Cell Broadband Engine Linux Reference Implementation Application Binary Interface Specification

Version 1.2

CBEA JSRE Series

Cell Broadband Engine Architecture Joint Software Reference Environment Series

August 22, 2007

Printed in the United States of America October 2007

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

August 22, 2007

Table of Contents

List of Figures

List of Tables

About This Document

This document describes particular features of the Cell Broadband Engine™ (CBE) Linux® Reference Implementation Application Binary Interface (ABI). Thus, it supplements the basic ABI specifications cited in "Related Documentation". If the material in this document differs from the *SPU Application Binary Interface Specification,* the content specified here takes precedence.

Audience

This document is intended for system and application programmers who develop language processors and other software tools for the Cell Broadband Engine.

Version History

This section describes significant changes made to the *CBE Linux Reference Implementation ABI Specification* for each version of the document.

Version Number & Date	Changes
v 1.2 August 22, 2007	Modification corresponding to: • LWG_RFC0008 – add opcodes for externally assisted SPE library calls • LWG_RFC0025 – isolation mode errors and stop-and-signal types • LWG_RFC0028 – add additional PPE assisted POSIX opcodes
v. 1.1 November 27, 2006	Modifications corresponding to: • LWG_RFC0006 - special symbol mangling, the _EAR_ symbol • LWG_RFC0007 - .toe section type change
v. 1.0 November 9, 2005	Initial release of this document.

Related Documentation

The following documents provide basic specifications that apply to the Cell Broadband Engine:

PowerPC® Microprocessor Family: The Programming Environments Manual for 64-Bit Microprocessors

PowerPC Microprocessor Family: The Programming Environments Manual for 32-Bit Microprocessors

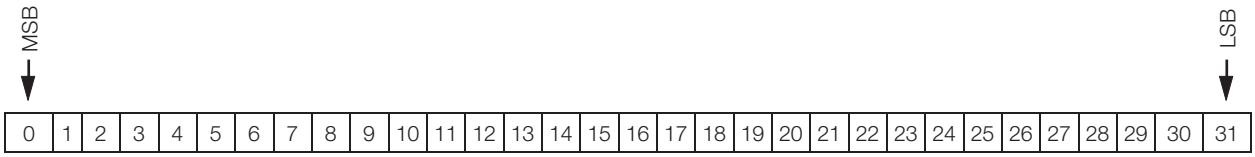
SPU Application Binary Interface Specification

You can find additional specifications in the *Linux Standard Base Specification for the PPC32 and PPC64 Architectures*. For details about the Linux specification, see http://www.linuxbase.org/spec.

Bit Notation and Typographic Conventions Used in This Document

Bit Notation

Standard bit notation is used throughout this document. Bits and bytes are numbered in ascending order from left to right. Thus, for a 4-byte word, bit 0 is the most significant bit and bit 31 is the least significant bit, as shown in the following figure:



MSB = Most significant bit
LSB = Least significant bit

Notation for bit encoding is as follows:

- Hexadecimal values are preceded by `0x`. For example: `0x0A00`.
- Binary values in sentences appear in single quotation marks. For example: ‘1010’.

Other Typographic Conventions

In addition to bit notation, the following typographic conventions are used throughout this document:

Convention	Meaning
`courier`	Indicates programming code, processing instructions, register names, data types, events, file names, and other literals. Also indicates function and macro names. This convention is only used where it facilitates comprehension, especially in narrative descriptions.
`courier + italics`	Indicates arguments, parameters, and variables, including variables of type `const`. This convention is only used where it facilitates comprehension, especially in narrative descriptions.
italics (without courier)	Indicates emphasis. Except when hyperlinked, book references are in italics. When a term is first defined, it is often in italics.
blue	Indicates a hyperlink (color printers or online only).

1. Introduction

This document defines a standard interface that allows compiled objects to be linked and run on the Linux Reference Implementation for the Cell Broadband Engine without recompilation or recoding. The purpose of this capability is to achieve greater application portability.

2. CBEA Embedded SPE Object Format (CESOF)

The Cell Broadband Engine Architecture (CBEA) Embedded Synergistic Processor Element (SPE) Object Format (CESOF) contains three layers. These layers enable an SPE Executable and Linking Format (ELF) executable to participate in the program linking and execution with other PowerPC Processor Element (PPE) ELF objects. The first layer is a new special section, the table of effective-address (.toe) references section, in the SPE relocatable format. The second layer is a segment containing the .toe sections in the SPE executable format. The third layer is the CESOF layout and data structure that embeds the SPE executable in a PPE relocatable. This document describes how CESOF runtime support can interact with the structures in the CESOF object.

2.1. The .toe Section in the SPE Relocatable Object

The CESOF introduces a new special SPE ELF section, the .toe section, in the SPE ELF relocatable object. It provides a space to keep the effective-address references (EARs) used by the SPE program. This section plays the central role in extending the PPE symbol space into the SPE symbol space.

Contained in the .toe section is an array of effective-address references from the giving SPE ELF relocatable object. Each effective-address reference is a 16-byte structure that can be specified by the following C structure:

```
typedef struct elf_toe_entry {
       Elf64_Addr   ea_value;
       Elf64_Addr   ea_info;     /* reserved */
} EAR __attribute__ ((aligned(16)));
```

The EAR structures are quadword (16-byte) aligned for faster load-and-store operations into a preferred slot of an SPE register. The structure members are purposely kept the same for both 32-bit and 64-bit PPE runtime environments. This allows the `ea_value` to be handled consistently as a 64-bit value in the SPE program. The same SPE binary file can be reused in both 32-bit and 64-bit environments. A programmer still has the option to cast the `ea_value` into a 32-bit effective-address when the SPE program is optimized only for a 32-bit environment.

The EAR structure has an 8-byte space, `ea_info`, which is reserved for future use. In particular, the SPE program loader can fill in EAR entry-related information for the system runtime environment.

All the EAR structures in the same SPE object are placed consecutively in an array and placed in this special .toe section. This array is usually generated by a compiler or hand-crafted by a programmer using an assembler.

Listed below are the values of a typical .toe section header. It is usually generated by an assembler or compiler back-end:

```
sh_name:     index into the .toe in the .shstrtab section
sh_type:     SHT_NOBITS
sh_flags:    SHF_ALLOC
sh_addr:     0
sh_offset:   the file offset to this section
sh_size:     the number of EAR entries times 16 bytes
sh_link:     SHN_UNDEF
sh_info:     0
sh_addralign:0x10 (16)
sh_entsize:  0x10 (16)
```

2.2. The toe Segment in an SPE Executable Object

After the SPE objects are linked into the SPE executable image, all .toe sections are collected into a single loadable segment as shown in Figure 2-1. An SPE linker groups all .toe sections consecutively into a loadable segment, `PT_LOAD`, in the SPE executable. There is no gap between the adjacent sections in the segment. From the size of the section (that is, the EAR array), the tool can determine the number of EAR entries in this object file. The number of EAR entries is only limited by the available physical local store at runtime.

A .toe section should not exist when there is no effective-address reference. Thus, some linked SPE executables might not have a toe segment if one is not needed. When this loadable segment exists, it is aligned to a 128-byte boundary for better DMA efficiency by the SPE program loader. No sections of other names can be included in this segment. The segment is given a permission of `PF_R`.

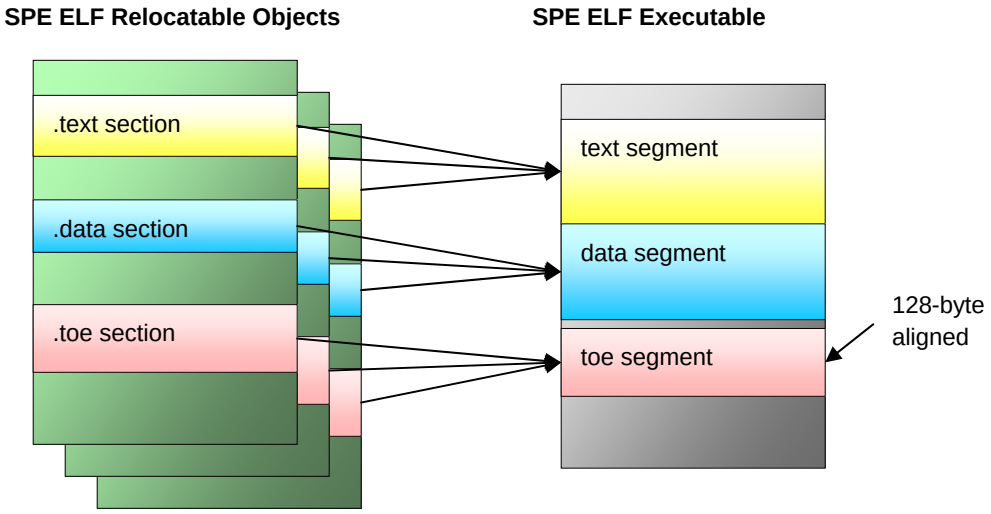
`p_filesz` of the segment is zero. `p_mmseze` is the actual array size allocated for the EAR entries.

It is a linking error when more than one EAR definition of symbols has the same name. Those SPE relocatable objects accessing the same effective-address memory object must share the same EAR entry. Another restriction is that a non-EAR symbol must not use the signature "_EAR_".

In the final SPE executable, the symbol table section for the EAR variables in the .toe sections must not be stripped off. The CESOF wrapping tool needs the symbol information to create additional elements in the wrapping layer.

The initial values of the EAR entries in the object file are not defined. The SPE loader runtime environment updates the resolved values by overwriting the whole segment before the SPE program starts.

Figure 2-1: SPE toe Segment



2.3. Symbol Mangling

The name of the EAR symbol in the local-store space is mangled from the symbol name of the memory object in the effective-address space. The name-mangling scheme establishes the association between two symbols in two different symbol spaces referring to the same global memory object. (Note that an EAR symbol contains only effective-address information and no type information about the referenced memory object. It cannot be de-referenced directly back into an object in the local store without additional compiler runtime support. Such support involves DMA operations between the effective-address space and the local-store space.)

The following C code-segment in an SPE program illustrates a declaration style of EAR entries in a C program, if its compiler supports the section attribute.

```
// EAR entry for the global object g_mem_obj_1
const EAR _EAR_g_mem_obj_1 __attribute__((section (“.toe”))) = {0};

// EAR entry for the global object g_mem_obj_2
const EAR _EAR_g_mem_obj_2 __attribute__((section (“.toe”))) = {0};
```

If the compiler does not support the special section, a separate piece of assembly code can similarly define these EAR entries as followed:

```
.section .toe
.aligned 4
_EAR_g_mem_obj_1:
                            .quad 0
                            .quad 0
_EAR_g_mem_obj_2:
                            .quad 0
                            .quad 0
```

The use of the effective-address reference symbol is illustrated by the following SPE DMA operation that copies the content of the global memory object from the effective-address space into the local store. (Note: An SPE compiler runtime environment can use a similar DMA operation to support the de-referencing of an EAR object in the local store.)

```
extern EAR _EAR_g_mem_obj_1;

spu_mfcdma64(ls_buf, (_EAR_g_mem_obj_1.ea_value >> 32) & 0xFFFFFFFF,
       _EAR_g_mem_obj_1.ea_value & 0xFFFFFFFF,
       size_of_the_actual_global_memory_object,
       0,
       MFC_GET_CMD);
```

The values of the `ea_value` fields are not initialized to any value in the SPE executable image. The CESOF runtime environment updates the actual effective-address values after the SPE executable image is loaded into the local store. See Section 2.4.4 CESOF Runtime Support for a description of the updating mechanism.

Since the mangling algorithm uses the same signature for all the EAR symbols, a tool can distinguish the symbols for the effective-address reference from those for the local-store space. It is a restriction that no other symbols may use names containing this signature.

Below is a typical symbol table entry for an effective-address reference. It is the same as those generated by a compiler or an assembler. Rather than making changes to a linker to generate such

entries, it is simpler and less error-prone to generate assembly code and allow an assembler to generate the section and its associated symbol table entries.

```
st_name:     index “_EAR_<effective_address_obj_name>” in .strtab
st_value:    offset to this EAR (multiple of 16 bytes)
st_size:     8 (64-bit value)
st_info:     (STB_GLOBAL | STT_OBJECT)
st_other:    0
st_shndx:    index to the .toe section
```

2.3.1. Special Symbol Mangling

These symbols establish an association between a symbol in SPE symbol space and a well-known effective address in main storage. These symbols do not mangle the name of an EAR symbol in local-storage space to a symbol name of a memory object in the effective-address space.

2.3.1.1. The _EAR_Symbol

The _EAR_ symbol (without any <effective_address_object_name> suffix) associates this symbol in SPE symbol space and the effective address of the start of the SPE executable image in main storage. This symbol allows an SPE program, such as a kernel or overlay manager, to operate on a program in which it is linked.

The following C code-segment in an SPE program illustrates a declaration style of EAR entries in a C program, if its compiler supports the section attribute.

```
// EAR entry for the starting address of SPE executable image
const EAR _EAR_ __attribute__((section (".toe"))) = {0};
```

If the compiler does not support the special section, a separate piece of assembly code can similarly define these EAR entries as follows:

```
.section     .toe
.aligned     4
_EAR_:
             .quad 0
             .quad 0
```

2.4. CESOF Object Layout

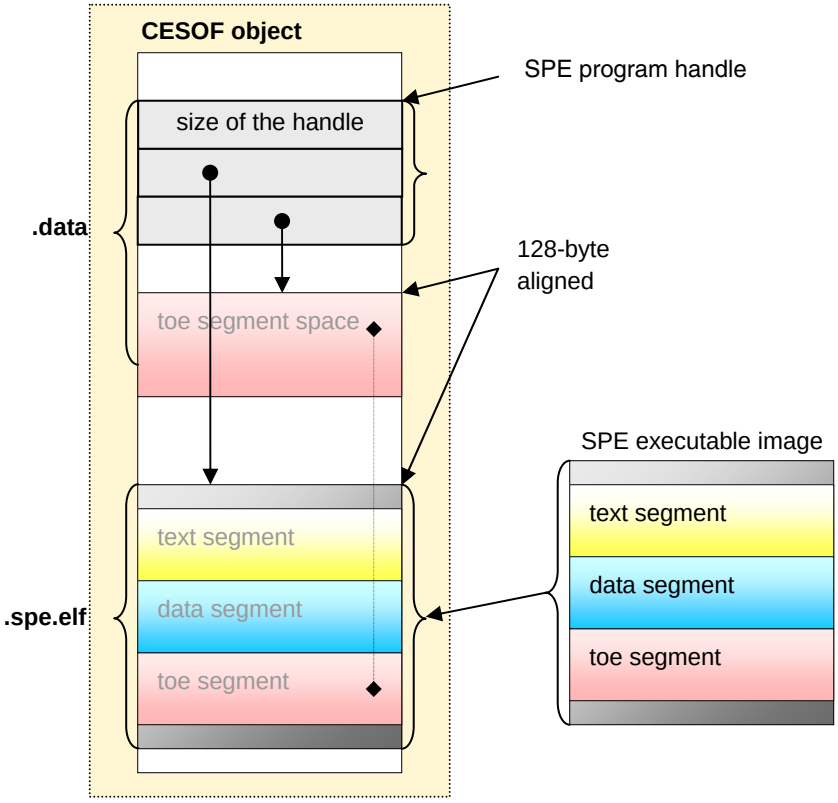
A CESOF object is itself an instance of a PPE ELF relocatable object. It encloses the entire SPE executable image as a special section along with an additional wrapping layer to enable the linking and execution process against other PPE ELF objects.

Instead of operating on the SPE executable image directly, the PPE toolchain operates only on the elements in the wrapping layer. This shields the PPE toolchain from linking into a different architectural space. The wrapping layer holds the linking and execution result, whereas the CESOF runtime environment later updates the result into the SPE executable image before an SPE program is executed.

The CESOF object layout of the wrapping elements is illustrated in Figure 2-2. The wrapping layer contains additional elements for linking and execution with other PPE ELF objects.

Figure 2-2 shows three new elements enclosed in a CESOF object: the embedded image of an SPE ELF executable, a shadow section for the SPE toe segment, and a structure, called the SPE program handle, which enables programs to access the previous two elements. The following sections provide detailed specifications.

Figure 2-2: CESOF Layout



2.4.1. The Embedded SPE ELF Executable Image

The SPE executable image is included in a special section called `.spe.elf` in the CESOF object — a particular operating system toolchain is allowed to rename this section to a name (such as `.rodata.speelf`) that is consistent with the section naming convention of its linker. We use `.spe.elf` in this document to refer this special section. To share a single copy of the embedded SPE ELF executable image with several processes in an operating system, the section should be linked into a per-system segment when shared (for example, the text segment of a shared library).

However, the CESOF specification also allows the section to be included in any other segment (for example) the data segment based on the need of the target operating system.

For this image, a local symbol is defined that the SPE program handle (see Section 2.4.3) can reference. This image is aligned to a DMA-boundary (128-byte) for efficient DMA operation.

Its section header must contain the following values:

```
sh_name:      index into the ".spe.elf" in .shstrtab section
sh_type:      SHT_PROGBITS
sh_flags:     SHF_ALLOC
sh_addr:      0
sh_offset:    the file offset to this section
sh_size:      the size of the raw executable image
sh_link:      SHN_UNDEF
sh_info:      0
sh_addralign:0x80 (128)
sh_entsize:   0
```

2.4.2. The Shadow Section for the SPE toe Segment

A new section is introduced in the wrapping layer to enable the linking and resolution of the effective-address references of the embedded SPE executable. Its main function is to shadow the toe segment of the embedded SPE executable image so that a PPE linker does not need to directly modify the embedded image. This separation allows the embedded SPE executable image to become position-independent and thus sharable by different processes.

An SPE relocatable object keeps its EAR entries in its .toe section. When the SPE linker links several SPE relocatable objects into the SPE executable, it joins all .toe sections into a toe segment. Subsequently, the whole SPE executable image is included in the `.spe.elf` section of a CESOF object. Later, when the CESOF object is linked with other PPE relocatable objects, its `.spe.elf` section should be included in a per-system loadable read-only segment of a linked PPE executable or PPE shared library.

In the case of a shared library, the per-system read-only segment is sharable among different processes in the operating system. In order to maintain its position independency, constructs similar to a global offset table must be used. The shadow section serves a similar purpose for the embedded toe segment. The embedded SPE executable image will remain in the per-system read-only segment, whereas the shadow section is mapped to a memory region that is private for a specific processor.

This shadow section is of the same size and alignment as the toe segment of the embedded SPE executable image. An effective-address reference entry inside the toe segment has the same space and location offset in the shadow section.

The entries in the shadow sections are undefined PPE symbol references that a PPE linker can resolve in the linking process. Because the original symbol names of the EARs used in the SPE executable are mangled with the special signature, the wrapping tool demangles the symbol names.

The demangled symbol names reside in the effective-address symbol space for global memory objects.

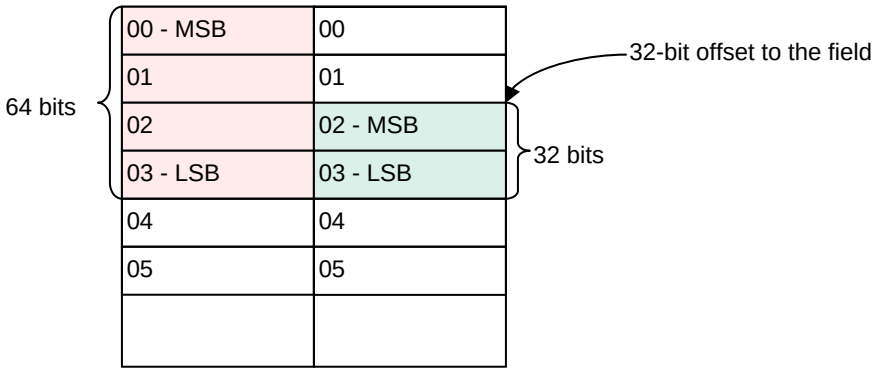
Updating the runtime image with the resolved references is straightforward after both the PPE and SPE images are loaded into the system memory and the SPE local-store respectively. Because the shadow section collects all the resolved references, a simple DMA operation from the effective address of the resolved shadow section to the local-store address of the uninitialized SPE toe segment is sufficient to update all the relocated values into the SPE runtime environment.

This shadow section may be included as a `.data` or `.data.spetoe` section depending on its linker naming convention and should be mapped into a per-process space. Like the shadowed toe segment, this section is aligned to a DMA boundary for optimal DMA efficiency. Its section header contains the following values:

```
sh_name:     index into the ".data" in .shstrtab section
sh_type:     SHT_PROGBITS
sh_flags:    SHF_ALLOC
sh_addr:     0
sh_offset:   the file offset to this section
sh_size:     the size of the embedded toe segment
sh_link:     SHN_UNDEF
sh_info:     0
sh_addralign:0x80 (128)
sh_entsize:  0x10 (16)
```

For each SPE EAR entry in the toe segment, a wrapping tool must create a corresponding undefined symbol in the shadow section with the symbol name properly demangled. In the case of big-endian, the relocation entry offset is different between the 64-bit and 32-bit environments as shown in Figure 2-3. A wrapping tool must properly handle such differences.

Figure 2-3: 64-Bit Environment versus 32-Bit Environment



Instead of providing an exact specification of the relocation entries for the shadow section, it is clearer to illustrate it by using the following assembly code.

The SPE .toe segment can be defined in the SPE assembly code in this manner:

```
.section .toe, “a”, @progbits
.align 4
_EAR_symbol_name_1:
       .quad 0      // EAR entry
       .quad 0
_EAR_symbol_name_2:
       .quad 0      // EAR entry
       .quad 0
… …
```

Below is the PPE assembly code of the shadow section for a 64-bit PPE environment.

```
.section .data, “a”, @progbits
.align 7
.extern symbol_name_1     // de-mangled symbol name
       .quad symbol_name_1
       .quad 0

       .extern symbol_name_2
       .quad symbol_name_2
       .quad 0
… …
```

Below is the shadow section for a 32-bit PPE (big-endian) environment.

```
.section .data, “a”, @nobits
.align 7
.extern symbol_name_1
       .int 0 // offset for 32-bit env.
       .int symbol_name_1
       .quad 0
.extern symbol_name_2
       .int 0
       .int symbol_name_2
       .quad 0
… …
```

If a wrapping tool generates the assembly code above, then a PPE assembler can generate directly the proper relocation entries.

2.4.3. The SPE Program Handle Structure

A handle structure is defined in the CESOF object to allow a PPE programmer to access the SPE program image. It also allows the supporting runtime environment to locate the embedded SPE executable image and the resolved shadowed toe segment. If the CESOF object is linked as shared, the actual content of the handle structure will be relocated only after a process image is properly mapped in the effective-address space. Two symbols, local to each CESOF object, are defined for the embedded SPE executable image and the shadow section respectively: `_spe_elf_image` and `_spe_toe_shadow`.

Here is the structure of the SPE program handle in both 32-bit and 64-bit environments:

32-bit environment:

```
typedef struct spe_program_handle {
       int    handle_size; // size of(spe_program_handle_t)
       void*  elf_image;          // pointer to the embedded SPE image
       void*  toe_shadow;         // pointer to the shadowed toe
} spe_program_handle_t;
```

64-bit environment:

```
typedef struct spe_program_handle {
       int    handle_size;
       int    padding;
       void*  elf_image;
       void*  toe_shadow;
} spe_program_handle_t;
```

The 32-bit handle is 4-byte aligned, and the 64-bit handle is 8-byte aligned.

A 32-bit wrapping tool generates the following code to construct the handle structure:

```
.global spe_program_handle

.align 2
_spe_program_handle:
       .int         12

       .int         _spe_elf_image
       .int         _spe_toe_shadow

.section .spe.elf, “a”, @progbits
.align 7
_spe_elf_image:
       .incbin      “spe_executable_name”

.section .data, “a”, @progbits
.align 7
_spe_toe_shadow:
.extern symbol_name_1
       .int   0 // offset for 32-bit env.
       .int   symbol_name_1
       .quad  0
.extern symbol_name_2
       .int   0
       .int   symbol_name_2
       .quad  0
… …
```

The CESOF does not specify the symbol name associated with the SPE program. A programmer can specify the symbol name by using the wrapping tool.

A 64-bit layout can be similarly illustrated:

```
.global spe_program_handle

.align 3
_spe_program_handle:
       .int   24
       .int    0
       .quad  _spe_elf_image
       .quad  _spe_toe_shadow

.section .spe.elf, “a”, @progbits
.align 7
_spe_elf_image:
       .incbin “spe_executable_name”

.section .data, “a”, @progbits
.align 7
_spe_toe_shadow:
.extern symbol_name_1
       .quad  symbol_name_1
       .quad  0
.extern symbol_name_2
       .quad  symbol_name_2
       .quad  0
… …
```

2.5. CESOF Runtime Support

Several CESOF runtime supports carry out the proper relocation and execution of CESOF objects. The runtime supports use the SPE program handle to obtain the needed information for the embedded SPE executable.

Loading the embedded SPE executable image into the system memory

Because the embedded SPE executable image is grouped in the loadable segment together with other PPE code or data sections, loading the SPE executable image into the system memory is a direct result of loading the PPE segments.

In the case where a CESOF object is linked in a shared library, the operating system loads only one copy of the per-system segment image in the shared library into the system memory. Hence, the embedded SPE executable image appears only once in the system memory.

Locating the SPE executable image

After the SPE executable image is mapped into the system memory, the CESOF runtime environment accesses the image through the SPE program handle structure. The structure contains two pointers pointing to the embedded SPE executable image and the shadow section of its toe segment. Depending on the operating system environment, the pointers in the handle structure can either be a 32-bit or a 64-bit pointer value. The structure is included here again.

```
Typedef struct spe_program_handle {
       int    handle_size; // sizeof(spe_program_handle_t)
       void*  elf_image;          // pointer to the embedded SPE image
       void*  toe_shadow;         // pointer to the shadowed toe
} spe_program_handle_t;
```

A symbol is defined for this handle structure by the wrapping tool and is passed on to the CESOF runtime environment by the programmer. Here is an example of its application:

```
extern spe_program_handle_t spe_foo_handle;
…
spe_pid = spe_create_thread(0, &spe_foo_handle, 0, NULL, -1, 0);
…
```

Loading the SPE executable image

The SPE executable image is an ELF executable object. Its ELF header provides the information for all the SPE segments. The SPE loader uses the information to load the segments from the system memory into the local store of the SPE.

The segments can be copied into the local store by using the proper DMA operations. The DMA operations can be initiated either by the PPE runtime environment or by a bootstrapping loader on the SPE. In either case, parsing the ELF header of the SPE executable image is necessary.

The segments of the SPE executable image are aligned to DMA boundaries; this enables DMA operations to efficiently copy the needed image to the SPE local-store space.

Loading the resolved toe segment of an SPE executable image

When the SPE loader copies the segments of the SPE executable into the local store, it does not copy the SPE toe segment from the executable because the segment is not relocated by the PPE linker. Instead, the loader loads the shadow section of the CESOF object whose entries are properly relocated by the PPE linker. The shadow section should be mapped to a memory region that is private for a specific processor.

The system memory location of this shadow section is obtained from its SPE program handle. Its size and target local-store address are obtained from the ELF header of the SPE executable image.

Once the shadow section for the toe segment is copied into the local store, the SPE program is ready for execution with the properly relocated effective-address references. It does not matter if the CESOF object is statically or dynamically linked; this loading mechanism works in both cases.

3. SPE Execution Environment

This chapter specifies the low-level system information and the conventions adopted in the SPE execution environment that is made available to Synergistic Processor Element (SPE) programs by the operating system.

3.1. SPE Program Initialization

This section describes the machine state that the SPE loader creates for the SPE program execution, including argument passing, register usage, and stack frame layout. Although this section does not describe C program initialization, it gives the information necessary to implement the call to the entry point of the SPE program.

3.1.1. Standard Environment

Programming language systems use the initial program state to establish a standard environment for their application programs. For example, the C interface for the SPE's entry point is conventionally declared as follows:

```
extern int
main(unsigned long long spe_id, unsigned long long param,
     unsigned long long env);
```

In this declaration, *spe_id* is a unique SPE task identifier, *param* is a system memory address to application parameters, and *env* is a system memory address of runtime environment information.

3.1.2. Registers

When the SPE program is first entered, the contents of registers other than those listed below are unspecified. For security purposes, unspecified registers may be cleared when loading SPE programs of a different process or execution domain. However, a program that requires registers to have specific values must set them explicitly during process initialization. The program should not rely on the loader to set any register other than those shown in Table 3-1.

Table 3-1: Initial SPE Register Contents

Register	Contents
1	Top of the stack, as specified in the Program Initialization section of *SPU Application Binary Interface Specification,* version 1.3
2	Runtime stack size. This 32-bit unsigned integer is typically saved by the startup code in `crt0` and used to check runtime stack overflow.
3	SPE task identifier. This is a 64-bit unsigned integer.
4	System memory pointer. This is a 64-bit pointer to application-defined program parameters.
5	System memory pointer. This is a 64-bit pointer to the SPE task environment structure. The contents of the structure are implementation defined.

3.2. SPE Stop-and-Signal Signal Types

The `stop` instruction contains a 14-bit signal type. When executed, this value is written to bits 0 to 13 in the SPU Status register. These bits are used to specify why a program stopped. The signal types are partitioned as follows:

- Signal types with the most significant bit of '0' are reserved for application use.
- Signal types with the most significant bit of '1' are reserved for runtime or privileged services.

Table 3-2 describes the reserved signal types.

Table 3-2: Reserved Signal Types

Signal Type	Description
0x0000	Data executed as an instruction.
0x2000 - 0x20FF	Return from main or exit. Return or exit status is encoded in the least significant byte of the stop and signal types. `exit(EXIT_SUCCESS) == 0x2000`. `exit(EXIT_FAILURE) == 0x2001`.
0x2100 - 0x21FF	Externally assisted PPE library calls. See Section 3.3.2 "Standardized Library Classes and Call Opcodes" for additional details.
0x2200 – 0x220F	SPE isolation mode errors.
0x3FFE	Stack overflow detected.
0x3FFF	Debugger breakpoint.

3.3. Externally Assisted SPE Library Calls

Externally assisted SPE library calls are those functions that cannot be fully serviced by the SPE and require the PowerPC Processor Unit (PPU) to assist in their execution since the operating system runs in the PPE. The calls are referred to as assisted calls throughout the remainder of the document.

3.3.1. Performing an Assisted Call

To perform an assisted call, the SPE must:

- Construct a local-store memory image as the union of the input and output parameters.
- Copy all input parameters from the registers or the stack into the input/output memory image. Each parameter is padded to a quadword boundary. For example, consider function copy.

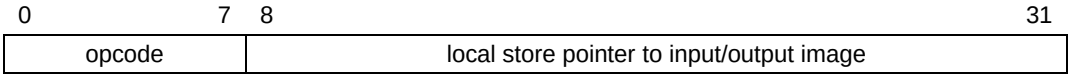
  void * copy(void * dest, void * src, size_t n)

  The parameter image would be:

0	void * dest	pad	pad	pad
16	void * src	pad	pad	pad
32	size_t n	pad	pad	pad

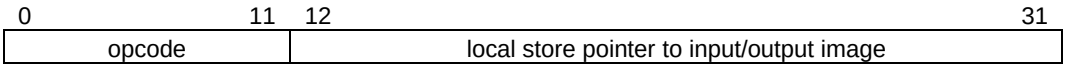
- Combine the pointer to the local-store input/output memory image with the specific library function opcode to form a 32-bit message. Table 3-3 shows the bit layout for the standard assisted call messages.

  Table 3-3: 32-Bit Assisted Call Message



- Operating-system-dependent system call messages are free to define a different allocation of opcode bits and local-store pointer bits to support a larger set of opcodes. For example, the opcode could be defined to be 12 bits with a 20-bit local-store pointer as shown in Table 3-4.

Table 3-4: Operating-System-Dependent syscall Message



- Place the message into the local-store memory immediately following a `stop` instruction. The signal type of the `stop` instruction is chosen according the classification of the library function being performed.
- Signal the PPE to perform the library function on behalf of the SPE by executing the `stop and signal` instruction.

To service the assisted call, the PPE must:

- Get the assisted call message from the 32-bit local-store word pointed to by the SPE's next program counter (NPC). Remember to strip off the least significant bit, the Interrupt Enable bit, of the NPC when determining the address of the assisted call message.
- Increment the NPC by 4 bytes.
- Dispatch control to the indicated assisted call handler based upon the stop and signal type and opcode specified by the message.
- Get the assisted call parameters from the local-store memory image pointed to by the pointer contained within the assisted call message.
- Perform the requested assisted library call.
- Place all return values into the local-store memory image pointed to by the pointer contained within the assisted call message. Assisted calls that generate errors, that by standard set the error return value, *errno,* return the value of errno in word element 3 of the quadword return value.
- Resume SPE execution at the new NPC.

To complete the assisted call, the SPE must:

- Get the return values from the memory image and place them into the return registers in accordance with the *SPU Application Binary Interface Specification*.
- If the assisted call is specified to set errno, the return value is tested to determine if an error occurred. If so, word element 3 of the return value's quadword is written to the local store's errno variable.

Here is an example that shows how to implement the SPE `fopen` external assisted call.

```
extern FILE * fopen(const char *path, const char *mode);
       # Position independent fopen assisted call
```

```
        .extern errno   # assumed aligned on 3rd word element
        .text
        .align 2
        .global fopen
fopen:
        # Place input parameters onto the stack to form the
        # local store memory image.
        stqd    $3, -32($sp)
        stqd    $4, -16($sp)

        # Construct a message consisting of the 8-bit opcode
        # and 24-bit local store pointer to the input
        # parameters and place it following the stop and signal
        # instruction.
        ila     $2, 0x3FFFF     # address mask
        ilhu    $3, FOPEN_CMD << 8
        ai      $4, $sp, -32    # parameter pointer
        selb    $3, $3, $4, $2  # combine command & address ptr
        brsl    $2, next        # inst addr for pic addressing
next:
        lqr     $4, message
        cwd     $2, message-next($2)
        shufb   $3, $3, $4, $2  # insert msg into inst qword
        stqr    $3, message     # store cmd/ptr into msg word

        dsync

        # Notify the PPE to perform the assisted call request
        # by issuing a stop and signal with a signal code of
        # 0x2100 (C99 class)
        stop    0x2100

message:
        .word   0

        # Fetch returns value from local store memory image
        # and return in R3. If the return value is 0, then
        # store the return errno into the local errno variable
        lqd     $3, -32($sp)
        lqa     $4, errno
        ceqi    $5, $3, 0
        rotqmbyi $5, $5, -12
        selb    $4, $4, $3, $5
        stqa    $4, errno
        bi      $0              # return to caller
```

Here are two C-language examples that show how implement the SPE external assisted calls for `fopen` (fixed-length argument list) and `vfprintf` (variable-length argument list).

```
#define SPE_C99_SIGNALCODE       0x2100
#define SPE_C99_FOPEN            10
#define SPE_C99_VFPRINTF         35

void __send_to_ppe(unsigned int signalcode, unsigned int opcode, void *data)
{
  unsigned int combined = ((opcode << 24) | (data & 0x00FFFFFF));
  vector unsigned int stopfunc = {
    signalcode,           /* stop */
```

```
    combined,              /* call message */
    0x4020007F             /* nop */
    0x35000000             /* bi $0 */
  }
  void (*f)(void) = (void *)&stopfunc;
  asm("sync");
  f();
  errno = ((unsigned int *)data[3];
)

typedef struct {
  char *fp;
  unsigned int pad0[3];
  char *mode;
  unsigned int pad1[3];
} c99_fopen_t;

FILE *fopen(const char *file, const char *mode)
{
  FILE **ret;
  c99_fopen_t args;
  args.fp = file;
  args.mode = mode;
  ret = (FILE **) &args;
  __send_to_ppe(SPE_C99_SIGNALCODE, SPE_C99_FOPEN, &args);
  return *ret;
}

typedef struct {
  FILE *fp;
  unsigned int pad0[3];
  char *fmt;
  unsigned int pad1[3];
  va_list ap;
 } c99_vfprintf_t;

 int vfprintf(FILE * fp, const char *fmt0, va_list ap)
 {
   int *ret;
   c99_vfprintf_t args;
   ret = (int *) &args;
   args.fp = fp;
   args.fmt = (char *) fmt0;
   va_copy(args.ap, ap);
   __send_to_ppe(SPE_C99_SIGNALCODE, SPE_C99_VFPRINTF, &args);
   return *ret;
 }
```

This fprintf function is illustrative of how to implement functions with variable arguments.

```
 int fprintf(FILE * f, const char *fmt, ...)
 {
   int ret;
   va_list fprintf_list;
   va_start(fprintf_list, fmt);
   ret = vfprintf(f, fmt, fprintf_list);
   va_end(fprintf_list);
   return ret;
```

```
}
```

3.3.2. Standardized Library Classes and Call Opcodes

The assisted library calls are classified according to the standard in which they are specified. Each class is assigned a unique `stop and signal` type.

Table 3-5: Library Classes

Stop type	Standard
0x2100	ISO/IEC C Standard 9899:1999 (C99)
0x2101	POSIX.1 (IEEE Standard 1003.1)
0x2102	POSIX.1b
0x2103	Operating-System-Dependent system calls

The opcodes for each library class are assigned and registered as needed. The registry of opcodes is done through this specification and additions to the registry require a request-for-change (RFC). If an unregistered opcode is called, the SPE program is stopped, see the SPE Runtime Management Library chapter on PPE-assisted library facilities for details on the handling of unregistered callbacks.

Table 3-6: Registered ISO/IEC C Standard 9899:1999 (C99) opcodes

Opcode	Function Prototype
1	void clearerr(FILE *stream)
2	int fclose(FILE *stream)
3	int feof(FILE *stream)
4	int ferror(FILE *stream)
5	int fflush(FILE *stream)
6	int fgetc(FILE *stream)
7	int fgetpos(FILE *stream, fpos_t *pos)
8	char *fgets(char *s, int size, FILE *stream)
9	int fileno(FILE *stream)
10	FILE *fopen(const char *path, const char *mode)
11	int fputc(int c, FILE *stream)
12	int fputc(int c, FILE *stream)
13	size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)
14	FILE *freopen(const char *path, const char *mode, FILE *stream)
15	int fseek(FILE *stream, long offset, int whence)
16	int fsetpos(FILE *stream, fpos_t *pos)
17	long ftell(FILE *stream)
18	size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)
19	int getc(FILE *stream)
20	int getchar(void)
21	char *gets(char *s)
22	void perror(const char *s)
23	int putc(int c, FILE *stream)
24	int putchar(int c)
25	int puts(const char *s)
26	int remove(const char *pathname)
27	int rename(const char *oldpath, const char *newpath)

Opcode	Function Prototype
28	void rewind(FILE *stream)
29	void setbuf(FILE *stream, char *buf)
30	int setvbuf(FILE *stream, char *buf, int mode , size_t size)
31	int system(const char *command)
32	FILE *tmpfile (void)
33	char *tmpnam(char *s)
34	int ungetc(int c, FILE *stream)
35	int vfprintf(FILE *stream, const char *format, va_list ap)
36	int vfscanf(FILE *stream, const char *format, va_list ap)
37	int vprintf(const char *format, va_list ap)
38	int vscanf(const char *format, va_list ap)
39	int vsnprintf(char *str, size_t size, const char *format, va_list ap)
40	int vsprintf(char *str, const char *format, va_list ap)
41	int vsscanf(const char *str, const char *format, va_list ap)

Note: All pointers are SPU local addresses, and FILE * are unsigned ints. The definition of the data structures and types are specific to the SPU ABI and should not be assumed to match the ABI that is used by the host operating system.

Table 3-7: Registered POSIX.1 (IEEE Standard 1003.1) opcodes

Opcode	Function Prototype
1	int adjtimex(struct timex *buf)
2	int close(int fd)
3	int creat(const char *pathname, mode_t mode)
4	int fstat(int filedes, struct stat *buf)
5	key_t ftok(const char *pathname, int proj_id)
6	int getpagesize(void)
7	int gettimeofday(struct timeval *tv, struct timezone *tz)
8	int kill(pid_t pid, int sig)
9	off_t lseek(int fildes, off_t offset, int whence)
10	int lstat(const char *path, struct stat *buf)
11	void *[<1>]mmap(void *start[<1>], size_t length, int prot, int flags, int fd, off_t offset)
12	void *[<1>]mremap(void *old_address[<1>], size_t old_size , size_t new_size, unsigned long flags)
13	int msync(void *start[<1>], size_t length, int flags)
14	int munmap(void *start[<1>], size_t length)
15	int open(const char *pathname, int flags, mode_t mode)
16	ssize_t read(int fd, void *buf, size_t count)
17	void *shmat(int shmid, const void *shmaddr[<1>], int shmflg)
18	int shmctl(int shmid, int cmd, struct shmid_ds *buf)
19	int shmdt(const void *shmaddr[<1>])
20	int shmget(key_t key, size_t size, int shmflg)
21	int shm_open(const char *name, int oflag, mode_t mode)
22	int shm_unlink(const char *name)
23	int stat(const char *path, struct stat *buf)

Opcode	Function Prototype
24	int unlink(const char *pathname)
25	pid_t wait(int *status)
26	pid_t waitpid(pid_t pid, int *status, int options)
27	ssize_t write(int fd, const void *buf, size_t count)
28	int ftruncate(int fd, off_t length)
29	int access(const char *pathname, int mode)
30	int dup(int oldfd)
31	time_t time(time_t *t)
32	int nanosleep(const struct timespec *req, struct timespec *rem)
33	int chdir(const char *path)
34	int fchdir(int fd)
35	int mkdir(const char *pathname, mode_t mode)
36	int mknod(const char *pathname, mode_t mode, dev_t dev)
37	int rmdir(const char *pathname)
38	int chmod(const char *path, mode_t mode)
39	int fchmod(int fildes, mode_t mode)
40	int chown(const char *path, uid_t owner, gid_t group)
41	int fchown(int fd, uid_t owner, gid_t group)
42	int lchown(const char *path, uid_t owner, gid_t group)
43	char *getcwd(char *buf, size_t size)
44	int link(const char *oldpath, const char *newpath)
45	int symlink(const char *oldpath, const char *newpath)
46	ssize_t readlink(const char *path, char *buf, size_t bufsiz)
47	void sync(void)
48	int fsync(int fd)
49	int fdatasync(int fd)
50	int dup2(int oldfd, int newfd)
51	int lockf(int fd, int cmd, off_t len)
52	int truncate(const char *path, off_t length)
53	int mkstemp(char *template)
54	char *mktemp(char *template)
55	DIR *<1>opendir(const char *name)
56	int closedir(DIR *<1>dir)
57	struct dirent *readdir(DIR *<1>dir);
58	void rewinddir(DIR *<1>dir)
59	void seekdir(DIR *<1>dir, off_t offset)
60	off_t telldir(DIR *<1>dir)
61	int sched_yield(void)

Note: All pointers are SPU local addresses unless otherwise designated. The definition of the data structures and types are specific to the SPU ABI and should not be assumed to match the ABI that is used by the host operating system. Pointers marked by the superscript <1> are 64-bit main storage effective address.

Table 3-8: Registered POSIX.1b opcodes

Opcode	Function Prototype

Table 3-9: Registered Operating-System-Dependent system calls opcodes

Opcode	Function Prototype

3.3.3. Pointer Parameters

Pointer parameters can be either local-store pointers (32-bits) or effective-address (EA) pointers (64-bits). The opcode registry shall specify the type of pointer for each opcode that contains pointer parameters. Therefore, it is possible that two versions of assisted calls will be provided—one with local-store pointer parameters, and one with EA pointer parameters.

3.3.4. Debugger Considerations

Since assisted library calls introduce data within the SPE's instruction sequence, special accommodations must be made within debuggers in order to successfully support single stepping.

Debugger single step is typically implemented by replacing the next instruction with a `stopd` instruction. The next instruction depends upon the current instruction and the state of the registers.

For example, if the current instruction is the conditional branch instruction, `brz`, then the next instruction can be either the next sequential instruction (PC+4) or the instruction of the branch address specified by the instruction, depending upon the value of the specified register. Therefore, debuggers must be aware of the current instruction in order to determine the next instruction in which to place the next `stopd` instruction.

Externally assisted library calls introduce additional instructions that debuggers must be aware of in order to correctly predict the next instruction. If the current instruction is a `stop and signal` instruction with a type of 0x2100 through 0x21FF, then the next instruction is PC+8, not the next sequential instruction, PC+4.

END OF DOCUMENT