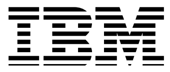


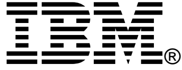
SPU Application Binary Interface Specification

Version 1.7

CBEA JSRE Series

Cell Broadband Engine Architecture
Joint Software Reference Environment
Series

March 8, 2007



© Copyright International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation 2003, 2004, 2005, 2006, 2007

All Rights Reserved
Printed in the United States of America   March 2007

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

IBM   PowerPC
IBM Logo
ibm.com

Cell Broadband Engine is a trademark of Sony Computer Entertainment, Inc.

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

March 8, 2007

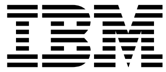


Table of Contents

List of Tables iv

List of Figures iv

About This Document v
- Audience v
- Version History v
- Related Documentation vi
- Bit Notation and Typographic Conventions Used in This Document vii

1. Introduction 1

2. Low-Level System Information 3
- 2.1. Data Representation 3
  - 2.1.1. Byte Ordering 3
  - 2.1.2. Register Layout 4
  - 2.1.3. Fundamental Types 4
  - 2.1.4. Aggregates and Unions 5
  - 2.1.5. Bit-Fields 6
  - 2.1.6. Volatiles 7
- 2.2. Function Calling Sequence 7
  - 2.2.1. Registers 8
  - 2.2.2. The Stack Frame 8
  - 2.2.3. Argument Passing 10
  - 2.2.4. Variable Argument Lists 11
  - 2.2.5. Return Values 13
  - 2.2.6. Out-of-Module Function Calls 13
- 2.3. Coding Examples 13
  - 2.3.1. Code Model Overview 13
  - 2.3.2. Function Prologue and Epilogue 14
  - 2.3.3. Register Saving and Restoring Functions 14
  - 2.3.4. Data Objects 16
  - 2.3.5. Function Calling by Name 17
  - 2.3.6. Function Calling by Pointer 17
  - 2.3.7. Dynamic Stack Space Allocation 18
- 2.4. Debug Format 19
  - 2.4.1. DWARF Register Number Mapping 19
  - 2.4.2. Address Class Code 19
- 2.5. Operating System Interface 19
  - 2.5.1. Program Initialization 19

3. Object Files 21
- 3.1. File Format 22
- 3.2. ELF Header 22
- 3.3. Symbols 23
- 3.4. Sections 23
- 3.5. Relocation 23
  - 3.5.1. Relocation Types 23

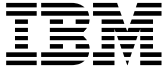
4. Program Loading and Dynamic Linking 25
- 4.1. Program Header 26
  - 4.1.1. SPU Environment Note 26
  - 4.1.2. SPU Name Note 27

List of Tables

Table 2-1: Fundamental Data Types 4
Table 2-2: Vector Types 5
Table 2-3: Bit-Field Ranges 7
Table 2-4: General-Purpose Register Conventions 8
Table 2-5: Example of Register and Stack Assignment 11
Table 2-6: Relative-Addressed Named Function Calls 17
Table 2-7: Absolute-Addressed Named Function Calls 17
Table 2-8: SPU Register Number Mapping 19
Table 2-9: SPU Address Class Code 19
Table 3-10: SPU ELF Header Fields 22
Table 3-11: SPU Special Sections 23
Table 3-12: Relocation Fields 23
Table 3-13: Relocation Types 24
Table 4-14: SPU Environment Note 26
Table 4-15: spu_env Structure 26
Table 4-16: SPU Name Note 27

List of Figures

Figure 2-1: Bit and Byte Numbering of Halfwords 3
Figure 2-2: Bit and Byte Numbering of Words 3
Figure 2-3: Bit and Byte Numbering of Doublewords 3
Figure 2-4: Bit and Byte Numbering of Quadwords 3
Figure 2-5: Register Layout of Data Types 4
Figure 2-6: Vector Data Types Byte Ordering and Element Numbering 5
Figure 2-7: Structure Smaller Than a Word 6
Figure 2-8: Structure with No Padding 6
Figure 2-9: Structure with Internal Padding 6
Figure 2-10: Structure with Internal and Trailing Padding 6
Figure 2-11: Union Allocation 6
Figure 2-12: Standard Stack Frame 9
Figure 2-13: Layout of the Parameter List Area 10
Figure 2-14: Contents of stdarg.h 12
Figure 2-15: Pseudo-Code Implementations of Variable Argument List Macros 12
Figure 2-16: Sample Register Save Functions 15
Figure 2-17: Sample Register Restore Functions 16
Figure 2-18: Position-Independent Load and Store 17
Figure 2-19: Function Calling by Pointer 18
Figure 2-20: Dynamic Stack Space Allocation 18
Figure 2-21: Memory Stack 20
Figure 3-22: Object File Format 22



About This Document

This document defines the Application Binary Interface (ABI) of the Synergistic Processor Unit (SPU).

Audience

This document is intended for system and application programmers who develop language processors and other software for the SPU of a processor compliant with the Cell Broadband Engine™ Architecture (CBEA).

Version History

This section describes significant changes made to each version of this document.

Version Number & Date	Changes
v. 1.7 March 8, 2007	Corrected minor organizational, grammatical, and spelling issues (TWG_RFC00094-0: CORRECTION NOTICE). Corrected minor errors (TWG_RFC00101-0: CORRECTION NOTICE)
v. 1.6 December 4, 2006	Corrected the placement of the label in the sample register save function and added a missing label to the sample register restore function (TWG_RFC00084-0). Corrected the spelling of the `va_start` macro and added `va_copy` to the list declared in `stdarg.h` (TWG_RFC00085-0). Modified relocations to match the actual implementation, and added two new relocations (TWG_RFC00088-0).
v. 1.5 (corrected) October 11, 2006	Applied TWG_RFC00069-1. Changed the SPUNAME descriptor size from a fixed size (32 bytes) to a variable size that is a multiple of 4 bytes (TWG_RFC00048-0). Revised the comment corresponding to the `e_machine` SPU ELF header field, reflecting the fact that 23 has been officially accepted as its value. Made miscellaneous editorial changes. Applied the changes made in the following requests: TWG_RFC00063-2, TWG_RFC00064-0, TWG_RFC00065-1.
v. 1.4 October 20, 2005	Defined a standard process for memory heap initialization and stack management (TWG_RFC00024-3). Changed the section describing rules that apply to the stack frame (TWG_RFC00030-0). Changed “Broadband Processor Architecture” to “Cell Broadband Engine Architecture”, and changed “BPA” to “CBEA” (TWG_RFC00037-0: CORRECTION NOTICE). Added several restrictions that apply to allocatable ELF sections that will be loaded into local storage (TWG_RFC00038-2 as amended by TWG_RFC00044-0). Specified that the `R2` register will be used as an environment pointer for languages that require one (TWG_RFC00039-0). Corrected several documentation errors (TWG_RFC00041-0: CORRECTION NOTICE, TWG_RFC00045-0: CORRECTION NOTICE).
v. 1.3 July 11, 2005	Deleted several sections in the “About This Document” chapter and corrected several documentation errors. For example, in the Relocation Types table, the “Field” entry corresponding to `R_SPU_ADDR7` was changed from `I7*` to `I7` (TWG_RFC00032-0: CORRECTION NOTICE).

Version Number & Date	Changes
v. 1.2 June 10, 2005	Changed “Broadband Engine” or “BE” to “a processor compliant with the Broadband Processor Architecture” or “a processor compliant with BPA”; and changed Synergistic Processing Unit to Synergistic Processor Unit. Defined a PPU as a PowerPC Processor Unit on first major instance. Corrected several book references and changed copyright page so that trademark owners were specified. (All changes per TWG_RFC00031-0: CORRECTION NOTICE.) Made miscellaneous changes to the “About This Document” section.
v. 1.1 May 9, 2005	Changed PU to PPU (TWG_RFC00028-0: CORRECTION NOTICE).
v.1.0 December 14, 2004	Added a `PT_NOTE` section to all SPU ELF executables (TWG_RFC00019-0). Modified stack layout to eliminate a requirement for minimum space in the Parameter List Area, and increased the number of registers used for argument passing and for the return value (TWG_RFC00020-0).
v. 0.9 July 16, 2004	Changed the description of the `.bss` SPU special section. This description now specifies that the program loader is responsible for initializing `.bss` (TWG_RFC00001-2). Changed general-purpose register conventions to reflect a re-allocation among volatile and non-volatile registers. Specifically, the number of non-volatile registers has been decreased. This change also affected several figures (TWG_RFC00004-5). Made miscellaneous editorial changes.
v. 0.8 March 12, 2004	Added requirement that global data types must always be aligned to a 16-byte boundary, as requested in TWG_RFC00006. Made miscellaneous editorial changes.
v. 0.7 February 25, 2004	Changed formatting of document so that it reflects typographic conventions described on page vii. Made miscellaneous editorial changes.
v. 0.6 January 23, 2004	Changed document to new format, including front matter. Made miscellaneous editorial changes.
v. 0.5 September 15, 2003	Added `R_SPU_ADDR10I` relocation type. Added `SPUNAME` section note to support SPU plug-ins. Added description for ELF header field `e_type`.
v. 0.4 June 15, 2003	Changed mechanism of stack initialization and overflow detection. Added additional relocation types `R_SPU_ADDR7`, `R_SPU_REL9` and `R_SPU_REL9I`. Added program header section describing SPU environment notes used for embedding SPU objects with PU/PPC objects.
v. 0.3 March 7, 2003	Provided structure padding examples. Added additional conventions regarding volatiles. Removed description of TOC register and its usage. Edited register save/restore sample functions. Specified symbol name mangling convention.
v. 0.2 November 21, 2002	Added register layout diagram. Added out-of-module function calling sequence. Specified that the parameter list area equal at least 8 quadwords. Provided examples of a parameter passing convention. Began adding support for interrupt handling. Edited register save/restore and function calling code samples.
v. 0.1 September 30, 2002	Initial release of this document.

Related Documentation

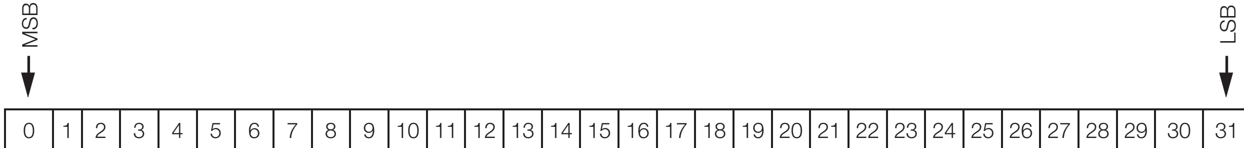
The following table provides a list of references and supporting materials for this document:

Document Title	Version	Date
Tool Interface Standard (TIS), Executable and Linking Format (ELF) Specification	1.2	May 1995
Tool Interface Standard (TIS), DWARF Debugging Information Format Specification	2.0	May 1995

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

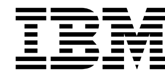
Notation for bit encoding is as follows:

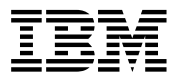
- Hexadecimal values are preceded by `0x`. For example: `0x0A00`.
- Binary values in sentences appear in single quotation marks. For example: ‘1010’.

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).

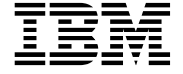


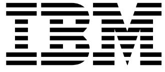


1. Introduction

The SPU Application Binary Interface defines the system interface for compiled application programs, which enables these programs to be run without recompilation or recoding on a Synergistic Processor Unit of a CBEA-compliant system. The purpose of this document is to standardize the set of binary interface specifications to achieve portability.

This document defines low-level language binding conventions. Although the C programming language is used to illustrate these conventions, other languages are not precluded from use.





2. Low-Level System Information

This chapter prescribes the rules that language processors must follow. By adhering to these rules, language processors will be able to accomplish the following objectives:

- Generate conforming code for function-calling sequences, including passing arguments, returning values, and using registers.
- Allow access to a program's global data from code modules written in different source languages. (Only rules for common data types are defined.)
- Mix object modules generated by language processors from different vendors.

2.1. Data Representation

2.1.1. Byte Ordering

The SPU architecture defines the following machine data types:

- 8-bit byte
- 16-bit halfword
- 32-bit word
- 64-bit doubleword
- 128-bit quadword

Byte ordering defines how the bytes that comprise halfwords, words, doublewords, and quadwords are ordered in memory. The SPU supports most significant byte (MSB) ordering. An MSB, or "big endian", ordering means that the most significant byte is located in the lowest addressed byte position in a storage unit (byte 0).

Figure 2-1 through Figure 2-4 illustrate the conventions for bit and byte numbering within various width storage units. These conventions apply to both integer and floating-point data (where the most significant byte holds the sign and at least the start of the exponent). The following figures show byte numbers on the top and bit numbers in the lower corners.

Figure 2-1: Bit and Byte Numbering of Halfwords

0	*1*
MSB	*LSB*
0 7	8 15

Figure 2-2: Bit and Byte Numbering of Words

0	*1*	*2*	*3*
MSB			*LSB*
0 7	8 15	16 23	24 31

Figure 2-3: Bit and Byte Numbering of Doublewords

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
MSB							*LSB*
0 7	8 15	16 23	24 31	32 39	40 47	48 55	56 63

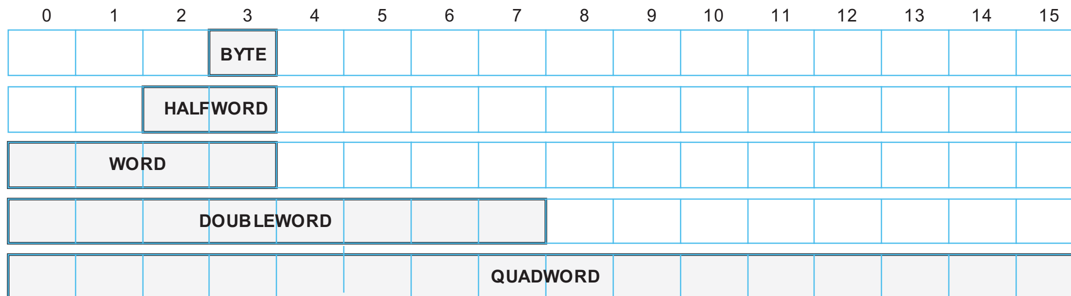
Figure 2-4: Bit and Byte Numbering of Quadwords

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*
MSB															*LSB*
0 7	8 15	16 23	24 31	32 39	40 47	48 55	56 63	64 71	72 79	80 87	88 95	96 103	104 111	112 119	120 127

2.1.2. Register Layout

The general-purpose registers are 128-bits wide. Within these registers, data types that are less than 128 bits in size are placed in a specified location that is referred to as the “preferred slot”. Figure 2-5 illustrates how data types are laid out in a general-purpose register.

Figure 2-5: Register Layout of Data Types



2.1.3. Fundamental Types

Table 2-1 shows the standard C data types, their size and alignment, and their corresponding SPU machine data type. Global variables must always be aligned to a 16-byte boundary regardless of their data type. Although 16-byte alignment of global variables increases the amount of data memory that is used, this alignment enables faster access to the data with fewer instructions, thereby compensating for the additional data memory usage.

Table 2-1: Fundamental Data Types

Type	C Type	Sizeof	Alignment (bytes)	SPU Machine Data Type
Character	char unsigned char	1	1	unsigned byte
	signed char	1	1	signed byte
	short signed short	2	2	signed halfword
	unsigned short	2	2	unsigned halfword
Integral	_Bool	1	1	unsigned byte
	int signed int long int signed long enum	4	4	signed word
	unsigned int unsigned long	4	4	unsigned word
	long long signed long long	8	8	signed doubleword
	unsigned long long	8	8	unsigned doubleword
Pointer	any type * any type (*) ( )	4	4	unsigned word
Floating-Point	float	4	4	single precision
	double	8	8	double precision
	long double	8	8	double precision
Vector	any type	16	16	quadword

This ABI does not specify IEEE 754 double extended precision (128-bit) floating-point. Programs that use this standard are not ABI conformant, and platforms that implement this ABI are not required to support these programs. If a platform supports double extended precision, it must be implemented with a sign bit, a 15-bit exponent with a bias of 16383 and 112 fraction bits with a leading "implicit" bit. Alignment must be 16 bytes.

The SPU supports several vector data types. All of the vector types are 128-bits and contain multiple scalar elements. Table 2-2 describes the supported vector types.

Table 2-2: Vector Types

Vector Data Type	Contents
qword	128-bit quadword vector of unspecified type
vector unsigned char	16 8-bit unsigned integer characters (bytes)
vector signed char	16 8-bit signed integer characters (bytes)
vector unsigned short	8 16-bit unsigned integer halfwords
vector signed short	8 16-bit signed integer halfwords
vector unsigned int	4 32-bit unsigned integer words
vector signed int	4 32-bit signed integer words
vector unsigned long long	2 64-bit unsigned integer doublewords
vector signed long long	2 64-bit signed integer doublewords
vector float	4 32-bit single precision floats
vector double	2 64-bit double precision floats

Vectors and vector elements also use MSB ordering, as shown in Figure 2-6:

Figure 2-6: Vector Data Types Byte Ordering and Element Numbering

<table>
<tr><td>Byte 0 (MSB)</td><td>Byte 1</td><td>Byte 2</td><td>Byte 3</td><td>Byte 4</td><td>Byte 5</td><td>Byte 6</td><td>Byte 7</td><td>Byte 8</td><td>Byte 9</td><td>Byte 10</td><td>Byte 11</td><td>Byte 12</td><td>Byte 13</td><td>Byte 14</td><td>Byte 15 (LSB)</td></tr>
<tr><td colspan="8">doubleword 0</td><td colspan="8">doubleword 1</td></tr>
<tr><td colspan="4">word 0</td><td colspan="4">word 1</td><td colspan="4">word 2</td><td colspan="4">word 3</td></tr>
<tr><td colspan="2">halfword 0</td><td colspan="2">halfword 1</td><td colspan="2">halfword 2</td><td colspan="2">halfword 3</td><td colspan="2">halfword 4</td><td colspan="2">halfword 5</td><td colspan="2">halfword 6</td><td colspan="2">halfword 7</td></tr>
<tr><td>char 0</td><td>char 1</td><td>char 2</td><td>char 3</td><td>char 4</td><td>char 5</td><td>char 6</td><td>char 7</td><td>char 8</td><td>char 9</td><td>char 10</td><td>char 11</td><td>char 12</td><td>char 13</td><td>char 14</td><td>char 15</td></tr>
</table>

2.1.4. Aggregates and Unions

Aggregates, whether structures or arrays, and unions assume the alignment of their most strictly aligned component (the component with the largest alignment). The size of any object, including aggregates and unions, is always a multiple of the alignment of the object.

An array uses the same alignment as its elements. Structure and union objects might require padding to meet size and alignment constraints, according to the following criteria:

- An entire structure or union object is aligned on the same boundary as its most strictly aligned member.
- Each member is assigned to the lowest available offset with the appropriate alignment. This might require internal padding, depending on the previous member.
- If necessary, the size of a structure is increased to make it a multiple of the structure's alignment. This might require tail padding, depending on the last member.

To improve structure access efficiency, compilers may place further restrictions on outer-most structures to achieve quadword alignment.

The examples in Figure 2-7 through Figure 2-11 illustrate each of the above alignment rules:

Figure 2-7: Structure Smaller Than a Word

Figure 2-8: Structure with No Padding

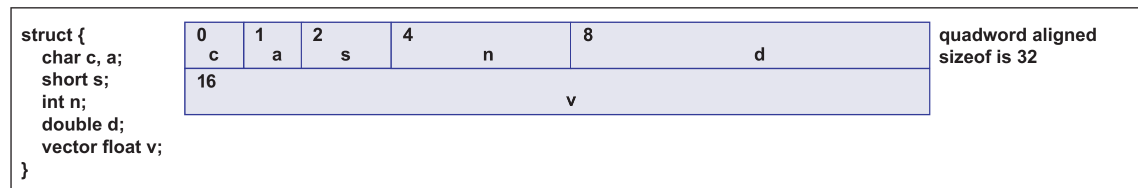


Figure 2-9: Structure with Internal Padding

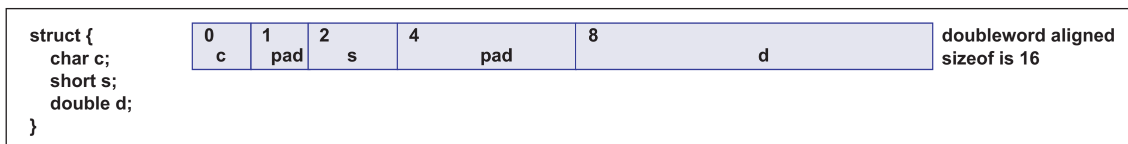


Figure 2-10: Structure with Internal and Trailing Padding

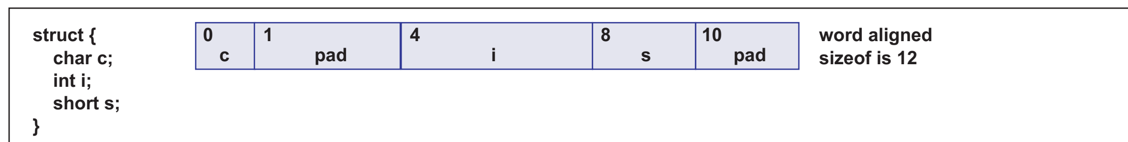
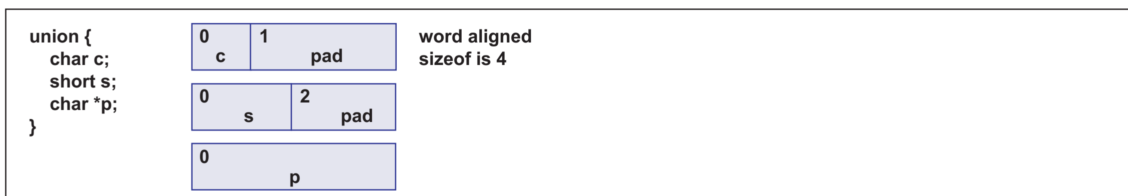


Figure 2-11: Union Allocation



2.1.5. Bit-Fields

C structs or unions may have “bit-fields” defining integral objects that have a specified number of bits. “Plain” bit-fields (those that are neither *signed* nor *unsigned*) always have non-negative values. Although bit-fields may be of type `short`, `int`, `long`, or `long long` (which may have negative values), bit-fields of these types have the same range as bit-fields of the same size of a corresponding unsigned type.

Bit-fields use the same size and alignment rules as other structure and union members, with the following additions:

- Bit-fields are allocated from left to right, that is, from the most to the least significant bit.
- A bit-field must be completely located in a storage unit appropriate for its declared data type. Thus, a bit-field never crosses its unit boundary.

- Bit-fields must share a storage unit with other structure and union members (either bit-field or non bit-field) if and only if there is sufficient space within the storage unit.
- Unnamed bit-field data types do not affect the alignment of a structure or union, although the member offsets of an individual bit-field adhere to the alignment constraints. An unnamed zero-width bit-field must prevent any other member, whether a bit-field or another kind of member, from residing in the storage unit corresponding to the data type of the zero-width bit-field.

Table 2-3 shows the width and ranges for each supported bit-field data type.

Table 2-3: Bit-Field Ranges

Bit-Field Data Type	Width (w)	Range
signed char	1 to 8	-2^{w-1} to $2^{w-1}-1$
char		0 to $2^{w}-1$
unsigned char		0 to $2^{w}-1$
signed short	1 to 16	-2^{w-1} to $2^{w-1}-1$
short		0 to $2^{w}-1$
unsigned short		0 to $2^{w}-1$
signed int	1 to 32	-2^{w-1} to $2^{w-1}-1$
int		0 to $2^{w}-1$
enum		0 to $2^{w}-1$
unsigned int		0 to $2^{w}-1$
signed long		-2^{w-1} to $2^{w-1}-1$
long		0 to $2^{w}-1$
unsigned long		0 to $2^{w}-1$
signed long long	1 to 64	-2^{w-1} to $2^{w-1}-1$
long long		0 to $2^{w}-1$
unsigned long long		0 to $2^{w}-1$

2.1.6. Volatiles

The SPU processor only supports quadword data accesses. Volatile qualified variables must reside in their own quadwords in order to achieve correct volatile semantics. The programmer is responsible for ensuring that these semantics are followed. The ABI does not provide additional specific rules for alignment or allocation of volatile qualified variables.

2.2. Function Calling Sequence

This section describes the standard function calling sequence, including stack frame layout, register usage, and argument passing.

The standard calling sequence requirements apply only to global functions. Local functions that are not reachable from other compilation units may use different conventions; however, using non-standard calling sequences is not recommended.

2.2.1. Registers

The SPU has 128 general-purpose registers. These registers are each 128-bits wide. Table 2-4 shows the status and usage of these registers.

Table 2-4: General-Purpose Register Conventions

Register	Status	Usage
R0 (LR)	Dedicated	Return Address / Link Register. This register contains the address to which a called function normally returns. It is volatile across function calls and must be saved by a non-leaf function.
R1 (SP)	Dedicated	Stack pointer information. Word element 0 of the SP register contains the current stack pointer. The stack pointer is always 16-byte aligned, and it must always point to the lowest allocated valid stack frame and grow towards low addresses. The contents of the word at the stack-frame address always point to the previous allocated stack frame. Word element 1 of the SP register contains the number of bytes of Available Stack Space. See section "2.2.2. The Stack Frame" for more details.
R2	Volatile	Environment pointer. This register is used as an environment pointer for languages that require one.
R3 – R74	Volatile	First seventy-two quadwords of a function's argument list and its return value.
R75 – R79	Volatile	Scratch Registers.
R80 - R127	Non-volatile	Local variable registers. These must be preserved across function calls.

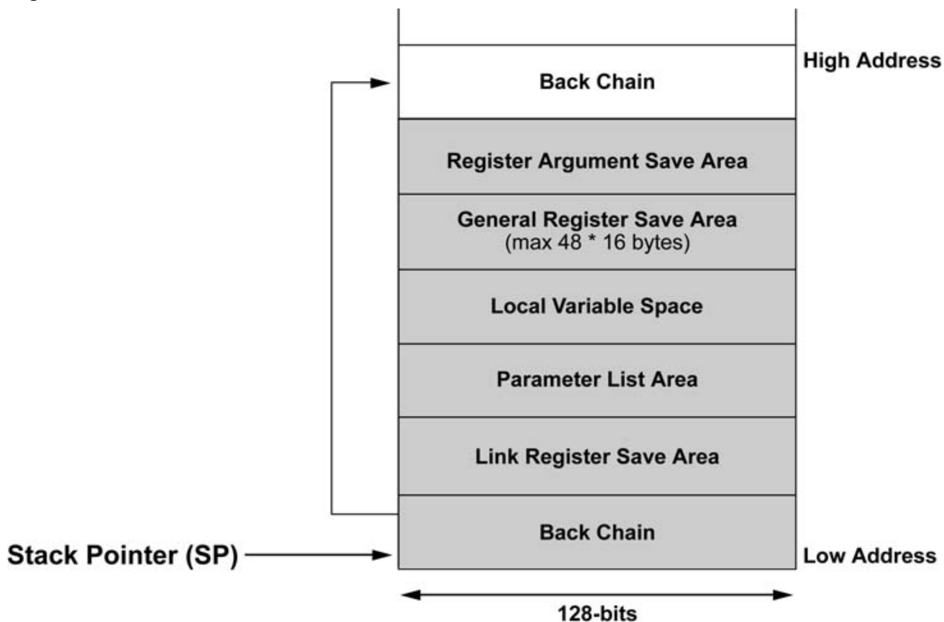
Registers `R0`, `R2`, and `R3` through `R79` are volatile; values in these registers are not preserved across function calls. Values in register `R0` and `R75` through `R79` may not even be preserved during the function call sequence, so a function cannot depend on these registers having the same values that were placed in them by the caller.

Registers `R1` and `R80` through `R127` are non-volatile. A called function must save the values in these registers before it changes them, and it must restore the former values to these registers before it returns.

2.2.2. The Stack Frame

In addition to using registers, each function call may have a stack frame on the runtime stack. The runtime stack grows downward from high addresses. Figure 2-12 shows the stack frame organization. In this figure, *SP* denotes the stack pointer (word element 0 of the general-purpose register `R1`) of the called function after it has executed the code that establishes its stack frame.

Figure 2-12: Standard Stack Frame



The following requirements apply to the stack frame:

- The stack pointer must maintain 16-byte (quadword) alignment.
- The stack pointer must point to the first word of the lowest allocated stack frame, the Back Chain word. The stack must grow downward (toward lower addresses). The first word of the stack frame must always point to the previously allocated stack frame (toward higher addresses), except for the first stack frame, which must have a back chain pointer of 0 (NULL).
- If a stack pointer is required, all word elements of the Stack Pointer Information register (SP) must be decremented by the called function and restored prior to its return.
- Storing to memory using the stack pointer plus an offset must never be done with an offset less than -2000 (-125*16). This allows interrupt handlers to use the application stack by first adding -2000 to the stack pointer.
- When a stack frame is allocated, stack overflow can be tested by evaluating the Available Stack Space word (word element 1 of `R1`) of the decremented Stack Pointer Information register. If the Available Stack Space word is negative, an overflow is detected and program execution is halted.
- A Parameter List Area must be allocated by the caller if the caller needs to pass more than seventy-two quadwords of arguments. See section "2.2.3. Argument Passing". If the Parameter List Area is needed, it must be large enough to contain all of the arguments that are not passed in registers. Its contents are not preserved across function calls.
- Before a function changes the value of any non-volatile register, it must save the value of the entire 128-bit register in a quadword in the General Register Save Area.
- Other areas depend on the compiler and the code being compiled. The standard calling sequence does not define the maximum stack frame size. The minimum stack frame consists of the first two quadwords, described below. The calling sequence also does not restrict how a language uses the Local Variable Space of the standard stack frame or how large the Local Variable Space must be.

The stack frame header consists of both the Back Chain quadword and the Link Register Save Area quadword. The 32 most significant bits of the 128-bit quadword contain a Back Chain pointer and return address, respectively. The remaining 96 bits of each quadword are reserved for use by the tool chain.

Before a function calls another function, the calling function must:

- Save the contents of the 128-bit Link Register at the time the function was entered in the Link Register Save Area of its caller's stack frame
- Establish its own stack frame

Except for the stack frame header, a function is not required to allocate space for the areas that it does not use. If a function does not call any other functions and does not require any of the other parts of the stack frame, it does not need to establish a stack frame.

Any padding of the entire frame must be within the Local Variable Space. The Parameter List Area must immediately follow the stack frame header, and the Register Save Area must not contain padding.

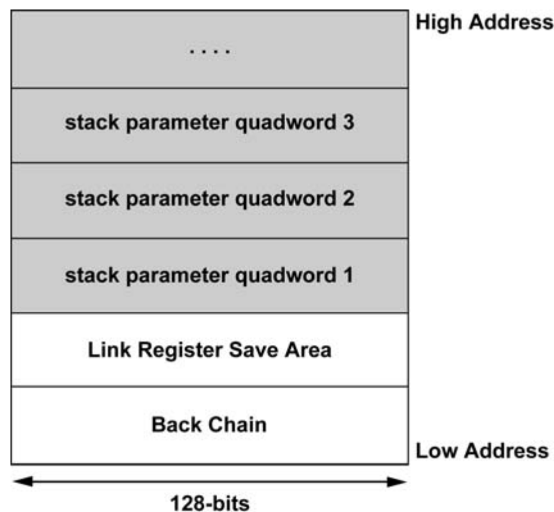
2.2.3. Argument Passing

It is more efficient to pass function arguments in registers than it is to construct an argument list in storage or to push arguments onto the stack. There are two reasons for this efficiency: 1) all computations must be performed in registers, and 2) memory traffic can be eliminated if the caller computes arguments into registers and passes these arguments in the same registers to the called function. In the second case, the called function can then use the same registers for further computation.

For the SPU, up to seventy-two quadwords are passed in general-purpose registers, loaded sequentially into registers `R3` through `R74`. If fewer than seventy-two argument registers are needed, the unneeded registers are not loaded, and any values that they contain when entering the called function are undefined.

When arguments passed to a callee function will not fit into these seventy-two registers, the caller function must allocate additional space for these arguments in its Parameter List Area, as shown in Figure 2-13.

Figure 2-13: Layout of the Parameter List Area



The following algorithm specifies where argument data is passed for the C language. For illustrative purposes, consider the arguments ordered from left, for the first argument, to right, although the actual order of evaluation of the arguments is unspecified.

- Initialize *reg* = 3 and *stack_arg* = address of parameter quadword 1.
- For each argument, determine the type of argument and store it according to the following rules:
  - For simple arguments (scalars, vectors, or pointers to an object), if *reg* is less than or equal to 74, copy the argument into register *reg* and then increment *reg*.

– For *structs* or *unions*, if the entire structure will fit into the remaining argument registers, place a memory image of the argument, aligned to the alignment of the structure, into registers 16 bytes at a time until the entire argument has been copied. Otherwise, place the entire structure into the stack, as described below. (See section “2.1.4. Aggregates and Unions”.)
– Pass non-simple arguments or simple arguments with *reg* greater than 74 (that is, arguments not handled above) in the parameter quadwords of the caller’s stack frame. The values passed on the stack are identical to those that have been placed in registers; thus, the stack contains register images. This stack assignment can be accomplished by doing the following:

  (a) Pad *stack_arg* to quadword alignment and copy the argument byte-for-byte (beginning with the lowest addressed byte), into *stack_arg*, ..., *stack_arg+size-1*, where *size* is the number of bytes in the argument.

  (b) Set *stack_arg* to *stack_arg+size*.
– Place simple arguments in the “preferred slot” of the quadword, as described in section “2.1.2. Register Layout”.

The contents of the registers and words skipped by the above alignment algorithm are undefined.

Example of Parameter Passing

```
struct {
   int i;
   double d;
   vector unsigned int v[36];
} s, t;
int a, b;
float x, y, z;

x = func(a, x, y, z, s, t, b);
```

In this example, the parameters are passed in registers and on the stack, as shown in Table 2-5.

Table 2-5: Example of Register and Stack Assignment

Parameter	Register(s)	Parameter List Area Offset
a	R3	Not stored
x	R4	Not stored
y	R5	Not stored
z	R6	Not stored
s	R7 – R43	Not stored
t	–	0 – 591
b	–	592 – 607

2.2.4. Variable Argument Lists

The ANSI C specification requires that a prototype containing trailing ellipses (...) be used when declaring a function with a variable argument list.

Some generally portable C programs depend on a particular argument-passing scheme. Such programs assume that all arguments are passed on the stack, and that arguments appear in increasing order on the stack. Programs that make these assumptions are not truly portable, although they might have performed correctly with many implementations. Nevertheless, these programs will not work with compliant SPU compilers because some of the arguments are passed in registers.

To manage variable argument lists, portable C programs use the `va_start`, `va_arg`, `va_end`, and `va_copy` macros, and the `va_list` type. These macros are defined by the compiler and are provided in the header file `stdarg.h`.

Arguments to variable-argument functions are passed using the same method that is used for passing arguments to fixed-argument functions. As described in section “2.2.3. Argument Passing”, arguments are placed in registers `R3` up to `R74`, and if necessary, the Parameter List Area. The callee, a variable-argument function, copies the argument registers into its Register Argument Save Area. The relative location of the Register Argument Save Area is shown in Figure 2-12.

Scalar variable arguments are implicitly promoted by the calling function in the same way as arguments without data types. Arguments of character and short data type are promoted to integers, and single-precision floats are promoted to double-precision floats. All other data types are not promoted.

The `va_list` type and the variable argument macros that are declared in `stdarg.h` are shown in Figure 2-14.

Figure 2-14: Contents of stdarg.h

```
/* Aligning the fields makes accessing them faster. */
typedef struct __va_list {
  char *next_arg  __attribute__ ((__aligned__ (16)));
  char *caller_stack  __attribute__ ((__aligned__ (16)));
} va_list;



#define va_start(v,l)   __builtin_va_start(v,l)
#define va_end(v)       /* nothing */
#define va_arg(v,l)     __builtin_va_arg(v,l)
#define va_copy(d,s)    (d) = (s)
```

The `__builtin_va_start` and `__builtin_va_arg` functions that are shown in Figure 2-14 are implemented within the compiler and behave according to the pseudo-code shown in Figure 2-15.

Figure 2-15: Pseudo-Code Implementations of Variable Argument List Macros

```
__builtin_va_start (AP, LAST)
{
  int paddedsize = (sizeof (LAST) + 15) & -16;

  AP.next_arg = (unsigned char *) &LAST;

  /* get caller's stack pointer */
  AP.caller_stack = __builtin_frame_address (1);

  if (AP.next_arg + paddedsize > AP.caller_stack   && AP.next_arg <= AP.caller_stack)
    AP.next_arg = AP.caller_stack + 32;
  else
    AP.next_arg += padded_size;
}

TYPE __builtin_va_arg(AP, TYPE)
{
  int padded_size = (sizeof(TYPE) + 15) & -16;
  char *argp;

  /* If this arg overlaps with AP.caller_stack, the
     whole argument must start at the beginning of the caller's
     arguments. */
```

```
if (AP.next_arg + paddedsize > AP.caller_stack
    && AP.next_arg <= AP.caller_stack)
  argp = AP.caller_stack + 32;
 else
  argp = AP.next_arg;
 AP.next_arg = argp + paddedsize;
 return *(TYPE *)argp;
}
```

2.2.5. Return Values

Functions must return scalars, vectors, and aggregates and unions beginning with register `R3`. (Scalars are pointers or one of the following data types: `char`, `short`, `int`, `enum`, `long int`, `long long`, `float`, or `double`. Aggregates are structures and arrays.) Aggregates occupy the most significant, or left-most, bytes of the register.

Aggregates and unions that are larger than 1152 bytes must be returned in a storage buffer allocated by the caller. The address of the buffer is passed as a hidden argument in `R3`. This address is passed as if it were the first argument, causing *`reg`* in the argument-passing algorithm to be initialized to 4 instead of 3.

2.2.6. Out-of-Module Function Calls

In general, SPU programs are statically bound because all of the symbols are fully resolved by the link editor; however, a limited form of dynamic binding, which is referred to as “plug-in” dynamic binding, is allowed. The following characteristics define plug-in dynamic binding:

- Plug-in modules contain no dynamic external references and have a single entry point.
- Plug-in modules are loaded by the SPU. The plug-in module’s entry point is returned as a function pointer from the SPU plug-in loader.
- Multiple plug-in modules may co-exist. The SPU program is responsible for plug-in storage management.
- Data sharing between the caller and plug-in callback functions may be passed to and from the plug-in by mutual agreement. This ABI does not enforce a specific mechanism.

Calling a plug-in causes a function call by pointer. See section “2.3.6. Function Calling by Pointer”.

2.3. Coding Examples

This section describes example code sequences for fundamental operations, such as calling functions, accessing static objects, and transferring control from one part of a program to another. Previous sections described how a program must use the system and what a program may assume about the execution environment. Unlike previous sections, this section describes how operations might be done, rather than how they must be done.

The examples in this section use ANSI C language conventions. Other programming languages may use the same conventions. Regardless, failure to use these conventions will not prevent a program from conforming to the ABI.

SPU code is normally position-independent; that is, the code does not depend on a specific load address, and it may be executed properly at various positions in local storage. Although it is possible to write code that is not position-independent, the following examples show only position-independent code.

2.3.1. Code Model Overview

The SPU processor fills the niche between a general-purpose and special-purpose processor, and its particular architectural features influence the techniques that are most effectively used to program it. Among the techniques that might be used are 1) the use of plug-in objects to support large programs within the limited local storage, and 2)

the use of coroutines to support multiple simultaneous execution threads without incurring either preemptive or non-preemptive context switching overhead.

Techniques, such as plug-in techniques, rely on the capability of the compiler to generate code that is position-independent. Position-independent code depends on:

- Control transfer instructions that hold addresses relative to the current address or use registers that hold the transfer address. (A relative branch computes its destination address in terms of the current address, not relative to an absolute address.)
- Computing absolute addresses during execution instead of embedding absolute addresses in the instructions.

These conditions are satisfied by the SPU architecture, which provides both relative and register-based branches and load/store instructions.

2.3.2. Function Prologue and Epilogue

This section describes function prologue and epilogue code. A function prologue establishes a stack frame, if necessary, and it saves any non-volatile registers that the function uses. A function epilogue restores registers that were saved in the prologue code, restores the previous stack frame, and returns to the caller.

Although this ABI does not mandate predetermined code sequences for function prologues and epilogues, the following rules, which permit reliable call-chain backtracking, must be followed:

1. If a function uses non-volatile general-purpose registers, it must save them in the General Register Save Area. This can be done prior to establishing a new stack frame by using negative offsets from the caller's stack frame. If stack overflow is tested, it must be done prior to saving any non-volatile registers in the General Register Save Area. The overflow test must also take into account the extent of stack that is used.
2. Before a function calls any other function, it must establish its own stack frame, which has a size that is a multiple of 16 bytes, and it must save the Link Register (`R0`) at the time of entry in the Link Register Save Area of its caller's stack frame.
3. Establishing a new stack frame involves adjusting all word elements of the SP register (`R1`) by the necessary negative displacement. Stack-frame overflow may be tested but testing is not required. Execution is halted prior to establishing the new stack, by adjusting the stack pointer.
4. The new stack pointer may be tested for stack overflow by testing word element 1 of the Stack Pointer Information register (`R1`). If an overflow is detected, that is, if word element 1 is negative, program execution is halted.
5. When a function de-allocates its stack frame, it must do so either by (a) loading the Stack Pointer Information register (`R1`) with the quadword value in the Back Chain or (b) incrementing all word elements of the Stack Pointer Information register by the same amount by which it was decremented.

In-line code may be used to save and restore non-volatile registers that a function uses. However, if there are many registers to be saved or restored, it might be more efficient to provide and use save and restore subroutines as described in section "2.3.3. Register Saving and Restoring Functions".

A nonstandard prologue may be used to enter a SPU program, and non-volatile registers do not need to be saved.

2.3.3. Register Saving and Restoring Functions

This section describes functions for saving and restoring registers. These functions use nonstandard calling conventions that are not part of the ABI. Nevertheless, the functions are included in this document to encourage uniformity among compilers.

These functions save/restore consecutive general registers from register 127 through register r, where r represents a value between 80 and 127. Each function represents a family of 48 sub-functions with identical behavior except for the number of registers that are affected.

To improve efficiency, a branch hint and no-operations (NOPS) could be appropriately inserted into these functions to avoid instruction fetch starvation. The following algorithm ensures that sufficient instructions to place the hint are available in the caller function and save/restore functions:

- Inline the save and restore functions if the number of registers to be saved/restored is less than some number *n*.
- If the number of registers to be saved/restored exceeds *n*, save/restore the first *n* registers inline, and then call the save/restore function to save/restore the remaining registers.

There are two functions: one function representing a family of sub-functions for saving registers, and the other representing a family of sub-functions for restoring registers:

- The register saving functions, `_savegrp_n`, save registers *n* through 127 and return. These functions expect LR to contain the return address, `R75` to contain the adjusted stack pointer, and SP to contain the address of the top of the Register Save Area. Code might also be inserted to test for stack overflow.
- The register restoring functions, `_restoregrp_n`, restore registers *n* through 127 and return. These functions expect that the 128-bit `LR` has been reloaded, `R75` contains the adjusted stack pointer, and `SP` contains the address of the top of the Register Save Area.

Figure 2-16 and Figure 2-17 show usage of the save and restore functions as called from a sample prologue and epilogue, respectively.

Figure 2-16: Sample Register Save Functions

```
# Sample prologue (saves register 94 though 127)

                    il      $75, <frame_size>
                    hbrr    prologue_branch, _savegpr_110
                    sf      $75, $75, $SP
                    stqd    $LR, 16($SP)
                    stqd    $94, -544($SP)
                    stqd    $95, -528($SP)
                    stqd    $96, -512($SP)
                    . . .
                    stqd    $108, -320($SP)
                    stqd    $109, -304($SP)
prologue_branch:    brsl    $LR, _savegpr_110


# Save function

_savegpr_80:        stqd    $80, -768($SP)
_savegpr_81:        stqd    $81, -752($SP)
_savegpr_82:        stqd    $82, -736($SP)
                    . . .

_savegpr_110:       stqd    $110, -288($SP)
_savegpr_111:       stqd    $111, -272($SP)
_savegpr_112:       hbr     _save_branch, $LR
                    stqd    $112, -256($SP)
                    stqd    $113, -240($SP)
                    stqd    $114, -224($SP)
                    stqd    $115, -208($SP)
                    . . .
```

```
                stqd   $125, -48($SP)
                stqd   $126, -32($SP)
                stqd   $127, -16($SP)
                lr     $SP, $75
_save_branch:   bi     $LR
```

Figure 2-17: Sample Register Restore Functions

```
# Sample epilogue (restores registers 94 through 127)

                 il     $75, <frame_size>
                 hbrr   epilogue_branch, _restoregpr_110
                 a      $75, $SP, $75
                 lr     $SP, $75
                 lqd    $94, -544($SP)
                 lqd    $95, -528($SP)
                 lqd    $96, -512($SP)
                 . . .
                 lqd    $108, -320($SP)
                 lqd    $109, -304($SP)
epilogue_branch: lqd    $LR, 16($SP)
                 br     _restoregpr_110


#Restore function

_restoregpr_80:  lqd    $80, -768($SP)
_restoregpr_81:  lqd    $81, -752($SP)
_restoregpr_82:  lqd    $82, -736($SP)
                 . . .
_restoregpr_110: lqd    $110, -288($SP)
_restoregpr_111: lqd    $111, -272($SP)
_restoregpr_112: hbr    _restore_branch,$LR
                 . . .
                 lqd    $112, -256($SP)
                 lqd    $113, -240($SP)
                 lqd    $114, -224($SP)
                 . . .
                 lqd    $125, -48($SP)
                 lqd    $126, -32($SP)
                 lqd    $127, -16($SP)
                 lr     $SP,$75
_restore_branch: bi     $LR
```

2.3.4. Data Objects

This section describes objects with static storage duration. It excludes stack-resident objects because programs always compute their addresses relative to the stack pointer or the frame pointer.

In the SPU architecture, only load and store instructions access memory. To maintain position-independent code, data objects must be addressed using the relative load and store instructions `lqr` and `stqr`. Examples of position-independent loads and stores are shown in Figure 2-18.

Figure 2-18: Position-Independent Load and Store

C	Assembly
extern vector unsigned int src; extern vector unsigned int dst; extern vector unsigned int *ptr;	.extern src .extern dst .extern ptr
	.text
dst = src;	lqr $5, src stqr $5, dst
ptr = &dst;	ila $2, base brsl $3, base base: ila $5, dst sf $3, $2, $3 a $5, $5, $3 stqr $5, ptr
*ptr = src;	lqr $5, ptr lqr $6, src stqd $6, 0($5)

2.3.5. Function Calling by Name

Because named functions must be statically bound, call addresses for these functions are resolved during link edit. To maintain position independence, relative branching instructions are used, as shown in Table 2-6. The instructions that are generated depend on the distance of the relative branch.

Table 2-6: Relative-Addressed Named Function Calls

Distance (bytes)	Instructions
-128K to 128K-1	brsl $LR, `relative_func_addr`

Relative-addressed function calls that are less than -128K or greater than 128K-1 bytes are supported by using a "trampoline" that is within the range of relative addressability of the SPU processor.

Position-dependent code may use absolute addressing, as shown in Table 2-7. The instructions that are generated depend on the address of the function being called.

Table 2-7: Absolute-Addressed Named Function Calls

Address	Instructions
0x00000000 to 0x0001FFFF 0xFFFE0000 to 0xFFFFFFFF	brasl $LR, `func_addr`
0x00020000 to 0xFFFDFFFF	ilhu $3, `func_addr`@h iohl $3, `func_addr`@l bisl $LR, $3

See section "3.5. Relocation" for relocation fix-up of function call branches. The notation `func_addr`@h and `func_addr`@l refers to the high and low parts of the function address.

2.3.6. Function Calling by Pointer

The code generated to support function calling by pointer is the same whether the function being called is an out-of-module or an intra-module function. Figure 2-19 shows an example of function calling by pointer.

Figure 2-19: Function Calling by Pointer

```
lqr     $11, func_ptr          # load pointer to function entry into register 11
bisl    $LR, $11               # call the out-of-module function
        ...
```

2.3.7. Dynamic Stack Space Allocation

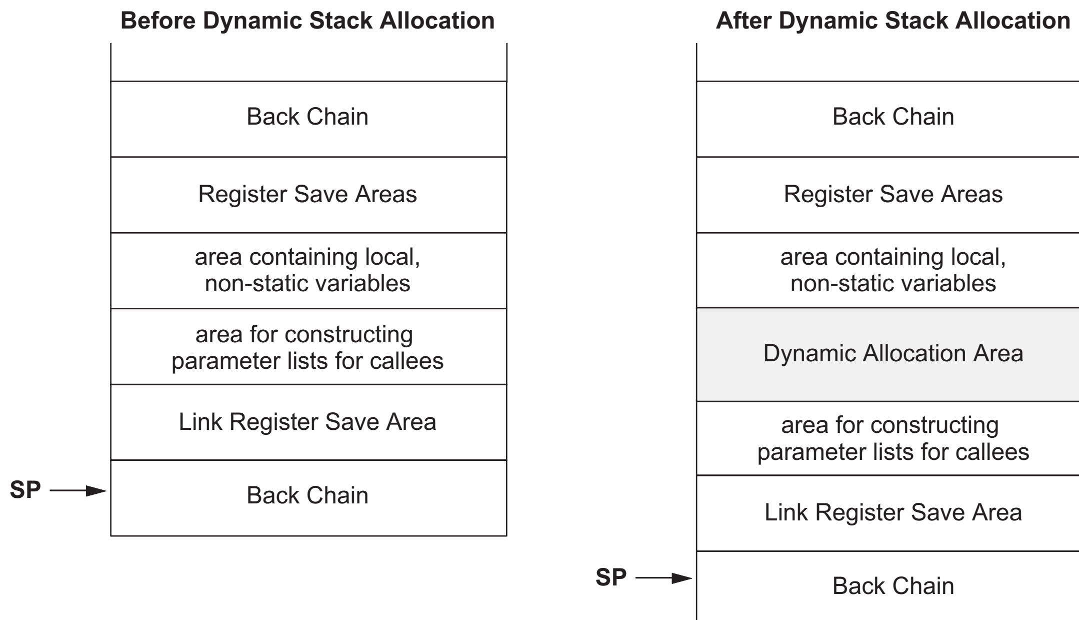
Frames are allocated dynamically on the program stack during program execution. Usually, individual stack frames have static sizes, but the SPU architecture provides facilities for dynamic allocation to support the `alloca` function.

The mechanism for allocating dynamic space is embedded completely within a function. This mechanism does not affect the standard calling sequence. Dynamic stack allocation is accomplished by “opening” the stack immediately above the Parameter List Area (at a higher address). The following steps describe the process in greater detail:

1. After a new stack frame is acquired and before the first dynamic space allocation, a new register (the frame pointer) is set to the value of the stack pointer. The frame pointer is used for references to the function’s local, non-static variables.
2. The amount of dynamic space to be allocated is rounded to a multiple of 16 bytes so that the 16-byte stack alignment is maintained.
3. The stack pointer is decreased by the rounded byte count, and the address of the previous stack frame (the Back Chain) is stored at the word addressed by the new stack pointer.

Figure 2-20 shows the organization of the stack frame before and after dynamic stack allocation.

Figure 2-20: Dynamic Stack Space Allocation



The above process can be repeated as many times as required within a single function activation. When it is time to return, the stack pointer is set to the value of the Back Chain, thus removing all dynamically allocated stack space in addition to the rest of the stack frame. A program must not reference the dynamically allocated stack area after it has been freed.

2.4. Debug Format

The debugging format used in objects targeted for the SPU may be the Debug with Arbitrary Record Format (DWARF). Although this ABI does not specify a particular debug format, all of the systems that implement DWARF must use the definitions described in the sections below.

2.4.1. DWARF Register Number Mapping

Register number mapping must be specified for the SPU registers. Table 2-8 describes the register number mapping for the SPU processor. Some general-purpose registers are reserved for special purposes and are thus accessible using several abbreviations.

Table 2-8: SPU Register Number Mapping

Register Name	Number	Abbreviation
General-Purpose Registers 0-127	0 - 127	R0 - R127
Link Register	0	LR
Stack Pointer	1	SP
Floating-Point Status and Control Register	128	FPSCR

2.4.2. Address Class Code

The DWARF version 2 specifications also require that processor-specific address class codes be defined. As shown in Table 2-9, SPU processors define the address class code.

Table 2-9: SPU Address Class Code

Code	Value	Meaning
ADDR_none	0	No class specified

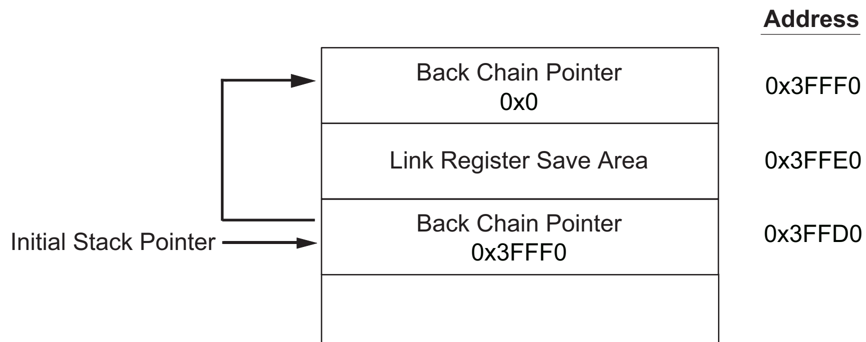
2.5. Operating System Interface

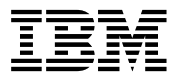
Because the SPU does not generally execute an operating system, it relies on operating system services provided by the controlling PowerPC® Processor Unit (PPU). Operating system interfaces between the PPU and SPU are specified by the CBEA Application Binary Interface specification for the respective operating system.

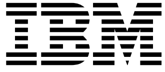
2.5.1. Program Initialization

When an SPU program is first entered, the contents of register `r1` (*SP*) are initialized to the top of the stack. Generally, the top of the stack is a minimal stack located at the largest quadword address. As shown in Figure 2-21, a system with 256-Kbytes of local storage initializes the stack pointer to `0x3FFD0`. This address contains a Back Chain pointer to `0x3FFF0`. The Back Chain pointer at `0x3FFF0` contains a NULL (0) pointer. Space is allocated for the entry function to save the Link Register (address `0x3FFE0`). The contents of all other registers are unspecified. Thus, if a program requires registers to have specified values, it must explicitly set them.

Figure 2-21: Memory Stack







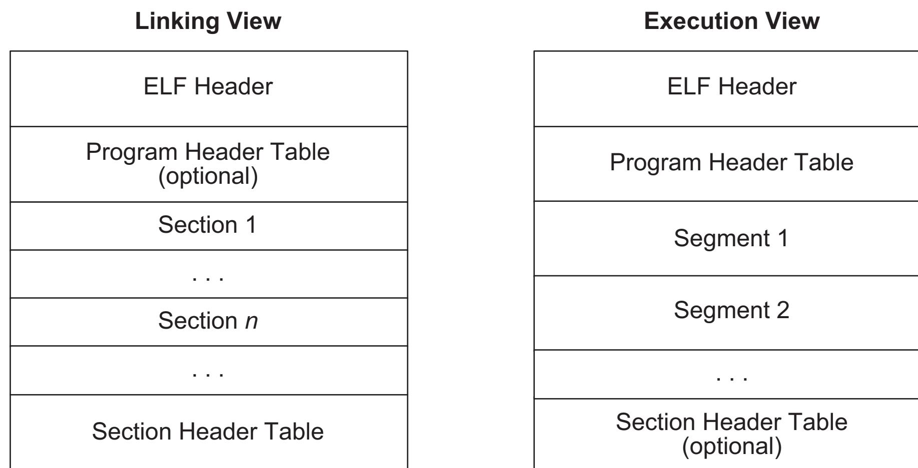
3. Object Files

The SPU object file format must be the Executable and Linking Format (ELF). This document does not completely specify the ELF standard; instead, it provides an overview of ELF while specifying the sections and fields necessary to ensure portability of object files between software tools.

3.1. File Format

Object files are involved in two activities: 1) program linking (building a program) and 2) program execution (running a program). For convenience and efficiency, the object file format provides a parallel view of a file's contents, reflecting the differing needs of these activities. Figure 3-22 shows these two views.

Figure 3-22: Object File Format



3.2. ELF Header

The ELF header contains machine-specific information. Table 3-10 shows the specific information for SPU objects.

Table 3-10: SPU ELF Header Fields

Field	Value	Comments
e_ident[EI_CLASS]	ELFCLASS32	32-bit implementation.
e_ident[EI_DATA]	ELFDATA2MSB	Big endian data encoding.
e_type	ET_NONE	No file type.
	ET_REL	Relocatable file. A relocatable file that holds code and data suitable for linking with objects to create an executable or plug-in file.
	ET_EXEC	Executable file. An executable file that holds a program suitable for execution.
	ET_DYN	Plug-in file. The plug-in file must contain a SPUNAME note section for each named plug-in. See section "2.2.6. Out-of-Module Function Calls" for additional information.
e_machine	EM_SPU	SPU processor identification. The defined value is 23.
e_flags	0	Currently no flags have been defined. Therefore, this member must contain zero.

3.3. Symbols

The global symbols produced by a compiler must not be “mangled”; that is, these symbols must not be prepended by any leading characters.

3.4. Sections

Table 3-11 shows ELF sections that hold program data and code.

Table 3-11: SPU Special Sections

Name	Type	Attributes	Section Contents
.bss	SHT_NOBITS	SHF_ALLOC SHF_WRITE	Uninitialized data that contributes to the program’s memory image. By definition, the program loader initializes the data with zeros when the program begins to run.
.data	SHT_PROGBITS	SHF_ALLOC SHF_WRITE	Initialized data contributing to the program’s memory image.
.text	SHT_PROGBITS	SHF_ALLOC SHF_EXECINSTR	The *text*, or executable instructions, of a program.

The following restrictions apply to allocatable ELF sections that will be loaded into local storage:

- The lower boundary of the section must begin on a 16-byte aligned address.
- The section size should be a multiple of 16 bytes. If the total size of the section contents is not a multiple of 16 bytes, the section will be expanded to the next multiple of 16 bytes. Each byte in the expanded area will be zero filled.

If ELF sections are not loaded into local storage, they do not need to comply with these restrictions.

Because of these restrictions, (1) all loadable ELF segments will begin on a 16-byte boundary; and (2) both memory size and file size will be a multiple of 16 bytes.

This specification defines the minimum requirement. For some CBEA implementations, data-transfer performance can be improved by using a larger segment-alignment constraint, for example, to enable more efficient DMA transfers.

3.5. Relocation

3.5.1. Relocation Types

Relocation entries describe how to change the instructions and data relocation fields. Relocation is performed on a word or a subset of a word. The calculations shown in Table 3-13 assume that the actions are transforming a relocatable file into an executable. Conceptually, the link editor merges one or more relocatable files to form the output file. As part of the process, it first determines how to combine and locate the input files. Next, it updates the symbol values, and then it performs the necessary relocations. Table 3-12 shows the relocation fields and their description.

Table 3-12: Relocation Fields

Field	Description
word32	This specifies a 32-bit field occupying 4 bytes, the alignment of which is 4.
I7	This specifies a 7-bit field contained within bits 11-17 of a word with 4-byte alignment. The other bits of the word are unchanged.
I9	This specifies a 9-bit field contained within bits 7-8 and 25-31 of a word with 4-byte alignment. The other bits of the word are unchanged.
I9I	This specifies a 9-bit field contained within bits 16-17, 25-31 of a word with 4-byte alignment. The other bits of the word are unchanged.

Field	Description
I10	This specifies a 10-bit field contained within bits 8-17 of a word with 4-byte alignment. The other bits of the word are unchanged.
I16	This specifies a 16-bit field contained within bits 9-24 of a word with 4-byte alignment. The other bits of the word are unchanged.
I18	This specifies an 18-bit field contained within bits 7-24 of a word with 4-byte alignment. The other bits of the word are unchanged.

Table 3-13 shows relocation types. (See the notes following Table 3-13 for an explanation of the notational conventions used in the table.)

Table 3-13: Relocation Types

Name	Value	Field[1]	Calculation[2]	Code Generating Example
R_SPU_NONE	0	none	none	-
R_SPU_ADDR10	1	I10*	(S + A) >> 4	lqd $3, symbol($4)
R_SPU_ADDR16	2	I16*	(S + A) >> 2	brasl $LR, function
R_SPU_ADDR16_HI	3	I16	#hi(S + A)	ilhu $3, symbol@h
R_SPU_ADDR16_LO	4	I16	#lo(S + A)	iohl $3, symbol@l
R_SPU_ADDR18	5	I18*	S + A	ila $3, symbol
R_SPU_ADDR32	6	word32	S + A	.word symbol
R_SPU_REL16	7	I16*	(S + A - P) >> 2	brsl $LR, function
R_SPU_ADDR7	8	I7	S + A	cwd $3, symbol($4)
R_SPU_REL9	9	I9*	(S + A - P) >> 2	hbra function, -100
R_SPU_REL9I	10	I9I*	(S + A - P) >> 2	hbr function, $3
R_SPU_ADDR10I	11	I10*	S + A	ai $3, $3, symbol
R_SPU_ADDR16I	12	I16*	S + A	il $3, symbol
R_SPU_REL32	13	word32	S + A - P	.word symbol
R_SPU_ADDR16X	14	I16*	S + A	ilh $3, symbol

[1] Those relocation types whose Field entry in the table contains an asterisk are subject to failure if the value of the relocation does not fit in the allocated bits.

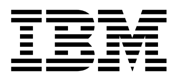
[2] The following notation is used to describe the Calculation entry in the table:

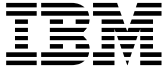
- The letters `A`, `P`, and `S` represent:
  `A`: the addend used to compute the value of the relocatable field.
  `P`: the place (section offset or address) of the storage unit being relocated. This is computed using `r_offset`.
  `S`: the value of the symbol whose index is located in the relocation entry.

- The “+” and “-” symbols denote 32-bit modulus addition and subtraction, respectively. “>>” denotes arithmetic right-shifting (shifting with sign copying) of the value of the left operand by the numbers of bits given by the right operand.

- For relocation types that update the subset of a word, the upper bits must all be the same before being shifted. For relocation types that perform shifting, the shifted number of least significant bits must be 0.

- `#hi(value)` and `#lo(value)` denote the most and least significant 16-bits, respectively, of the indicated value. That is, `#lo(x) = (x & 0xFFFF)` and `#hi(x) = ((x >> 16) & 0xFFFF`.





4. Program Loading and Dynamic Linking

This chapter describes object file structures that relate to program execution. This chapter should be read in conjunction with "3. Object Files."

4.1. Program Header

The program header table is a primary data structure. It contains the location of the segment images within the file and other information necessary to create the memory image for a program.

4.1.1. SPU Environment Note

SPU objects may contain sections of type `SHT_NOTE` with program header elements of type `PT_NOTE` that define the attributes and runtime environment of a SPU program. Table 4-14 and Table 4-15 provide details about the SPU environment note.

Table 4-14: SPU Environment Note

Field	Size (bytes)	Value
namesz	4	8
descsz	4	sizeof(spu_env)
type	4	1
name	8	"IBM SPU"
desc	sizeof(spu_env)	spu_env structure

The SPU environment note contains an instance of the `spu_env` structure. Entries in the `spu_env` structure are shown in Table 4-15.

Table 4-15: spu_env Structure

Type	Name	Description
Elf32_Word	revision	Structure revision number. Initial structure revision number is 1. Future additions to this structure are added to the end, and the revision number is incremented.
Elf32_Word	ls_size	Size of SPU local storage where the program is targeted to run. Specifies the required AMR (Address Memory Range) register setting. A size of 0 indicates that the AMR register must be set to the entire available address range.
Elf32_Word	stack_size	Runtime SPU stack size. Used to establish the Available Stack Space (word element 1 of register `R1`). If the SPU environment is unspecified or if the `stack_size` is specified as zero, the value of Available Stack Space is initialized to `<top_of_stack> - _end`. Otherwise, the Available Stack Space is initialized to `stack_size`.
Elf32_Word	flags	`ELF_SPU_ENCRYPTED` (bit 31) - specifies that the SPU ELF program is encrypted and must be decrypted and authenticated before being executed.

4.1.2. SPU Name Note

An SPU object must be identified with a lookup name string, and this name must be contained within a `SHT_NOTE` with program header elements of type `PT_NOTE`.

Table 4-16 shows the size and values of fields within an SPU name note.

Table 4-16: SPU Name Note

Field	Size (bytes)	Value
namesz	4	8
descsz	4	The number of bytes in the `desc` field. This value must be a multiple of 4 bytes.
type	4	1
name	8	“SPUNAME”
desc	(see `descsz`)	A null terminated look-up string that identifies the path name of the object.

END OF DOCUMENT