read(fd, &data, sizeof (data));
                        if ((ret < 0) && (errno != EIO)) {
                                perror("spe_mssync_start: internal error");
                        }
                        return ret < 0 ? -1 : data;
                } else
                        return -1;
        }
}
```

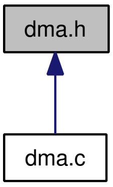
3.7 dma.h File Reference

```
#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

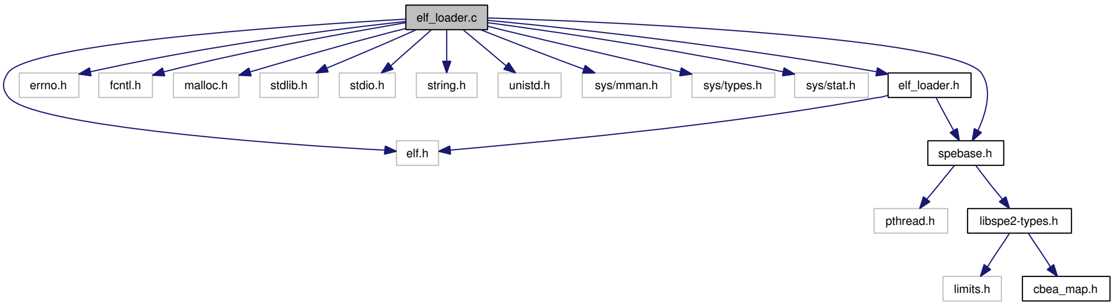
Definition at line 37 of file dma.h.

```
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().

```
{
        Elf32_Ehdr *ehdr;
        Elf32_Phdr *phdr;
        Elf32_Phdr *ph, *prev_ph;

        Elf32_Shdr *shdr;
        Elf32_Shdr *sh;

        Elf32_Off  toe_addr = 0;
        long    toe_size = 0;

        char* str_table = 0;

        int num_load_seg = 0;
        void *elf_start;
        int ret;

        DEBUG_PRINTF ("load_spe_elf(%p, %p)\n", handle, ld_buffer);

        elf_start = handle->elf_image;

        DEBUG_PRINTF ("load_spe_elf(%p, %p)\n", handle->elf_image, ld_buffer);
        ehdr = (Elf32_Ehdr *)(handle->elf_image);

        /* Check for a Valid SPE ELF Image (again) */
        if ((ret=check_spe_elf(ehdr)))
                return ret;

        /* Start processing headers */
        phdr = (Elf32_Phdr *) ((char *) ehdr + ehdr->e_phoff);
        shdr = (Elf32_Shdr *) ((char *) ehdr + ehdr->e_shoff);
        str_table = (char*)ehdr + shdr[ehdr->e_shstrndx].sh_offset;

        /* traverse the sections to locate the toe segment */
        /* by specification, the toe sections are grouped together in a segment */
        for (sh = shdr; sh < &shdr[ehdr->e_shnum]; ++sh)
        {
                DEBUG_PRINTF("section name: %s ( start: 0x%04x, size: 0x%04x)\n", str_table+sh->sh_nam
                if (strcmp(".toe", str_table+sh->sh_name) == 0) {
                        DEBUG_PRINTF("section offset: %d\n", sh->sh_offset);
                        toe_size += sh->sh_size;
                        if ((toe_addr == 0) || (toe_addr > sh->sh_addr))
                                toe_addr = sh->sh_addr;
                }
                /* Disabled : Actually not needed, only good for testing
                if (strcmp(".bss", str_table+sh->sh_name) == 0) {
                        DEBUG_PRINTF("zeroing .bss section:\n");
                        DEBUG_PRINTF("section offset: 0x%04x\n", sh->sh_offset);
                        DEBUG_PRINTF("section size: 0x%04x\n", sh->sh_size);
                        memset(ld_buffer + sh->sh_offset, 0, sh->sh_size);
                }  */

#ifdef DEBUG
                if (strcmp(".note.spu_name", str_table+sh->sh_name) == 0)
                        display_debug_output(elf_start, sh);
#endif /*DEBUG*/
        }

        /*
         * Load all PT_LOAD segments onto the SPE local store buffer.
         */
        DEBUG_PRINTF("Segments: 0x%x\n", ehdr->e_phnum);
        for (ph = phdr, prev_ph = NULL; ph < &phdr[ehdr->e_phnum]; ++ph) {
                switch (ph->p_type) {
                case PT_LOAD:
                        if (!overlay(ph, prev_ph)) {
                                if (ph->p_filesz < ph->p_memsz) {
                                        DEBUG_PRINTF("padding loaded image with zeros:\n");
                                        DEBUG_PRINTF("start: 0x%04x\n", ph->p_vaddr + ph->p_filesz);
                                        DEBUG_PRINTF("length: 0x%04x\n", ph->p_memsz - ph->p_filesz);
                                        memset(ld_buffer + ph->p_vaddr + ph->p_filesz, 0, ph->p_memsz
                                }
                                copy_to_ld_buffer(handle, ld_buffer, ph,
                                                  toe_addr, toe_size);
                                num_load_seg++;
                        }
                        break;
                case PT_NOTE:
                        DEBUG_PRINTF("SPE_LOAD found PT_NOTE\n");
                        break;
                }
        }
        if (num_load_seg == 0)
          {
                  DEBUG_PRINTF ("no segments to load");
                  errno = EINVAL;
                  return -errno;
          }

        /* Remember where the code wants to be started */
        ld_info->entry = ehdr->e_entry;
        DEBUG_PRINTF ("entry = 0x%x\n", ehdr->e_entry);

        return 0;

}
```

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.

```
{
        Elf32_Ehdr *ehdr = (Elf32_Ehdr *)handle->elf_image;
        Elf32_Phdr *phdr;

        if (!ehdr) {
                DEBUG_PRINTF("No ELF image has been loaded\n");
                errno = EINVAL;
                return -errno;
        }

        if (ehdr->e_phentsize != sizeof(*phdr)) {
                DEBUG_PRINTF("Invalid program header format (phdr size=%d)\n",
                                ehdr->e_phentsize);
                errno = EINVAL;
                return -errno;
        }

        if (ehdr->e_phnum != 1) {
                DEBUG_PRINTF("Invalid program header count (%d), expected 1\n",
                                ehdr->e_phnum);
                errno = EINVAL;
                return -errno;
        }

        phdr = (Elf32_Phdr *)(handle->elf_image + ehdr->e_phoff);

        if (phdr->p_type != PT_LOAD || phdr->p_memsz == 0) {
                DEBUG_PRINTF("SPE program segment is not loadable (type=%x)\n",
                                phdr->p_type);
                errno = EINVAL;
                return -errno;
        }

        if (addr)
                *addr = (uint64_t)(unsigned long)
                        (handle->elf_image + phdr->p_offset);

        if (size)
                *size = phdr->p_memsz;

        return 0;
}
```

3.8.2.3 int _base_spe_toe_ear (spe_program_handle_t ∗ *speh*)

Definition at line 354 of file elf_loader.c.

Referenced by _base_spe_image_open().

```
{
        Elf32_Ehdr *ehdr;
        Elf32_Shdr *shdr, *sh;
        char *str_table;
        char **ch;
        int ret;
        long toe_size;

        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.

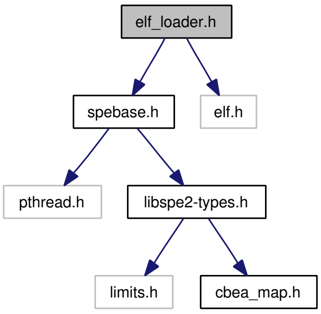
Referenced by _base_spe_emulated_loader_present(), and _base_spe_image_open().

```
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 }
```

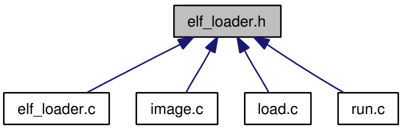
3.9 elf_loader.h File Reference

```
#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.

```
{
        Elf32_Ehdr *ehdr;
        Elf32_Phdr *phdr;
        Elf32_Phdr *ph, *prev_ph;

        Elf32_Shdr *shdr;
        Elf32_Shdr *sh;

        Elf32_Off  toe_addr = 0;
        long    toe_size = 0;

        char* str_table = 0;

        int num_load_seg = 0;
        void *elf_start;
        int ret;

        DEBUG_PRINTF ("load_spe_elf(%p, %p)\n", handle, ld_buffer);

        elf_start = handle->elf_image;

        DEBUG_PRINTF ("load_spe_elf(%p, %p)\n", handle->elf_image, ld_buffer);
        ehdr = (Elf32_Ehdr *)(handle->elf_image);

        /* Check for a Valid SPE ELF Image (again) */
        if ((ret=check_spe_elf(ehdr)))
                return ret;

        /* Start processing headers */
        phdr = (Elf32_Phdr *) ((char *) ehdr + ehdr->e_phoff);
        shdr = (Elf32_Shdr *) ((char *) ehdr + ehdr->e_shoff);
        str_table = (char*)ehdr + shdr[ehdr->e_shstrndx].sh_offset;

        /* traverse the sections to locate the toe segment */
        /* by specification, the toe sections are grouped together in a segment */
        for (sh = shdr; sh < &shdr[ehdr->e_shnum]; ++sh)
        {
                DEBUG_PRINTF("section name: %s ( start: 0x%04x, size: 0x%04x)\n", str_table+sh->sh_nam
                if (strcmp(".toe", str_table+sh->sh_name) == 0) {
                        DEBUG_PRINTF("section offset: %d\n", sh->sh_offset);
                        toe_size += sh->sh_size;
                        if ((toe_addr == 0) || (toe_addr > sh->sh_addr))
                                toe_addr = sh->sh_addr;
                }
                /* Disabled : Actually not needed, only good for testing
                if (strcmp(".bss", str_table+sh->sh_name) == 0) {
                        DEBUG_PRINTF("zeroing .bss section:\n");
                        DEBUG_PRINTF("section offset: 0x%04x\n", sh->sh_offset);
                        DEBUG_PRINTF("section size: 0x%04x\n", sh->sh_size);
                        memset(ld_buffer + sh->sh_offset, 0, sh->sh_size);
                }  */

#ifdef DEBUG
                if (strcmp(".note.spu_name", str_table+sh->sh_name) == 0)
                        display_debug_output(elf_start, sh);
#endif /*DEBUG*/
        }

        /*
         * Load all PT_LOAD segments onto the SPE local store buffer.
         */
        DEBUG_PRINTF("Segments: 0x%x\n", ehdr->e_phnum);
        for (ph = phdr, prev_ph = NULL; ph < &phdr[ehdr->e_phnum]; ++ph) {
                switch (ph->p_type) {
                case PT_LOAD:
                        if (!overlay(ph, prev_ph)) {
                                if (ph->p_filesz < ph->p_memsz) {
                                        DEBUG_PRINTF("padding loaded image with zeros:\n");
                                        DEBUG_PRINTF("start: 0x%04x\n", ph->p_vaddr + ph->p_filesz);
                                        DEBUG_PRINTF("length: 0x%04x\n", ph->p_memsz - ph->p_filesz);
                                        memset(ld_buffer + ph->p_vaddr + ph->p_filesz, 0, ph->p_memsz
                                }
                                copy_to_ld_buffer(handle, ld_buffer, ph,
                                                  toe_addr, toe_size);
                                num_load_seg++;
                        }
                        break;
                case PT_NOTE:
                        DEBUG_PRINTF("SPE_LOAD found PT_NOTE\n");
                        break;
                }
        }
        if (num_load_seg == 0)
          {
                  DEBUG_PRINTF ("no segments to load");
                  errno = EINVAL;
                  return -errno;
          }

        /* Remember where the code wants to be started */
        ld_info->entry = ehdr->e_entry;
        DEBUG_PRINTF ("entry = 0x%x\n", ehdr->e_entry);

        return 0;

}
```

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.

```
{
        Elf32_Ehdr *ehdr = (Elf32_Ehdr *)handle->elf_image;
        Elf32_Phdr *phdr;

        if (!ehdr) {
                DEBUG_PRINTF("No ELF image has been loaded\n");
                errno = EINVAL;
                return -errno;
        }

        if (ehdr->e_phentsize != sizeof(*phdr)) {
                DEBUG_PRINTF("Invalid program header format (phdr size=%d)\n",
                                ehdr->e_phentsize);
                errno = EINVAL;
                return -errno;
        }

        if (ehdr->e_phnum != 1) {
                DEBUG_PRINTF("Invalid program header count (%d), expected 1\n",
                                ehdr->e_phnum);
                errno = EINVAL;
                return -errno;
        }

        phdr = (Elf32_Phdr *)(handle->elf_image + ehdr->e_phoff);

        if (phdr->p_type != PT_LOAD || phdr->p_memsz == 0) {
                DEBUG_PRINTF("SPE program segment is not loadable (type=%x)\n",
                                phdr->p_type);
                errno = EINVAL;
                return -errno;
        }

        if (addr)
                *addr = (uint64_t)(unsigned long)
                        (handle->elf_image + phdr->p_offset);

        if (size)
                *size = phdr->p_memsz;

        return 0;
}
```

3.9.2.3 int _base_spe_toe_ear (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.

```
{
        Elf32_Ehdr *ehdr;
        Elf32_Shdr *shdr, *sh;
        char *str_table;
        char **ch;
        int ret;
        long toe_size;

        ehdr = (Elf32_Ehdr*) (speh->elf_image);
        shdr = (Elf32_Shdr*) ((char*) ehdr + ehdr->e_shoff);
        str_table = (char*)ehdr + shdr[ehdr->e_shstrndx].sh_offset;

        toe_size = 0;
        for (sh = shdr; sh < &shdr[ehdr->e_shnum]; ++sh)
                if (strcmp(".toe", str_table + sh->sh_name) == 0)
                        toe_size += sh->sh_size;

        ret = 0;
        if (toe_size > 0) {
                for (sh = shdr; sh < &shdr[ehdr->e_shnum]; ++sh)
                        if (sh->sh_type == SHT_SYMTAB || sh->sh_type ==
                            SHT_DYNSYM)
                                ret = toe_check_syms(ehdr, sh);
                if (!ret && toe_size != 16) {
                        /* Paranoia */
                        fprintf(stderr, "Unexpected toe size of %ld\n",
                                toe_size);
                        errno = EINVAL;
                        ret = 1;
                }
        }
        if (!ret && toe_size) {
                /*
                 * Allocate toe_shadow, and fill it with elf_image.
                 */
                speh->toe_shadow = malloc(toe_size);
                if (speh->toe_shadow) {
                        ch = (char**) speh->toe_shadow;
                        if (sizeof(char*) == 8) {
                                ch[0] = (char*) speh->elf_image;
                                ch[1] = 0;
                        } else {
                                ch[0] = 0;
                                ch[1] = (char*) speh->elf_image;
                                ch[2] = 0;
                                ch[3] = 0;
                        }
                } else {
                        errno = ENOMEM;
                        ret = 1;
                }
        }
        return ret;
}
```

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.

```
{
        Elf32_Ehdr *ehdr;
        void *elf_start;

        elf_start = handle->elf_image;
        ehdr = (Elf32_Ehdr *)(handle->elf_image);

        return check_spe_elf(ehdr);
}
```

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 & 0xffffff)
- #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 & 0xffffff)

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:

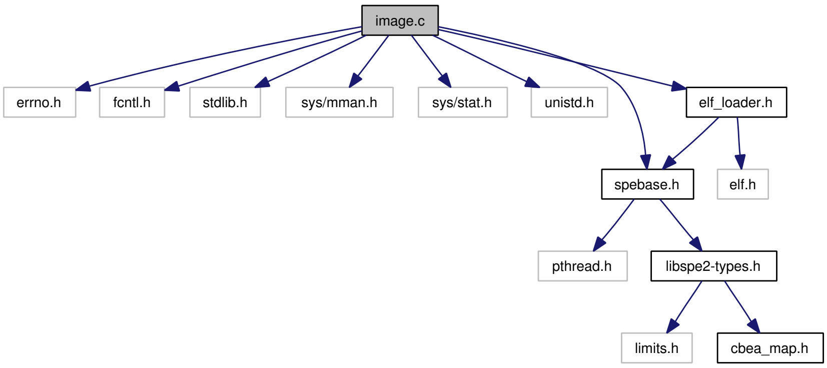
```
ret->slot[0] = (unsigned int) (_a);                 \
    ret->slot[1] = (unsigned int) (_b);                 \
    ret->slot[2] = (unsigned int) (_c);                 \
    ret->slot[3] = (unsigned int) (_d);                 \
    __asm__ __volatile__ ("sync" : : : "memory")
```

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.

-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.

```
{
       int ret = 0;
       struct image_handle *ih;

        if (!handle) {
                errno = EINVAL;
                return -1;
        }

        ih = (struct image_handle *)handle;

        if (!ih->speh.elf_image || !ih->map_size) {
                errno = EINVAL;
                return -1;
        }

        if (ih->speh.toe_shadow)
                free(ih->speh.toe_shadow);

        ret = munmap(ih->speh.elf_image, ih->map_size );
        free(handle);

        return ret;
}
```

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

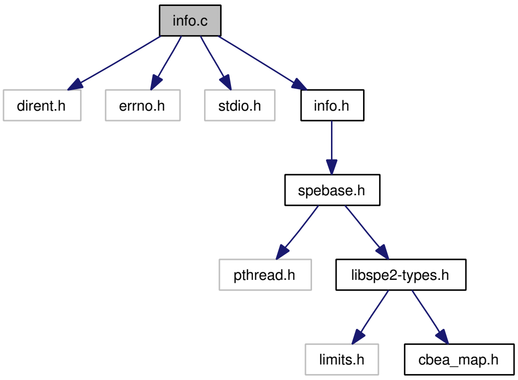
Definition at line 37 of file image.c.

```
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().

```
{
        const char    *buff = "/sys/devices/system/cpu";
        DIR     *dirp;
        int ret = -2;
        struct  dirent  *dptr;

        DEBUG_PRINTF ("spe_count_physical_cpus()\n");

        // make sure, cpu_node is in the correct range
        if (cpu_node != -1) {
                errno = EINVAL;
                return -1;
        }

        // Count number of CPUs in /sys/devices/system/cpu
        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

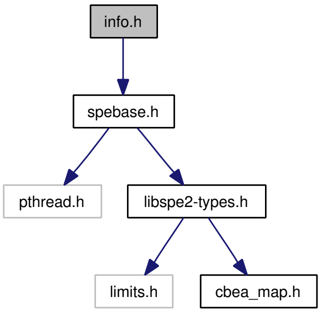
Definition at line 105 of file info.c.

```
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 }
```

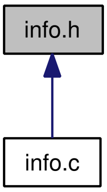
3.13 info.h File Reference

```
#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*)

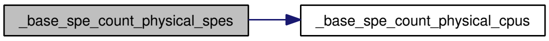
Definition at line 71 of file info.c.

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 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 
```

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

Here is the call graph for this function:



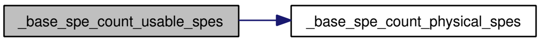
3.13.2.3 int _base_spe_count_usable_spes (int *cpu_node*)

Definition at line 62 of file info.c.

References _base_spe_count_physical_spes().

```
{
        return _base_spe_count_physical_spes(cpu_node); // FIXME
}
```

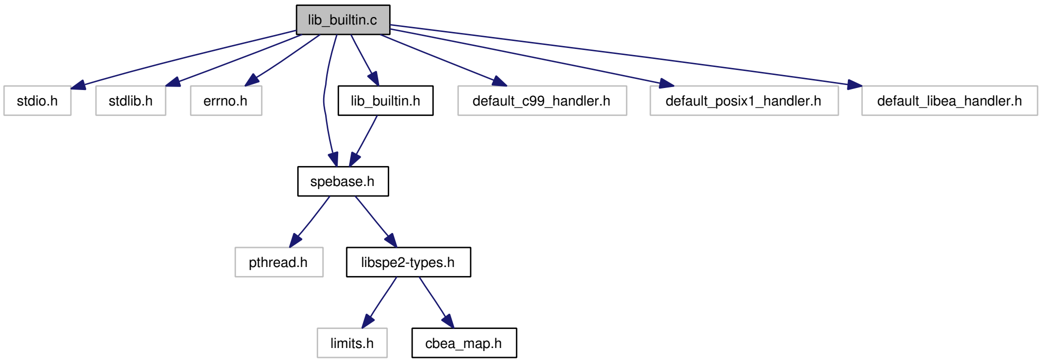
Here is the call graph for this function:



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.

```
{
        errno = 0;
        if (callnum > MAX_CALLNUM) {
                errno = EINVAL;
                return -1;
        }
        if (callnum < RESERVED) {
                errno = EACCES;
                return -1;
        }
        if (handlers[callnum] == NULL) {
                errno = ESRCH;
                return -1;
        }

        handlers[callnum] = NULL;
        return 0;
}
```

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.

```
{
        errno = 0;

        if (callnum > MAX_CALLNUM) {
                errno = EINVAL;
                return NULL;
        }
        if (handlers[callnum] == NULL) {
                errno = ESRCH;
                return NULL;
        }
        return handlers[callnum];
}
```

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.

```
{
        errno = 0;

        if (callnum > MAX_CALLNUM) {
                errno = EINVAL;
                return -1;
        }

        switch(mode){
        case SPE_CALLBACK_NEW:
                if (callnum < RESERVED) {
                        errno = EACCES;
                        return -1;
                }
                if (handlers[callnum] != NULL) {
                        errno = EACCES;
                        return -1;
                }
                handlers[callnum] = handler;
                break;

        case SPE_CALLBACK_UPDATE:
                if (handlers[callnum] == NULL) {
                        errno = ESRCH;
                        return -1;
                }
                handlers[callnum] = handler;
                break;
        default:
                errno = EINVAL;
                return -1;
                break;
        }
        return 0;

}
```

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.

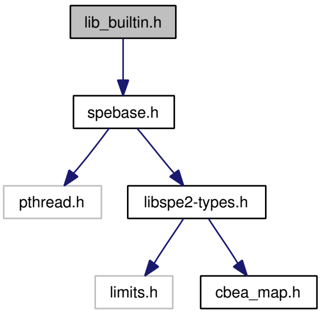
Referenced by _base_spe_context_run().

```
{
        int (*handler)(void *, unsigned int);
        int rc;

        errno = 0;
        if (!handlers[callnum]) {
                DEBUG_PRINTF ("No SPE library handler registered for this call.\n");
                errno=ENOSYS;
                return -1;
        }

        handler=handlers[callnum];

        /* For emulated isolation mode, position the
         * npc so that the buffer for the PPE-assisted
         * library calls can be accessed. */
        if (spe->base_private->flags & SPE_ISOLATE_EMULATE)
                npc = SPE_EMULATE_PARAM_BUFFER;

        rc = handler(spe->base_private->mem_mmap_base, npc);
        if (rc) {
                DEBUG_PRINTF ("SPE library call unsupported.\n");
                errno=ENOSYS;
                return rc;
        }
        return 0;
}
```

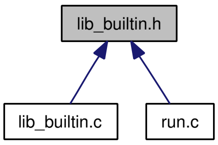
3.15 lib_builtin.h File Reference

```
#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.

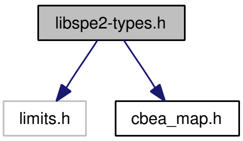
References spe_context::base_private, DEBUG_PRINTF, spe_context_base_priv::flags, spe_context_base_priv::mem_mmap_base, SPE_EMULATE_PARAM_BUFFER, and SPE_ISOLATE_EMULATE.

```
{
        int (*handler)(void *, unsigned int);
        int rc;

        errno = 0;
        if (!handlers[callnum]) {
                DEBUG_PRINTF ("No SPE library handler registered for this call.\n");
                errno=ENOSYS;
                return -1;
        }

        handler=handlers[callnum];

        /* For emulated isolation mode, position the
         * npc so that the buffer for the PPE-assisted
         * library calls can be accessed. */
        if (spe->base_private->flags & SPE_ISOLATE_EMULATE)
                npc = SPE_EMULATE_PARAM_BUFFER;

        rc = handler(spe->base_private->mem_mmap_base, npc);
        if (rc) {
                DEBUG_PRINTF ("SPE library call unsupported.\n");
                errno=ENOSYS;
                return rc;
        }
        return 0;
}
```

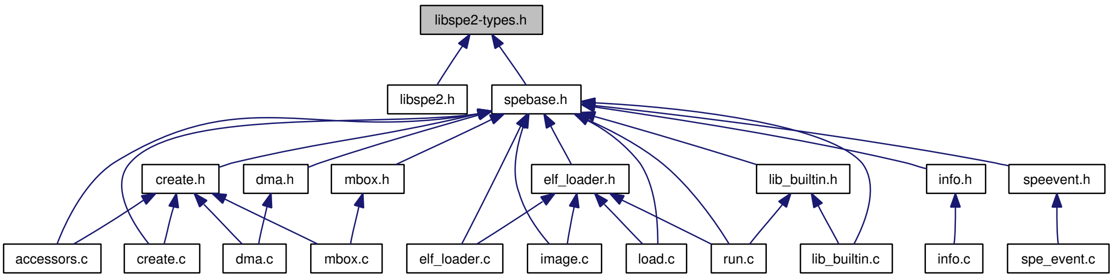
3.16 libspe2-types.h File Reference

```
#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

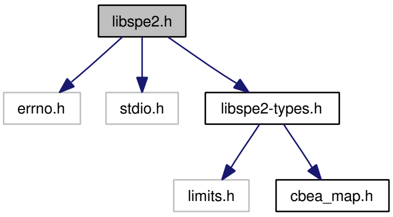
Definition at line 171 of file libspe2-types.h.

```
171 { SPE_MSSYNC_AREA, SPE_MFC_COMMAND_AREA, SPE_CONTROL_AREA, SPE_SIG_NOTIFY_1_AREA, SPE_SIG_NOTIFY_2_ARE
```

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*)

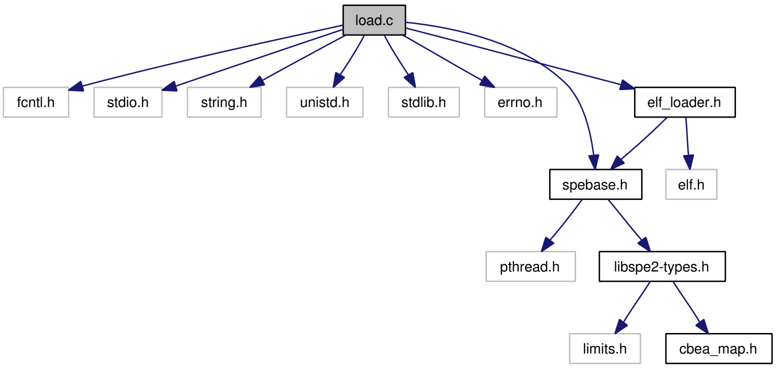
3.17.1.24 int spe_mfcio_getf (spe_context_ptr_t *spe*, unsigned int *ls*, void * *ea*, unsigned int *size*, unsigned int *tag*, unsigned int *tid*, unsigned int *rid*)

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 
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 }
```

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.

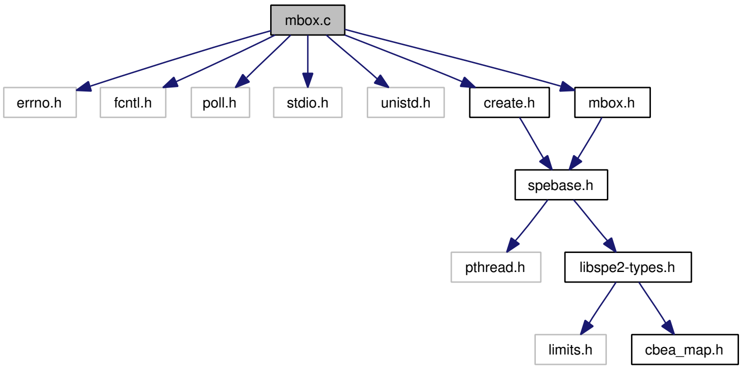
Referenced by _base_spe_context_run(), and _base_spe_program_load().

```
{
        int objfd, len;
        char buf[20];
        spe_program_handle_t *program;

        program = spectx->base_private->loaded_program;

        if (!program || !program->elf_image) {
                DEBUG_PRINTF("%s called, but no program loaded\n", __func__);
                return;
        }

        objfd = openat(spectx->base_private->fd_spe_dir, "object-id", O_RDWR);
        if (objfd < 0)
                return;

        len = sprintf(buf, "%p", program->elf_image);
        write(objfd, buf, len + 1);
        close(objfd);

        __spe_context_update_event();
}
```

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.

```
{
        int rc, ret;
        volatile struct spe_spu_control_area *cntl_area =
                spectx->base_private->cntl_mmap_base;

        if (spectx->base_private->flags & SPE_MAP_PS) {
                ret = (cntl_area->SPU_Mbox_Stat >> 8) & 0xFF;
        } else {
                rc = read(_base_spe_open_if_closed(spectx,FD_WBOX_STAT, 0), &ret, 4);
                if (rc != 4)
                        ret = -1;
        }

        return ret;

}
```

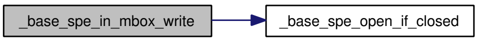
3.19.1.2 int _base_spe_in_mbox_write (spe_context_ptr_t *spectx*, unsigned int ∗ *mbox_data*, int *count*, int *behavior_flag*)

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.

```
{
        int rc;
        int total;
        unsigned int *aux;
        struct pollfd fds;

        if (mbox_data == NULL || count < 1){
                errno = EINVAL;
                return -1;
        }

        switch (behavior_flag) {
        case SPE_MBOX_ALL_BLOCKING: // write all, even if blocking
                total = rc = 0;
                if (spectx->base_private->flags & SPE_MAP_PS) {
                        do {
                                aux = mbox_data + total;
                                total += _base_spe_in_mbox_write_ps(spectx, aux, count - total);
                                if (total < count) { // we could not write everything, wait for space
                                        fds.fd = _base_spe_open_if_closed(spectx, FD_WBOX, 0);
                                        fds.events = POLLOUT;
                                        rc = poll(&fds, 1, -1);
                                        if (rc == -1 )
                                                return -1;
                                }
                        } while (total < count);
                } else {
                        while (total < 4*count) {
                                rc = write(_base_spe_open_if_closed(spectx,FD_WBOX, 0),
                                           (const char *)mbox_data + total, 4*count - total);
                                if (rc == -1) {
                                        break;
                                }
                                total += rc;
                        }
                        total /=4;
                }
                break;

        case  SPE_MBOX_ANY_BLOCKING: // write at least one, even if blocking
                total = rc = 0;
                if (spectx->base_private->flags & SPE_MAP_PS) {
                        do {
                                total = _base_spe_in_mbox_write_ps(spectx, mbox_data, count);
                                if (total == 0) { // we could not anything, wait for space
                                        fds.fd = _base_spe_open_if_closed(spectx, FD_WBOX, 0);
                                        fds.events = POLLOUT;
                                        rc = poll(&fds, 1, -1);
                                        if (rc == -1 )
                                                return -1;
                                }
                        } while (total == 0);
                } else {
                        rc = write(_base_spe_open_if_closed(spectx,FD_WBOX, 0), mbox_data, 4*count);
                        total = rc/4;
                }
                break;

        case  SPE_MBOX_ANY_NONBLOCKING: // only write, if non blocking
                total = rc = 0;
                // write directly if we map the PS else write via spufs
                if (spectx->base_private->flags & SPE_MAP_PS) {
                        total = _base_spe_in_mbox_write_ps(spectx, mbox_data, count);
                } else {
                        rc = write(_base_spe_open_if_closed(spectx,FD_WBOX_NB, 0), mbox_data, 4*count)
                        if (rc == -1 && errno == EAGAIN) {
                                rc = 0;
                                errno = 0;
                        }
                        total = rc/4;
                }
                break;

        default:
                errno = EINVAL;
                return -1;
        }

        if (rc == -1) {
                errno = EIO;
                return -1;
        }

        return total;
}
```

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 reaad, 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.

```
{
        int rc, ret;
        volatile struct spe_spu_control_area *cntl_area =
                spectx->base_private->cntl_mmap_base;

        if (spectx->base_private->flags & SPE_MAP_PS) {
                ret = (cntl_area->SPU_Mbox_Stat >> 16) & 0xFF;
        } else {
                rc = read(_base_spe_open_if_closed(spectx,FD_IBOX_STAT, 0), &ret, 4);
                if (rc != 4)
                        ret = -1;

        }
        return ret;
}
```

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.

```
{
        int rc;

        if (mbox_data == NULL || count < 1){
                errno = EINVAL;
                return -1;
        }

        if (spectx->base_private->flags & SPE_MAP_PS) {
                rc = _base_spe_out_mbox_read_ps(spectx, mbox_data, count);
        } else {
                rc = read(_base_spe_open_if_closed(spectx,FD_MBOX, 0), mbox_data, count*4);
                DEBUG_PRINTF("%s read rc: %d\n", __FUNCTION__, rc);
                if (rc != -1) {
                        rc /= 4;
                } else {
                        if (errno == EAGAIN ) { // no data ready to be read
                                errno = 0;
                                rc = 0;
                        }
                }
        }
        return rc;
}
```

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.

```
{
        int rc, ret;
        volatile struct spe_spu_control_area *cntl_area =
                spectx->base_private->cntl_mmap_base;

        if (spectx->base_private->flags & SPE_MAP_PS) {
                ret = cntl_area->SPU_Mbox_Stat & 0xFF;
        } else {
                rc = read(_base_spe_open_if_closed(spectx,FD_MBOX_STAT, 0), &ret, 4);
                if (rc != 4)
                        ret = -1;
        }

        return ret;

}
```

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

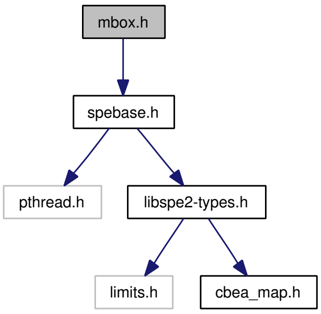
Definition at line 307 of file mbox.c.

```
{
        int rc;

        if (spectx->base_private->flags & SPE_MAP_PS) {
                if (signal_reg == SPE_SIG_NOTIFY_REG_1) {
                        spe_sig_notify_1_area_t *sig = spectx->base_private->signal1_mmap_base;

                        sig->SPU_Sig_Notify_1 = data;
                } else if (signal_reg == SPE_SIG_NOTIFY_REG_2) {
                        spe_sig_notify_2_area_t *sig = spectx->base_private->signal2_mmap_base;

                        sig->SPU_Sig_Notify_2 = data;
                } else {
                        errno = EINVAL;
                        return -1;
                }
                rc = 0;
        } else {
                if (signal_reg == SPE_SIG_NOTIFY_REG_1)
                        rc = write(_base_spe_open_if_closed(spectx,FD_SIG1, 0), &data, 4);
                else if (signal_reg == SPE_SIG_NOTIFY_REG_2)
                        rc = write(_base_spe_open_if_closed(spectx,FD_SIG2, 0), &data, 4);
                else {
                        errno = EINVAL;
                        return -1;
                }

                if (rc == 4)
                        rc = 0;

                if (signal_reg == SPE_SIG_NOTIFY_REG_1)
                        _base_spe_close_if_open(spectx,FD_SIG1);
                else if (signal_reg == SPE_SIG_NOTIFY_REG_2)
                        _base_spe_close_if_open(spectx,FD_SIG2);
        }

        return rc;
}
```

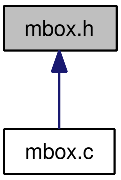
3.20 mbox.h File Reference

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

Include dependency graph for mbox.h:



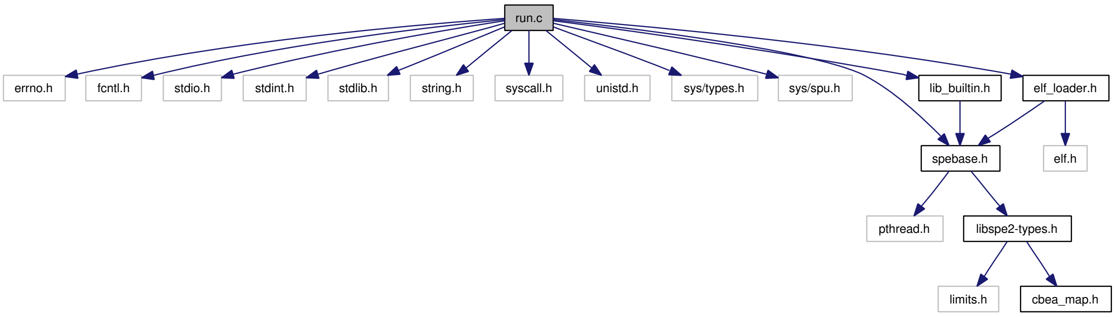
This graph shows which files directly or indirectly include this file:



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 programm. 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().

```
{
        int retval = 0, run_rc;
        unsigned int run_status, tmp_entry;
        spe_stop_info_t stopinfo_buf;
        struct spe_context_info this_context_info __attribute__((cleanup(cleanupspeinfo)));

        /* If the caller hasn't set a stopinfo buffer, provide a buffer on the
         * stack instead. */
        if (!stopinfo)
                stopinfo = &stopinfo_buf;


        /* In emulated isolated mode, the npc will always return as zero.
         * use our private entry point instead */
        if (spe->base_private->flags & SPE_ISOLATE_EMULATE)
                tmp_entry = spe->base_private->emulated_entry;

        else if (*entry == SPE_DEFAULT_ENTRY)
                tmp_entry = spe->base_private->entry;
        else
                tmp_entry = *entry;

        /* If we're starting the SPE binary from its original entry point,
         * setup the arguments to main() */
        if (tmp_entry == spe->base_private->entry &&
                        !(spe->base_private->flags &
                                (SPE_ISOLATE | SPE_ISOLATE_EMULATE))) {

```

```
                addr64 argp64, envp64, tid64, ls64;
                unsigned int regs[128][4];

                /* setup parameters */
                argp64.ull = (uint64_t)(unsigned long)argp;
                envp64.ull = (uint64_t)(unsigned long)envp;
                tid64.ull = (uint64_t)(unsigned long)spe;

                /* make sure the register values are 0 */
                memset(regs, 0, sizeof(regs));

                /* set sensible values for stack_ptr and stack_size */
                regs[1][0] = (unsigned int) LS_SIZE - 16;       /* stack_ptr */
                regs[2][0] = 0;                                                         /* stack_size

                if (runflags & SPE_RUN_USER_REGS) {
                        /* When SPE_USER_REGS is set, argp points to an array
                         * of 3x128b registers to be passed directly to the SPE
                         * program.
                         */
                        memcpy(regs[3], argp, sizeof(unsigned int) * 12);
                } else {
                        regs[3][0] = tid64.ui[0];
                        regs[3][1] = tid64.ui[1];

                        regs[4][0] = argp64.ui[0];
                        regs[4][1] = argp64.ui[1];

                        regs[5][0] = envp64.ui[0];
                        regs[5][1] = envp64.ui[1];
                }

                /* Store the LS base address in R6 */
                ls64.ull = (uint64_t)(unsigned long)spe->base_private->mem_mmap_base;
                regs[6][0] = ls64.ui[0];
                regs[6][1] = ls64.ui[1];

                if (set_regs(spe, regs))
                        return -1;
        }

        /*Leave a trail of breadcrumbs for the debugger to follow */
        if (!__spe_current_active_context) {
                __spe_current_active_context = &this_context_info;
                if (!__spe_current_active_context)
                        return -1;
                __spe_current_active_context->prev = NULL;
        } else {
                struct spe_context_info *newinfo;
                newinfo = &this_context_info;
                if (!newinfo)
                        return -1;
                newinfo->prev = __spe_current_active_context;
                __spe_current_active_context = newinfo;
        }
        /*remember the ls-addr*/
        __spe_current_active_context->spe_id = spe->base_private->fd_spe_dir;

do_run:
        /*Remember the npc value*/
        __spe_current_active_context->npc = tmp_entry;

        /* run SPE context */
        run_rc = spu_run(spe->base_private->fd_spe_dir,
                        &tmp_entry, &run_status);

        /*Remember the npc value*/
        __spe_current_active_context->npc = tmp_entry;
        __spe_current_active_context->status = run_status;

        DEBUG_PRINTF("spu_run returned run_rc=0x%08x, entry=0x%04x, "
                        "ext_status=0x%04x.\n", run_rc, tmp_entry, run_status);

        /* set up return values and stopinfo according to spu_run exit
         * conditions. This is overwritten on error.
         */
        stopinfo->spu_status = run_rc;

        if (spe->base_private->flags & SPE_ISOLATE_EMULATE) {
                /* save the entry point, and pretend that the npc is zero */
                spe->base_private->emulated_entry = tmp_entry;
                *entry = 0;
        } else {
                *entry = tmp_entry;
        }

        /* Return with stopinfo set on syscall error paths */
        if (run_rc == -1) {
                DEBUG_PRINTF("spu_run returned error %d, errno=%d\n",
                                run_rc, errno);
                stopinfo->stop_reason = SPE_RUNTIME_FATAL;
                stopinfo->result.spe_runtime_fatal = errno;
                retval = -1;

                /* For isolated contexts, pass EPERM up to the
                 * caller.
                 */
                if (!(spe->base_private->flags & SPE_ISOLATE
                                && errno == EPERM))
                        errno = EFAULT;

        } else if (run_rc & SPE_SPU_INVALID_INSTR) {
                DEBUG_PRINTF("SPU has tried to execute an invalid "
                                "instruction. %d\n", run_rc);
                stopinfo->stop_reason = SPE_RUNTIME_ERROR;
                stopinfo->result.spe_runtime_error = SPE_SPU_INVALID_INSTR;
                errno = EFAULT;
                retval = -1;

        } else if ((spe->base_private->flags & SPE_EVENTS_ENABLE) && run_status) {
                /* Report asynchronous error if return val are set and
                 * SPU events are enabled.
                 */
                stopinfo->stop_reason = SPE_RUNTIME_EXCEPTION;
                stopinfo->result.spe_runtime_exception = run_status;
                stopinfo->spu_status = -1;
                errno = EIO;
                retval = -1;

        } else if (run_rc & SPE_SPU_STOPPED_BY_STOP) {
                /* Stop & signals are broken down into three groups
                 *  1. SPE library call
                 *  2. SPE user defined stop & signal
                 *  3. SPE program end.
                 *
                 * These groups are signified by the 14-bit stop code:
                 */
                int stopcode = (run_rc >> 16) & 0x3fff;

                /* Check if this is a library callback, and callbacks are
                 * allowed (ie, running without SPE_NO_CALLBACKS)
                 */
                if ((stopcode & 0xff00) == SPE_PROGRAM_LIBRARY_CALL
                                && !(runflags & SPE_NO_CALLBACKS)) {
```

```

                        int callback_rc, callback_number = stopcode & 0xff;

                        /* execute library callback */
                        DEBUG_PRINTF("SPE library call: %d\n", callback_number);
                        callback_rc = _base_spe_handle_library_callback(spe,
                                                                        callback_number, *entry);

                        if (callback_rc) {
                                /* library callback failed; set errno and
                                 * return immediately */
                                DEBUG_PRINTF("SPE library call failed: %d\n",
                                                callback_rc);
                                stopinfo->stop_reason = SPE_CALLBACK_ERROR;
                                stopinfo->result.spe_callback_error =
                                        callback_rc;
                                errno = EFAULT;
                                retval = -1;
                        } else {
                                /* successful library callback - restart the SPE
                                 * program at the next instruction */
                                tmp_entry += 4;
                                goto do_run;
                        }

                } else if ((stopcode & 0xff00) == SPE_PROGRAM_NORMAL_END) {
                        /* The SPE program has exited by exit(X) */
                        stopinfo->stop_reason = SPE_EXIT;
                        stopinfo->result.spe_exit_code = stopcode & 0xff;

                        if (spe->base_private->flags & SPE_ISOLATE) {
                                /* Issue an isolated exit, and re-run the SPE.
                                 * We should see a return value without the
                                 * 0x80 bit set. */
                                if (!issue_isolated_exit(spe))
                                        goto do_run;
                                retval = -1;
                        }

                } else if ((stopcode & 0xfff0) == SPE_PROGRAM_ISOLATED_STOP) {

                        /* 0x2206: isolated app has been loaded by loader;
                         * provide a hook for the debugger to catch this,
                         * and restart
                         */
                        if (stopcode == SPE_PROGRAM_ISO_LOAD_COMPLETE) {
                                _base_spe_program_load_complete(spe);
                                goto do_run;
                        } else {
                                stopinfo->stop_reason = SPE_ISOLATION_ERROR;
                                stopinfo->result.spe_isolation_error =
                                        stopcode & 0xf;
                        }

                } else if (spe->base_private->flags & SPE_ISOLATE &&
                                !(run_rc & 0x80)) {
                        /* We’ve successfully exited isolated mode */
                        retval = 0;

                } else {
                        /* User defined stop & signal, including
                         * callbacks when disabled */
                        stopinfo->stop_reason = SPE_STOP_AND_SIGNAL;
                        stopinfo->result.spe_signal_code = stopcode;
                        retval = stopcode;
                }

```

```
        } else if (run_rc & SPE_SPU_HALT) {
                DEBUG_PRINTF("SPU was stopped by halt. %d\n", run_rc);
                stopinfo->stop_reason = SPE_RUNTIME_ERROR;
                stopinfo->result.spe_runtime_error = SPE_SPU_HALT;
                errno = EFAULT;
                retval = -1;

        } else if (run_rc & SPE_SPU_WAITING_ON_CHANNEL) {
                DEBUG_PRINTF("SPU is waiting on channel. %d\n", run_rc);
                stopinfo->stop_reason = SPE_RUNTIME_EXCEPTION;
                stopinfo->result.spe_runtime_exception = run_status;
                stopinfo->spu_status = -1;
                errno = EIO;
                retval = -1;

        } else if (run_rc & SPE_SPU_INVALID_CHANNEL) {
                DEBUG_PRINTF("SPU has tried to access an invalid "
                                "channel. %d\n", run_rc);
                stopinfo->stop_reason = SPE_RUNTIME_ERROR;
                stopinfo->result.spe_runtime_error = SPE_SPU_INVALID_CHANNEL;
                errno = EFAULT;
                retval = -1;

        } else {
                DEBUG_PRINTF("spu_run returned invalid data: 0x%04x\n", run_rc);
                stopinfo->stop_reason = SPE_RUNTIME_FATAL;
                stopinfo->result.spe_runtime_fatal = -1;
                stopinfo->spu_status = -1;
                errno = EFAULT;
                retval = -1;

        }

        freespeinfo();
        return retval;
}
```

3.21.3 Variable Documentation

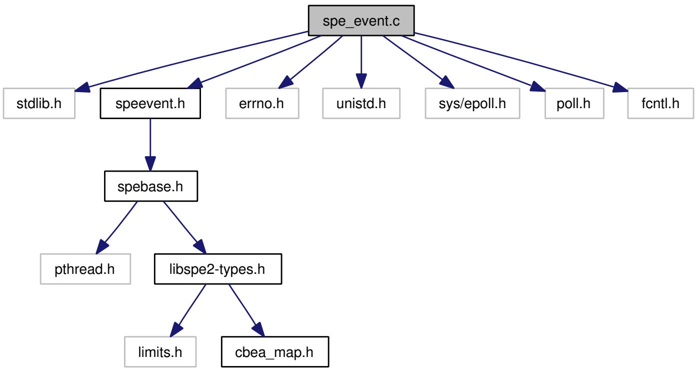
3.21.3.1 __thread struct spe_context_info∗ __spe_current_active_context

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.

```
{
  spe_context_event_priv_ptr_t evctx;

  if (!spe) {
    errno = ESRCH;
    return -1;
  }

  evctx = __SPE_EVENT_CONTEXT_PRIV_GET(spe);
  __SPE_EVENT_CONTEXT_PRIV_SET(spe, NULL);

  close(evctx->stop_event_pipe[0]);
  close(evctx->stop_event_pipe[1]);

  pthread_mutex_destroy(&evctx->lock);
  pthread_mutex_destroy(&evctx->stop_event_read_lock);

  free(evctx);

  return 0;
}
```

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.

```
{
  spe_context_event_priv_ptr_t evctx;
  int rc;
  int i;

  evctx = calloc(1, sizeof(*evctx));
  if (!evctx) {
    return NULL;
  }

  rc = pipe(evctx->stop_event_pipe);
  if (rc == -1) {
    free(evctx);
    return NULL;
  }
  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().

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

3.22.2.6 spe_event_handler_ptr_t _event_spe_event_handler_create (void)

Definition at line 110 of file spe_event.c.

```
{
  int epfd;
  spe_event_handler_t *evhandler;

  evhandler = calloc(1, sizeof(*evhandler));
  if (!evhandler) {
    return NULL;
  }

  epfd = epoll_create(__SPE_EPOLL_SIZE);
  if (epfd == -1) {
    free(evhandler);
    return NULL;
  }

  __SPE_EPOLL_FD_SET(evhandler, epfd);

  return evhandler;
}
```

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.

```
{
  int epfd;
  const int ep_op = EPOLL_CTL_DEL;
  spe_context_event_priv_ptr_t evctx;
  int fd;

  if (!evhandler) {
    errno = ESRCH;
    return -1;
  }
  if (!event || !event->spe) {
    errno = EINVAL;
    return -1;
  }
  if (!__SPE_EVENTS_ENABLED(event->spe)) {
    errno = ENOTSUP;
    return -1;
  }

  epfd = __SPE_EPOLL_FD_GET(evhandler);
  evctx = __SPE_EVENT_CONTEXT_PRIV_GET(event->spe);

  if (event->events & ~__SPE_EVENT_ALL) {
    errno = ENOTSUP;
    return -1;
  }

  _event_spe_context_lock(event->spe); /* for spe->event_private->events */

  if (event->events & SPE_EVENT_OUT_INTR_MBOX) {
    fd = __base_spe_event_source_acquire(event->spe, FD_IBOX);
    if (fd == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }
    if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }
    evctx->events[__SPE_EVENT_OUT_INTR_MBOX].events = 0;
  }

  if (event->events & SPE_EVENT_IN_MBOX) {
    fd = __base_spe_event_source_acquire(event->spe, FD_WBOX);
    if (fd == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }
    if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }
    evctx->events[__SPE_EVENT_IN_MBOX].events = 0;
  }

  if (event->events & SPE_EVENT_TAG_GROUP) {
    fd = __base_spe_event_source_acquire(event->spe, FD_MFC);
    if (fd == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }
    if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }
    evctx->events[__SPE_EVENT_TAG_GROUP].events = 0;
  }

  if (event->events & SPE_EVENT_SPE_STOPPED) {
    fd = evctx->stop_event_pipe[0];
    if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }
    evctx->events[__SPE_EVENT_SPE_STOPPED].events = 0;
  }

  _event_spe_context_unlock(event->spe);

  return 0;
}
```

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.

```
{
  int epfd;

  if (!evhandler) {
    errno = ESRCH;
    return -1;
  }

  epfd = __SPE_EPOLL_FD_GET(evhandler);
  close(epfd);

  free(evhandler);
  return 0;
}
```

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.

```
{
  int epfd;
  const int ep_op = EPOLL_CTL_ADD;
  spe_context_event_priv_ptr_t evctx;
  spe_event_unit_t *ev_buf;
  struct epoll_event ep_event;
  int fd;

  if (!evhandler) {
    errno = ESRCH;
    return -1;
  }
  if (!event || !event->spe) {
    errno = EINVAL;
    return -1;
  }
  if (!__SPE_EVENTS_ENABLED(event->spe)) {
    errno = ENOTSUP;
    return -1;
  }

  epfd = __SPE_EPOLL_FD_GET(evhandler);
  evctx = __SPE_EVENT_CONTEXT_PRIV_GET(event->spe);

  if (event->events & ~__SPE_EVENT_ALL) {
    errno = ENOTSUP;
    return -1;
  }

  _event_spe_context_lock(event->spe); /* for spe->event_private->events */

  if (event->events & SPE_EVENT_OUT_INTR_MBOX) {
    fd = __base_spe_event_source_acquire(event->spe, FD_IBOX);
    if (fd == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }

    ev_buf = &evctx->events[__SPE_EVENT_OUT_INTR_MBOX];
    ev_buf->events = SPE_EVENT_OUT_INTR_MBOX;
    ev_buf->data = event->data;

    ep_event.events = EPOLLIN;
    ep_event.data.ptr = ev_buf;
    if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }
  }

  if (event->events & SPE_EVENT_IN_MBOX) {
    fd = __base_spe_event_source_acquire(event->spe, FD_WBOX);
    if (fd == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }

    ev_buf = &evctx->events[__SPE_EVENT_IN_MBOX];
    ev_buf->events = SPE_EVENT_IN_MBOX;
    ev_buf->data = event->data;

    ep_event.events = EPOLLOUT;
    ep_event.data.ptr = ev_buf;
    if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }
  }

  if (event->events & SPE_EVENT_TAG_GROUP) {
    fd = __base_spe_event_source_acquire(event->spe, FD_MFC);
    if (fd == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }

    if (event->spe->base_private->flags & SPE_MAP_PS) {
            _event_spe_context_unlock(event->spe);
            errno = ENOTSUP;
            return -1;
    }

    ev_buf = &evctx->events[__SPE_EVENT_TAG_GROUP];
    ev_buf->events = SPE_EVENT_TAG_GROUP;
    ev_buf->data = event->data;

    ep_event.events = EPOLLIN;
    ep_event.data.ptr = ev_buf;
    if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }
  }

  if (event->events & SPE_EVENT_SPE_STOPPED) {
    fd = evctx->stop_event_pipe[0];

    ev_buf = &evctx->events[__SPE_EVENT_SPE_STOPPED];
    ev_buf->events = SPE_EVENT_SPE_STOPPED;
    ev_buf->data = event->data;

    ep_event.events = EPOLLIN;
    ep_event.data.ptr = ev_buf;
    if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }
  }

  _event_spe_context_unlock(event->spe);

  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*)

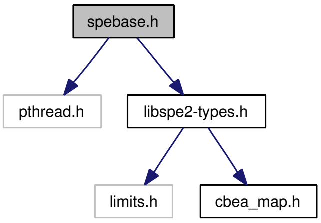
Definition at line 59 of file spe_event.c.

```
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 }
```

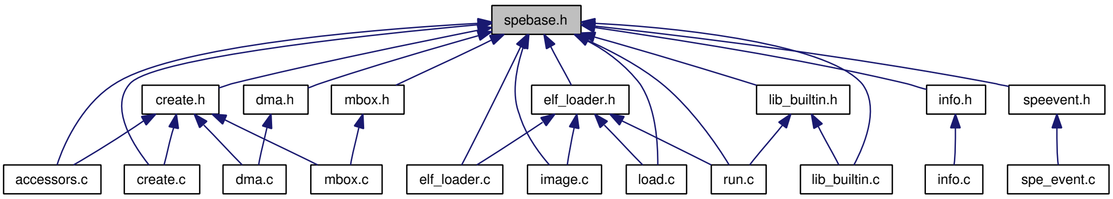
3.23 spebase.h File Reference

```
#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

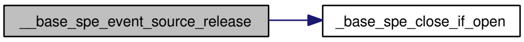
fdesc Specifies the event source

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.

```
{
        errno = 0;
        if (callnum > MAX_CALLNUM) {
                errno = EINVAL;
                return -1;
        }
        if (callnum < RESERVED) {
                errno = EACCES;
                return -1;
        }
        if (handlers[callnum] == NULL) {
                errno = ESRCH;
                return -1;
        }

        handlers[callnum] = NULL;
        return 0;
}
```

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.

```
{
        errno = 0;

        if (callnum > MAX_CALLNUM) {
                errno = EINVAL;
                return NULL;
        }
        if (handlers[callnum] == NULL) {
                errno = ESRCH;
                return NULL;
        }
        return handlers[callnum];
}
```

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.

```
{
        errno = 0;

        if (callnum > MAX_CALLNUM) {
                errno = EINVAL;
                return -1;
        }

        switch(mode){
        case SPE_CALLBACK_NEW:
                if (callnum < RESERVED) {
                        errno = EACCES;
                        return -1;
                }
                if (handlers[callnum] != NULL) {
                        errno = EACCES;
                        return -1;
                }
                handlers[callnum] = handler;
                break;

        case SPE_CALLBACK_UPDATE:
                if (handlers[callnum] == NULL) {
                        errno = ESRCH;
                        return -1;
                }
                handlers[callnum] = handler;
                break;
        default:
                errno = EINVAL;
                return -1;
                break;
        }
        return 0;

}
```

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

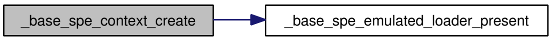
aff_spe specify NULL to skip affinity information

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.

```
{
        char pathname[256];
        int i, aff_spe_fd = 0;
        unsigned int spu_createflags = 0;
        struct spe_context *spe = NULL;
        struct spe_context_base_priv *priv;

        /* We need a loader present to run in emulated isolated mode */
        if (flags & SPE_ISOLATE_EMULATE
                        && !_base_spe_emulated_loader_present()) {
                errno = EINVAL;
                return NULL;
        }

        /* Put some sane defaults into the SPE context */
        spe = malloc(sizeof(*spe));
        if (!spe) {
                DEBUG_PRINTF("ERROR: Could not allocate spe context.\n");
                return NULL;
        }
        memset(spe, 0, sizeof(*spe));

        spe->base_private = malloc(sizeof(*spe->base_private));
        if (!spe->base_private) {
                DEBUG_PRINTF("ERROR: Could not allocate "
                                "spe->base_private context.\n");
                free(spe);
                return NULL;
        }

        /* just a convenience variable */
        priv = spe->base_private;

        priv->fd_spe_dir = -1;
        priv->mem_mmap_base = MAP_FAILED;
        priv->psmap_mmap_base = MAP_FAILED;
        priv->mssync_mmap_base = MAP_FAILED;
        priv->mfc_mmap_base = MAP_FAILED;
        priv->cntl_mmap_base = MAP_FAILED;
        priv->signal1_mmap_base = MAP_FAILED;
        priv->signal2_mmap_base = MAP_FAILED;
        priv->loaded_program = NULL;

        for (i = 0; i < NUM_MBOX_FDS; i++) {
                priv->spe_fds_array[i] = -1;
                pthread_mutex_init(&priv->fd_lock[i], NULL);
        }

        /* initialise spu_createflags */
        if (flags & SPE_ISOLATE) {
                flags |= SPE_MAP_PS;
                spu_createflags |= SPU_CREATE_ISOLATE | SPU_CREATE_NOSCHED;
        }

        if (flags & SPE_EVENTS_ENABLE)
                spu_createflags |= SPU_CREATE_EVENTS_ENABLED;

        if (aff_spe)
                spu_createflags |= SPU_CREATE_AFFINITY_SPU;

        if (flags & SPE_AFFINITY_MEMORY)
                spu_createflags |= SPU_CREATE_AFFINITY_MEM;

        /* Make the SPUFS directory for the SPE */
        if (gctx == NULL)
                sprintf(pathname, "/spu/spethread-%i-%lu",
                        getpid(), (unsigned long)spe);
        else
                sprintf(pathname, "/spu/%s/spethread-%i-%lu",
                        gctx->base_private->gangname, getpid(),
                        (unsigned long)spe);

        if (aff_spe)
                aff_spe_fd = aff_spe->base_private->fd_spe_dir;

        priv->fd_spe_dir = spu_create(pathname, spu_createflags,
                        S_IRUSR | S_IWUSR | S_IXUSR, aff_spe_fd);

        if (priv->fd_spe_dir < 0) {
                DEBUG_PRINTF("ERROR: Could not create SPE %s\n", pathname);
                perror("spu_create()");
                free_spe_context(spe);
                /* we mask most errors, but leave ENODEV */
                if (errno != ENODEV)
                        errno = EFAULT;
                return NULL;
        }

        priv->flags = flags;

        /* Map the required areas into process memory */
        priv->mem_mmap_base = mapfileat(priv->fd_spe_dir, "mem", LS_SIZE);
        if (priv->mem_mmap_base == MAP_FAILED) {
                DEBUG_PRINTF("ERROR: Could not map SPE memory.\n");
                free_spe_context(spe);
                errno = ENOMEM;
                return NULL;
        }

        if (flags & SPE_MAP_PS) {
                /* It's possible to map the entire problem state area with
                 * one mmap - try this first */
                priv->psmap_mmap_base =  mapfileat(priv->fd_spe_dir,
                                "psmap", PSMAP_SIZE);

                if (priv->psmap_mmap_base != MAP_FAILED) {
                        priv->mssync_mmap_base =
                                priv->psmap_mmap_base + MSSYNC_OFFSET;
                        priv->mfc_mmap_base =
                                priv->psmap_mmap_base + MFC_OFFSET;
                        priv->cntl_mmap_base =
                                priv->psmap_mmap_base + CNTL_OFFSET;
                        priv->signal1_mmap_base =
                                priv->psmap_mmap_base + SIGNAL1_OFFSET;
                        priv->signal2_mmap_base =
                                priv->psmap_mmap_base + SIGNAL2_OFFSET;

                } else {
                        /* map each region separately */
                        priv->mfc_mmap_base =
                                mapfileat(priv->fd_spe_dir, "mfc", MFC_SIZE);
                        priv->mssync_mmap_base =
                                mapfileat(priv->fd_spe_dir, "mss", MSS_SIZE);
                        priv->cntl_mmap_base =
                                mapfileat(priv->fd_spe_dir, "cntl", CNTL_SIZE);
                        priv->signal1_mmap_base =
                                mapfileat(priv->fd_spe_dir, "signal1",
                                                SIGNAL_SIZE);
                        priv->signal2_mmap_base =
                                mapfileat(priv->fd_spe_dir, "signal2",
                                                SIGNAL_SIZE);

                        if (priv->mfc_mmap_base == MAP_FAILED ||
                                        priv->cntl_mmap_base == MAP_FAILED ||
                                        priv->signal1_mmap_base == MAP_FAILED ||
                                        priv->signal2_mmap_base == MAP_FAILED) {
                                DEBUG_PRINTF("ERROR: Could not map SPE "
                                                "PS memory.\n");
                                free_spe_context(spe);
                                errno = ENOMEM;
                                return NULL;
                        }
                }
        }

        if (flags & SPE_CFG_SIGNOTIFY1_OR) {
                if (setsignotify(priv->fd_spe_dir, "signal1_type")) {
                        DEBUG_PRINTF("ERROR: Could not open SPE "
                                        "signal1_type file.\n");
                        free_spe_context(spe);
                        errno = EFAULT;
                        return NULL;
                }
        }

        if (flags & SPE_CFG_SIGNOTIFY2_OR) {
                if (setsignotify(priv->fd_spe_dir, "signal2_type")) {
                        DEBUG_PRINTF("ERROR: Could not open SPE "
                                        "signal2_type file.\n");
                        free_spe_context(spe);
                        errno = EFAULT;
                        return NULL;
                }
        }

        return spe;
}
```

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:

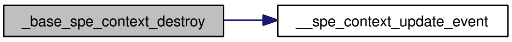
spectx Specifies the SPE context

Definition at line 406 of file create.c.

References __spe_context_update_event().

```
{
        int ret = free_spe_context(spe);

        __spe_context_update_event();

        return ret;
}
```

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 programm. 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.

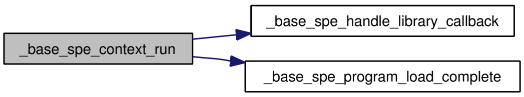
Definition at line 98 of file run.c.

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.

```
{
        int retval = 0, run_rc;
        unsigned int run_status, tmp_entry;
        spe_stop_info_t stopinfo_buf;
        struct spe_context_info this_context_info __attribute__((cleanup(cleanupspeinfo)));

        /* If the caller hasn't set a stopinfo buffer, provide a buffer on the
         * stack instead. */
        if (!stopinfo)
                stopinfo = &stopinfo_buf;


        /* In emulated isolated mode, the npc will always return as zero.
         * use our private entry point instead */
        if (spe->base_private->flags & SPE_ISOLATE_EMULATE)
                tmp_entry = spe->base_private->emulated_entry;

        else if (*entry == SPE_DEFAULT_ENTRY)
                tmp_entry = spe->base_private->entry;
        else
                tmp_entry = *entry;

        /* If we're starting the SPE binary from its original entry point,
         * setup the arguments to main() */
        if (tmp_entry == spe->base_private->entry &&
                        !(spe->base_private->flags &
                                (SPE_ISOLATE | SPE_ISOLATE_EMULATE))) {

                addr64 argp64, envp64, tid64, ls64;
                unsigned int regs[128][4];

                /* setup parameters */
                argp64.ull = (uint64_t)(unsigned long)argp;
                envp64.ull = (uint64_t)(unsigned long)envp;
                tid64.ull = (uint64_t)(unsigned long)spe;

                /* make sure the register values are 0 */
                memset(regs, 0, sizeof(regs));

                /* set sensible values for stack_ptr and stack_size */
                regs[1][0] = (unsigned int) LS_SIZE - 16;       /* stack_ptr */
                regs[2][0] = 0;                                                         /* stack_size

                if (runflags & SPE_RUN_USER_REGS) {
                        /* When SPE_USER_REGS is set, argp points to an array
                         * of 3x128b registers to be passed directly to the SPE
                         * program.
                         */
                        memcpy(regs[3], argp, sizeof(unsigned int) * 12);
                } else {
                        regs[3][0] = tid64.ui[0];
                        regs[3][1] = tid64.ui[1];

                        regs[4][0] = argp64.ui[0];
                        regs[4][1] = argp64.ui[1];

                        regs[5][0] = envp64.ui[0];
                        regs[5][1] = envp64.ui[1];
                }

                /* Store the LS base address in R6 */
                ls64.ull = (uint64_t)(unsigned long)spe->base_private->mem_mmap_base;
                regs[6][0] = ls64.ui[0];
                regs[6][1] = ls64.ui[1];

                if (set_regs(spe, regs))
                        return -1;
        }

        /*Leave a trail of breadcrumbs for the debugger to follow */
        if (!__spe_current_active_context) {
                __spe_current_active_context = &this_context_info;
                if (!__spe_current_active_context)
                        return -1;
                __spe_current_active_context->prev = NULL;
        } else {
                struct spe_context_info *newinfo;
                newinfo = &this_context_info;
                if (!newinfo)
                        return -1;
                newinfo->prev = __spe_current_active_context;
                __spe_current_active_context = newinfo;
        }
        /*remember the ls-addr*/
        __spe_current_active_context->spe_id = spe->base_private->fd_spe_dir;

do_run:
        /*Remember the npc value*/
        __spe_current_active_context->npc = tmp_entry;

        /* run SPE context */
        run_rc = spu_run(spe->base_private->fd_spe_dir,
                        &tmp_entry, &run_status);

        /*Remember the npc value*/
        __spe_current_active_context->npc = tmp_entry;
        __spe_current_active_context->status = run_status;

        DEBUG_PRINTF("spu_run returned run_rc=0x%08x, entry=0x%04x, "
                        "ext_status=0x%04x.\n", run_rc, tmp_entry, run_status);

        /* set up return values and stopinfo according to spu_run exit
         * conditions. This is overwritten on error.
         */
        stopinfo->spu_status = run_rc;

        if (spe->base_private->flags & SPE_ISOLATE_EMULATE) {
                /* save the entry point, and pretend that the npc is zero */
                spe->base_private->emulated_entry = tmp_entry;
                *entry = 0;
        } else {
                *entry = tmp_entry;
        }

        /* Return with stopinfo set on syscall error paths */
        if (run_rc == -1) {
                DEBUG_PRINTF("spu_run returned error %d, errno=%d\n",
                                run_rc, errno);
                stopinfo->stop_reason = SPE_RUNTIME_FATAL;
                stopinfo->result.spe_runtime_fatal = errno;
                retval = -1;

                /* For isolated contexts, pass EPERM up to the
                 * caller.
                 */
                if (!(spe->base_private->flags & SPE_ISOLATE
                                && errno == EPERM))
                        errno = EFAULT;

        } else if (run_rc & SPE_SPU_INVALID_INSTR) {
                DEBUG_PRINTF("SPU has tried to execute an invalid "
                                "instruction. %d\n", run_rc);
                stopinfo->stop_reason = SPE_RUNTIME_ERROR;
                stopinfo->result.spe_runtime_error = SPE_SPU_INVALID_INSTR;
                errno = EFAULT;
                retval = -1;

        } else if ((spe->base_private->flags & SPE_EVENTS_ENABLE) && run_status) {
                /* Report asynchronous error if return val are set and
                 * SPU events are enabled.
                 */
                stopinfo->stop_reason = SPE_RUNTIME_EXCEPTION;
                stopinfo->result.spe_runtime_exception = run_status;
                stopinfo->spu_status = -1;
                errno = EIO;
                retval = -1;

        } else if (run_rc & SPE_SPU_STOPPED_BY_STOP) {
                /* Stop & signals are broken down into three groups
                 *  1. SPE library call
                 *  2. SPE user defined stop & signal
                 *  3. SPE program end.
                 *
                 * These groups are signified by the 14-bit stop code:
                 */
                int stopcode = (run_rc >> 16) & 0x3fff;

                /* Check if this is a library callback, and callbacks are
                 * allowed (ie, running without SPE_NO_CALLBACKS)
                 */
                if ((stopcode & 0xff00) == SPE_PROGRAM_LIBRARY_CALL
                                && !(runflags & SPE_NO_CALLBACKS)) {

                        int callback_rc, callback_number = stopcode & 0xff;

                        /* execute library callback */
                        DEBUG_PRINTF("SPE library call: %d\n", callback_number);
                        callback_rc = _base_spe_handle_library_callback(spe,
                                                                        callback_number, *entry);

                        if (callback_rc) {
                                /* library callback failed; set errno and
                                 * return immediately */
                                DEBUG_PRINTF("SPE library call failed: %d\n",
                                                callback_rc);
                                stopinfo->stop_reason = SPE_CALLBACK_ERROR;
                                stopinfo->result.spe_callback_error =
                                        callback_rc;
                                errno = EFAULT;
                                retval = -1;
                        } else {
                                /* successful library callback - restart the SPE
                                 * program at the next instruction */
                                tmp_entry += 4;
                                goto do_run;
                        }

                } else if ((stopcode & 0xff00) == SPE_PROGRAM_NORMAL_END) {
                        /* The SPE program has exited by exit(X) */
                        stopinfo->stop_reason = SPE_EXIT;
                        stopinfo->result.spe_exit_code = stopcode & 0xff;

                        if (spe->base_private->flags & SPE_ISOLATE) {
                                /* Issue an isolated exit, and re-run the SPE.
                                 * We should see a return value without the
                                 * 0x80 bit set. */
                                if (!issue_isolated_exit(spe))
                                        goto do_run;
                                retval = -1;
                        }

                } else if ((stopcode & 0xfff0) == SPE_PROGRAM_ISOLATED_STOP) {

                        /* 0x2206: isolated app has been loaded by loader;
                         * provide a hook for the debugger to catch this,
                         * and restart
                         */
                        if (stopcode == SPE_PROGRAM_ISO_LOAD_COMPLETE) {
                                _base_spe_program_load_complete(spe);
                                goto do_run;
                        } else {
                                stopinfo->stop_reason = SPE_ISOLATION_ERROR;
                                stopinfo->result.spe_isolation_error =
                                        stopcode & 0xf;
                        }

                } else if (spe->base_private->flags & SPE_ISOLATE &&
                                !(run_rc & 0x80)) {
                        /* We’ve successfully exited isolated mode */
                        retval = 0;

                } else {
                        /* User defined stop & signal, including
                         * callbacks when disabled */
                        stopinfo->stop_reason = SPE_STOP_AND_SIGNAL;
                        stopinfo->result.spe_signal_code = stopcode;
                        retval = stopcode;
                }

        } else if (run_rc & SPE_SPU_HALT) {
                DEBUG_PRINTF("SPU was stopped by halt. %d\n", run_rc);
                stopinfo->stop_reason = SPE_RUNTIME_ERROR;
                stopinfo->result.spe_runtime_error = SPE_SPU_HALT;
                errno = EFAULT;
                retval = -1;

        } else if (run_rc & SPE_SPU_WAITING_ON_CHANNEL) {
                DEBUG_PRINTF("SPU is waiting on channel. %d\n", run_rc);
                stopinfo->stop_reason = SPE_RUNTIME_EXCEPTION;
                stopinfo->result.spe_runtime_exception = run_status;
                stopinfo->spu_status = -1;
                errno = EIO;
                retval = -1;

        } else if (run_rc & SPE_SPU_INVALID_CHANNEL) {
                DEBUG_PRINTF("SPU has tried to access an invalid "
                                "channel. %d\n", run_rc);
                stopinfo->stop_reason = SPE_RUNTIME_ERROR;
                stopinfo->result.spe_runtime_error = SPE_SPU_INVALID_CHANNEL;
                errno = EFAULT;
                retval = -1;

        } else {
                DEBUG_PRINTF("spu_run returned invalid data: 0x%04x\n", run_rc);
                stopinfo->stop_reason = SPE_RUNTIME_FATAL;
                stopinfo->result.spe_runtime_fatal = -1;
                stopinfo->spu_status = -1;
                errno = EFAULT;
                retval = -1;
```

```
360
361         }
362
363         freespeinfo();
364         return retval;
365 }
```

Here is the call graph for this function:


_base_spe_context_run
_base_spe_handle_library_callback
_base_spe_program_load_complete

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

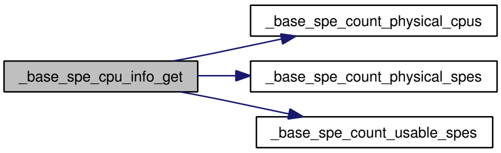
Definition at line 105 of file info.c.

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:
```

```
                errno = EINVAL;
                ret = -1;
        }
        return ret;
}
```

Here is the call graph for this function:


_base_spe_cpu_info_get
_base_spe_count_physical_cpus
_base_spe_count_physical_spes
_base_spe_count_usable_spes

3.23.4.16 int _base_spe_emulated_loader_present (void)

Check if the emulated loader is present in the filesystem

Returns:

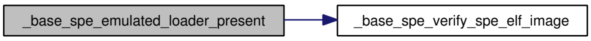
Non-zero if the loader is available, otherwise zero.

Definition at line 145 of file load.c.

References _base_spe_verify_spe_elf_image().

```
{
        spe_program_handle_t *loader = emulated_loader_program();

        if (!loader)
                return 0;

        return !_base_spe_verify_spe_elf_image(loader);
}
```

Here is the call graph for this function:


_base_spe_emulated_loader_present
_base_spe_verify_spe_elf_image

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.

```
{
        char pathname[256];
        struct spe_gang_context_base_priv *pgctx = NULL;
        struct spe_gang_context *gctx = NULL;

        gctx = malloc(sizeof(*gctx));
        if (!gctx) {
                DEBUG_PRINTF("ERROR: Could not allocate spe context.\n");
                return NULL;
        }
        memset(gctx, 0, sizeof(*gctx));

        pgctx = malloc(sizeof(*pgctx));
        if (!pgctx) {
                DEBUG_PRINTF("ERROR: Could not allocate spe context.\n");
                free(gctx);
                return NULL;
        }
        memset(pgctx, 0, sizeof(*pgctx));

        gctx->base_private = pgctx;

        sprintf(gctx->base_private->gangname, "gang-%i-%lu", getpid(),
                        (unsigned long)gctx);
        sprintf(pathname, "/spu/%s", gctx->base_private->gangname);

        gctx->base_private->fd_gang_dir = spu_create(pathname, SPU_CREATE_GANG,
                                S_IRUSR | S_IWUSR | S_IXUSR);

        if (gctx->base_private->fd_gang_dir < 0) {
                DEBUG_PRINTF("ERROR: Could not create Gang %s\n", pathname);
                free_spe_gang_context(gctx);
                errno = EFAULT;
                return NULL;
        }

        gctx->base_private->flags = flags;

        return gctx;
}
```

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.

```
{
        return free_spe_gang_context(gctx);
}
```

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.

```
{
        int ret = 0;
        struct image_handle *ih;

        if (!handle) {
                errno = EINVAL;
                return -1;
        }

        ih = (struct image_handle *)handle;

        if (!ih->speh.elf_image || !ih->map_size) {
                errno = EINVAL;
                return -1;
        }

        if (ih->speh.toe_shadow)
                free(ih->speh.toe_shadow);

        ret = munmap(ih->speh.elf_image, ih->map_size );
        free(handle);

        return ret;
}
```

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:

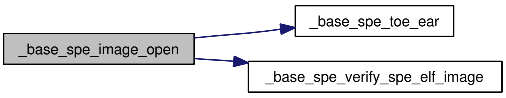
spe_create_thread

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.

```
{
        /* allocate an extra integer in the spe handle to keep the mapped size information */
        struct image_handle *ret;
        int binfd = -1, f_stat;
        struct stat statbuf;
        size_t ps = getpagesize ();

        ret = malloc(sizeof(struct image_handle));
        if (!ret)
                return NULL;

        ret->speh.handle_size = sizeof(spe_program_handle_t);
        ret->speh.toe_shadow = NULL;

        binfd = open(filename, O_RDONLY);
        if (binfd < 0)
                goto ret_err;

        f_stat = fstat(binfd, &statbuf);
        if (f_stat < 0)
                goto ret_err;

        /* Sanity: is it executable ?
         */
        if(!(statbuf.st_mode & (S_IXUSR | S_IXGRP | S_IXOTH))) {
                errno=EACCES;
                goto ret_err;
        }

        /* now store the size at the extra allocated space */
        ret->map_size = (statbuf.st_size + ps - 1) & ~(ps - 1);

        ret->speh.elf_image = mmap(NULL, ret->map_size,
                                                        PROT_WRITE | PROT_READ,
                                                        MAP_PRIVATE, binfd, 0);
        if (ret->speh.elf_image == MAP_FAILED)
                goto ret_err;

        /*Verify that this is a valid SPE ELF object*/
        if((_base_spe_verify_spe_elf_image((spe_program_handle_t *)ret)))
                goto ret_err;

        if (_base_spe_toe_ear(&ret->speh))
                goto ret_err;

        /* ok */
        close(binfd);
        return (spe_program_handle_t *)ret;

        /* err & cleanup */
ret_err:
        if (binfd >= 0)
                close(binfd);

        free(ret);
        return NULL;
}
```

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.

```
{
        int rc, ret;
        volatile struct spe_spu_control_area *cntl_area =
                spectx->base_private->cntl_mmap_base;

        if (spectx->base_private->flags & SPE_MAP_PS) {
                ret = (cntl_area->SPU_Mbox_Stat >> 8) & 0xFF;
        } else {
                rc = read(_base_spe_open_if_closed(spectx,FD_WBOX_STAT, 0), &ret, 4);
                if (rc != 4)
                        ret = -1;
        }

        return ret;

}
```

Here is the call graph for this function:

_base_spe_in_mbox_status
_base_spe_open_if_closed

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.

```
{
        return LS_SIZE;
}
```

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.

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

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.

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

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.

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

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.

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

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 i ssued 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.

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

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.

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

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.

```
{
        if ( mask != 0 ) {
                if (!(spectx->base_private->flags & SPE_MAP_PS))
                        mask = 0;
        } else {
                if ((spectx->base_private->flags & SPE_MAP_PS))
                        mask = spectx->base_private->active_tagmask;
        }

        if (!tag_status) {
                errno = EINVAL;
                return -1;
        }

        switch (behavior) {
        case SPE_TAG_ALL:
                return spe_mfcio_tag_status_read_all(spectx, mask, tag_status);
        case SPE_TAG_ANY:
                return spe_mfcio_tag_status_read_any(spectx, mask, tag_status);
        case SPE_TAG_IMMEDIATE:
                return spe_mfcio_tag_status_read_immediate(spectx, mask, tag_status);
        default:
                errno = EINVAL;
                return -1;
        }
}
```

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.

```
{
        int ret, fd;
        unsigned int data = 1; /* Any value can be written here */

        volatile struct spe_mssync_area *mss_area =
                spectx->base_private->mssync_mmap_base;

        if (spectx->base_private->flags & SPE_MAP_PS) {
                mss_area->MFC_MSSync = data;
                return 0;
        } else {
                fd = _base_spe_open_if_closed(spectx, FD_MSS, 0);
                if (fd != -1) {
                        ret = write(fd, &data, sizeof (data));
                        if ((ret < 0) && (errno != EIO)) {
                                perror("spe_mssync_start: internal error");
                        }
                        return ret < 0 ? -1 : 0;
                } else
                        return -1;
        }
}
```

Here is the call graph for this function:

_base_spe_mssync_start
_base_spe_open_if_closed

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.

```
{
        int ret, fd;
        unsigned int data;

        volatile struct spe_mssync_area *mss_area =
                spectx->base_private->mssync_mmap_base;

        if (spectx->base_private->flags & SPE_MAP_PS) {
                return  mss_area->MFC_MSSync;
        } else {
                fd = _base_spe_open_if_closed(spectx, FD_MSS, 0);
                if (fd != -1) {
                        ret = read(fd, &data, sizeof (data));
                        if ((ret < 0) && (errno != EIO)) {
                                perror("spe_mssync_start: internal error");
                        }
                        return ret < 0 ? -1 : data;
                } else
                        return -1;
        }
}
```

Here is the call graph for this function:

_base_spe_mssync_status
_base_spe_open_if_closed

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*)

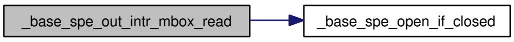
The _base_spe_out_intr_mbox_read function reads the contents of the SPE outbound interrupting mailbox for the SPE context.

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.

```
{
        int rc;
        int total;

        if (mbox_data == NULL || count < 1){
                errno = EINVAL;
                return -1;
        }

        switch (behavior_flag) {
        case SPE_MBOX_ALL_BLOCKING: // read all, even if blocking
                total = rc = 0;
                while (total < 4*count) {
                        rc = read(_base_spe_open_if_closed(spectx,FD_IBOX, 0),
                                  (char *)mbox_data + total, 4*count - total);
                        if (rc == -1) {
                                break;
                        }
                        total += rc;
                }
                break;

        case  SPE_MBOX_ANY_BLOCKING: // read at least one, even if blocking
                total = rc = read(_base_spe_open_if_closed(spectx,FD_IBOX, 0), mbox_data, 4*count);
                break;

        case  SPE_MBOX_ANY_NONBLOCKING: // only reaad, if non blocking
                rc = read(_base_spe_open_if_closed(spectx,FD_IBOX_NB, 0), mbox_data, 4*count);
                if (rc == -1 && errno == EAGAIN) {
                        rc = 0;
                        errno = 0;
                }
                total = rc;
                break;

        default:
                errno = EINVAL;
                return -1;
        }

        if (rc == -1) {
                errno = EIO;
                return -1;
        }

        return rc / 4;
}
```

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:

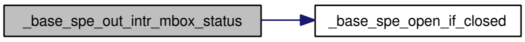
spe_read_out_mbox, spe_write_in_mbox, read (2)

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.

```
{
        int rc, ret;
        volatile struct spe_spu_control_area *cntl_area =
                spectx->base_private->cntl_mmap_base;

        if (spectx->base_private->flags & SPE_MAP_PS) {
                ret = (cntl_area->SPU_Mbox_Stat >> 16) & 0xFF;
        } else {
                rc = read(_base_spe_open_if_closed(spectx,FD_IBOX_STAT, 0), &ret, 4);
                if (rc != 4)
                        ret = -1;

        }
        return ret;
}
```

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.

```
{
        int rc;

        if (mbox_data == NULL || count < 1){
                errno = EINVAL;
                return -1;
        }

        if (spectx->base_private->flags & SPE_MAP_PS) {
                rc = _base_spe_out_mbox_read_ps(spectx, mbox_data, count);
        } else {
                rc = read(_base_spe_open_if_closed(spectx,FD_MBOX, 0), mbox_data, count*4);
                DEBUG_PRINTF("%s read rc: %d\n", __FUNCTION__, rc);
                if (rc != -1) {
                        rc /= 4;
                } else {
                        if (errno == EAGAIN ) { // no data ready to be read
                                errno = 0;
                                rc = 0;
                        }
                }
        }
        return rc;
}
```

Here is the call graph for this function:

_base_spe_out_mbox_read
_base_spe_open_if_closed

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:

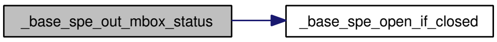
spe_read_out_mbox, spe_write_in_mbox, read (2)

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.

```
{
        int rc, ret;
        volatile struct spe_spu_control_area *cntl_area =
                spectx->base_private->cntl_mmap_base;

        if (spectx->base_private->flags & SPE_MAP_PS) {
                ret = cntl_area->SPU_Mbox_Stat & 0xFF;
        } else {
                rc = read(_base_spe_open_if_closed(spectx,FD_MBOX_STAT, 0), &ret, 4);
                if (rc != 4)
                        ret = -1;
        }

        return ret;

}
```

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

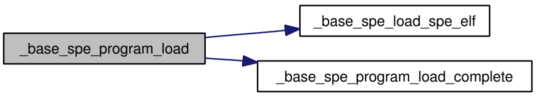
program handle to the ELF image

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.

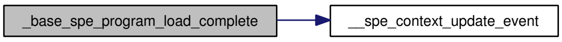
Register the SPE program's start address with the oprofile and gdb, by writing to the object-id file.

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 specifes 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:

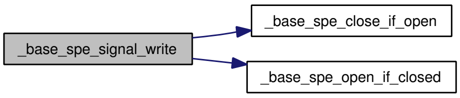
spe_get_ps_area, spe_write_in_mbox

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.

```
{
        int rc;

        if (spectx->base_private->flags & SPE_MAP_PS) {
                if (signal_reg == SPE_SIG_NOTIFY_REG_1) {
                        spe_sig_notify_1_area_t *sig = spectx->base_private->signal1_mmap_base;

                        sig->SPU_Sig_Notify_1 = data;
                } else if (signal_reg == SPE_SIG_NOTIFY_REG_2) {
                        spe_sig_notify_2_area_t *sig = spectx->base_private->signal2_mmap_base;

                        sig->SPU_Sig_Notify_2 = data;
                } else {
                        errno = EINVAL;
                        return -1;
                }
                rc = 0;
        } else {
                if (signal_reg == SPE_SIG_NOTIFY_REG_1)
                        rc = write(_base_spe_open_if_closed(spectx,FD_SIG1, 0), &data, 4);
                else if (signal_reg == SPE_SIG_NOTIFY_REG_2)
                        rc = write(_base_spe_open_if_closed(spectx,FD_SIG2, 0), &data, 4);
                else {
                        errno = EINVAL;
                        return -1;
                }

                if (rc == 4)
                        rc = 0;

                if (signal_reg == SPE_SIG_NOTIFY_REG_1)
                        _base_spe_close_if_open(spectx,FD_SIG1);
                else if (signal_reg == SPE_SIG_NOTIFY_REG_2)
                        _base_spe_close_if_open(spectx,FD_SIG2);
        }

        return rc;
}
```

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

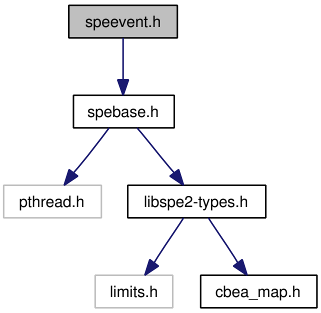
Parameters:

spectx Specifies the SPE context

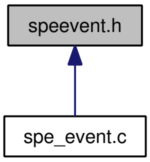
3.24 speevent.h File Reference

```
#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);
```

```
  __SPE_EVENT_CONTEXT_PRIV_SET(spe, NULL);

  close(evctx->stop_event_pipe[0]);
  close(evctx->stop_event_pipe[1]);

  pthread_mutex_destroy(&evctx->lock);
  pthread_mutex_destroy(&evctx->stop_event_read_lock);

  free(evctx);

  return 0;
}
```

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.

```
{
  spe_context_event_priv_ptr_t evctx;
  int rc;
  int i;

  evctx = calloc(1, sizeof(*evctx));
  if (!evctx) {
    return NULL;
  }

  rc = pipe(evctx->stop_event_pipe);
  if (rc == -1) {
    free(evctx);
    return NULL;
  }
  rc = fcntl(evctx->stop_event_pipe[0], F_GETFL);
  if (rc != -1) {
    rc = fcntl(evctx->stop_event_pipe[0], F_SETFL, rc | O_NONBLOCK);
  }
  if (rc == -1) {
    close(evctx->stop_event_pipe[0]);
    close(evctx->stop_event_pipe[1]);
    free(evctx);
    errno = EIO;
    return NULL;
  }

  for (i = 0; i < sizeof(evctx->events) / sizeof(evctx->events[0]); i++) {
    evctx->events[i].spe = spe;
  }

  pthread_mutex_init(&evctx->lock, NULL);
  pthread_mutex_init(&evctx->stop_event_read_lock, NULL);

  return evctx;
}
```

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 }
```

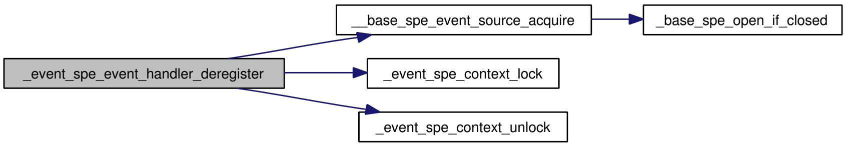
3.24.3.7 int _event_spe_event_handler_deregister (spe_event_handler_ptr_t *evhandler*, spe_event_unit_t ∗ *event*)

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) {
    fd = __base_spe_event_source_acquire(event->spe, FD_IBOX);
    if (fd == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }
    if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }
    evctx->events[__SPE_EVENT_OUT_INTR_MBOX].events = 0;
  }

  if (event->events & SPE_EVENT_IN_MBOX) {
    fd = __base_spe_event_source_acquire(event->spe, FD_WBOX);
    if (fd == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }
    if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }
    evctx->events[__SPE_EVENT_IN_MBOX].events = 0;
  }

  if (event->events & SPE_EVENT_TAG_GROUP) {
    fd = __base_spe_event_source_acquire(event->spe, FD_MFC);
    if (fd == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }
    if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }
    evctx->events[__SPE_EVENT_TAG_GROUP].events = 0;
  }

  if (event->events & SPE_EVENT_SPE_STOPPED) {
    fd = evctx->stop_event_pipe[0];
    if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }
    evctx->events[__SPE_EVENT_SPE_STOPPED].events = 0;
  }

  _event_spe_context_unlock(event->spe);

  return 0;
}
```

Here is the call graph for this function:


_event_spe_event_handler_deregister
__base_spe_event_source_acquire
_base_spe_open_if_closed
_event_spe_context_lock
_event_spe_context_unlock

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.

```
{
  int epfd;

  if (!evhandler) {
    errno = ESRCH;
    return -1;
  }

  epfd = __SPE_EPOLL_FD_GET(evhandler);
  close(epfd);

  free(evhandler);
  return 0;
}
```

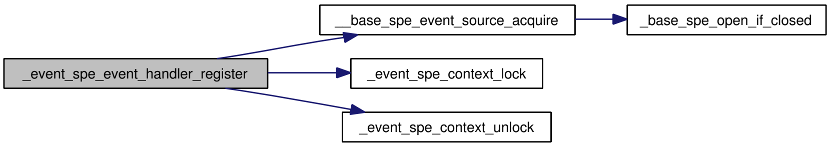
3.24.3.9 int _event_spe_event_handler_register (spe_event_handler_ptr_t *evhandler*, spe_event_unit_t ∗ *event*)

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.

```
{
  int epfd;
  const int ep_op = EPOLL_CTL_ADD;
  spe_context_event_priv_ptr_t evctx;
  spe_event_unit_t *ev_buf;
  struct epoll_event ep_event;
  int fd;

  if (!evhandler) {
    errno = ESRCH;
    return -1;
  }
  if (!event || !event->spe) {
    errno = EINVAL;
    return -1;
  }
  if (!__SPE_EVENTS_ENABLED(event->spe)) {
    errno = ENOTSUP;
    return -1;
  }

  epfd = __SPE_EPOLL_FD_GET(evhandler);
  evctx = __SPE_EVENT_CONTEXT_PRIV_GET(event->spe);

  if (event->events & ~__SPE_EVENT_ALL) {
    errno = ENOTSUP;
    return -1;
  }

  _event_spe_context_lock(event->spe); /* for spe->event_private->events */

  if (event->events & SPE_EVENT_OUT_INTR_MBOX) {
    fd = __base_spe_event_source_acquire(event->spe, FD_IBOX);
    if (fd == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }

    ev_buf = &evctx->events[__SPE_EVENT_OUT_INTR_MBOX];
    ev_buf->events = SPE_EVENT_OUT_INTR_MBOX;
    ev_buf->data = event->data;

    ep_event.events = EPOLLIN;
    ep_event.data.ptr = ev_buf;
    if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }
  }

  if (event->events & SPE_EVENT_IN_MBOX) {
    fd = __base_spe_event_source_acquire(event->spe, FD_WBOX);
    if (fd == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }

    ev_buf = &evctx->events[__SPE_EVENT_IN_MBOX];
    ev_buf->events = SPE_EVENT_IN_MBOX;
    ev_buf->data = event->data;

    ep_event.events = EPOLLOUT;
    ep_event.data.ptr = ev_buf;
    if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }
  }

  if (event->events & SPE_EVENT_TAG_GROUP) {
    fd = __base_spe_event_source_acquire(event->spe, FD_MFC);
    if (fd == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }

    if (event->spe->base_private->flags & SPE_MAP_PS) {
            _event_spe_context_unlock(event->spe);
            errno = ENOTSUP;
            return -1;
    }

    ev_buf = &evctx->events[__SPE_EVENT_TAG_GROUP];
    ev_buf->events = SPE_EVENT_TAG_GROUP;
    ev_buf->data = event->data;

    ep_event.events = EPOLLIN;
    ep_event.data.ptr = ev_buf;
    if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }
  }

  if (event->events & SPE_EVENT_SPE_STOPPED) {
    fd = evctx->stop_event_pipe[0];

    ev_buf = &evctx->events[__SPE_EVENT_SPE_STOPPED];
    ev_buf->events = SPE_EVENT_SPE_STOPPED;
    ev_buf->data = event->data;

    ep_event.events = EPOLLIN;
    ep_event.data.ptr = ev_buf;
    if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
      _event_spe_context_unlock(event->spe);
      return -1;
    }
  }

  _event_spe_context_unlock(event->spe);

  return 0;
}
```

Here is the call graph for this function:



_event_spe_event_handler_register
__base_spe_event_source_acquire
_base_spe_open_if_closed
_event_spe_context_lock
_event_spe_context_unlock

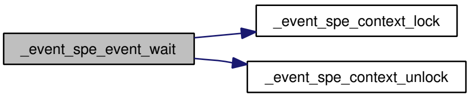
3.24.3.10 int _event_spe_event_wait (spe_event_handler_ptr_t *evhandler*, spe_event_unit_t ∗ *events*, int *max_events*, int *timeout*)

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.

```
{
  int epfd;
  struct epoll_event *ep_events;
  int rc;

  if (!evhandler) {
    errno = ESRCH;
    return -1;
  }
  if (!events || max_events <= 0) {
    errno = EINVAL;
    return -1;
  }

  epfd = __SPE_EPOLL_FD_GET(evhandler);

  ep_events = malloc(sizeof(*ep_events) * max_events);
  if (!ep_events) {
    return -1;
  }

  for ( ; ; ) {
    rc = epoll_wait(epfd, ep_events, max_events, timeout);
    if (rc == -1) { /* error */
      if (errno == EINTR) {
        if (timeout >= 0) { /* behave as timeout */
          rc = 0;
          break;
        }
        /* else retry */
      }
      else {
        break;
      }
    }
    else if (rc > 0) {
      int i;
      for (i = 0; i < rc; i++) {
        spe_event_unit_t *ev = (spe_event_unit_t *)(ep_events[i].data.ptr);
        _event_spe_context_lock(ev->spe); /* lock ev itself */
        events[i] = *ev;
        _event_spe_context_unlock(ev->spe);
      }
      break;
    }
    else { /* timeout */
      break;
    }
  }

  free(ep_events);

  return rc;
}
```

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.

```
{
  spe_context_event_priv_ptr_t evctx;
  int rc;
  int fd;
  size_t total;

  evctx = __SPE_EVENT_CONTEXT_PRIV_GET(spe);
  fd = evctx->stop_event_pipe[0];

  pthread_mutex_lock(&evctx->stop_event_read_lock); /* for atomic read */

  rc = read(fd, stopinfo, sizeof(*stopinfo));
  if (rc == -1) {
    pthread_mutex_unlock(&evctx->stop_event_read_lock);
    return -1;
  }

  total = rc;
  while (total < sizeof(*stopinfo)) { /* this loop will be executed in few cases */
    struct pollfd fds;
    fds.fd = fd;
    fds.events = POLLIN;
    rc = poll(&fds, 1, -1);
    if (rc == -1) {
      if (errno != EINTR) {
        break;
      }
    }
    else if (rc == 1) {
      rc = read(fd, (char *)stopinfo + total, sizeof(*stopinfo) - total);
      if (rc == -1) {
        if (errno != EAGAIN) {
          break;
        }
      }
      else {
        total += rc;
      }
    }
  }

  pthread_mutex_unlock(&evctx->stop_event_read_lock);

  return rc == -1 ? -1 : 0;
}
```

Index