NXP Semiconductors
Reference Manual

Document identifier: CWVSPA3ABIREF
Rev. 10.3.x, 07/2020

CodeWarrior Development Studio for VSPA3 Architecture Application Binary Interface (ABI) Reference Manual

Supports: VSPA3

Contents

Figures

Tables

Chapter 1
Introduction

The types of standards covered are as follows:

- Low-level run-time binary interface standards, such as:
  - Processor-specific binary interface, consisting of the instruction set and representation of fundamental data types.
  - Function calling conventions, specifying how arguments are passed and results are returned, how registers are assigned, and how the calling stack is organized.
- Source-level standards, such as:
  - C language, including preprocessor predefines, name mapping, and intrinsic functions.
- Assembler syntax and directives

NOTE

Features defined in this ABI reference manual are mandatory unless specifically stated otherwise. Optional features, if implemented, must conform to the ABI standards.

Chapter 2
Low-Level Binary Interface

The low-level binary interface for the VSPA3 architecture include:

- Processor-specific binary interface, consisting of the instruction set and representation of fundamental data types.
- Function calling conventions, specifying how arguments are passed and results are returned, how registers are assigned, and how the calling stack is organized.

This chapter explains:

2.1 Endian Support

The VSPA architecture supports little-endian implementations. The bytes that form the supported data types are ordered in the memory according to the least significant byte (LSB), located in the lowest address (byte 0).

2.2 Fundamental Data Types

The smallest addressable memory location is a 8-bit data type. The address of the 8-bit data types must be aligned to 8-bits words; the address of the 16-bit data types must be aligned to 16-bits words; the address of the 32-bit data types must be aligned to 32-bits words.

2.2.1 Pointers

Although a pointer takes up a full 32-bit word in a memory, . the data pointers are 21 bits wide and code pointers are 25 bits wide.

Table 1. VSPA C Data Types

Data Type	Size (in bits)	sizeof(T)	Alignment (in bits)
T*	32 (21 bits used)	4	32
T(*)()	32 (25 bits used)	4	32

NOTE

The high bits of a pointer in memory are undefined. Converting an integer type to a pointer and then back to an integer type has an undefined behavior.

NOTE

- Objects can only be accessed using are naturally aligned,
  - For example, a 32-bit access must be 32-bit aligned
  - Misaligned accesses are undefined

2.2.2 Mapping C Data Types to VSPA3 Architecture

VSPA3 architecture maps with the C data types as described in the following table:

Table 2. VSPA3 C Data Types

Data Type	Size (in bits)	sizeof(T)	Alignment (in bits)
_Bool / bool	8	1	8
signed/unsigned char	8	1	8
signed/unsigned short	16	2	16
signed/unsigned int	32	4	32
signed/unsigned long	32	4	32
signed/unsigned long long	64	8	64
___fx16 (16- fixed-point)	16	2	16
__fp16 (16-bit floating point)	16	2	16
float	32	4	32
double	64	8	64
long double	64	8	64
_Imaginary float	32	4	32
_Imaginary double	64	8	64
_Imaginary __fp16 / __fx16	16	2	16
_Imaginary long double	64	8	64
_Complex float	64	8	64
_Complex double	128	16	128
_Complex __fp16 / __fx16	32	4	32
_Complex long double	128	16	128

NOTE

VSPA3 supports 16-bit half-precision floating point (modified IEEE-754 format), 32-bit single-precision floating point (modified IEEE-754 format), and 64-bit double-precision floating point (modified IEEE-754 format).

NOTE

The `sizeof()` operator returns `1` for char, `2` for short, and 4 for int/long/float data type. This is because an 8-bit word is the smallest addressable unit.

2.3 Aggregates and Unions

Aggregates (structures and arrays) and unions assume the alignment of their most strictly aligned component; that is, 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 may require padding to meet size and alignment constraints:

- A structure or union with a size > 2 is always at least 32-bit word aligned and padded accordingly. So sizeof(<struct or union>) will always be a multiple of 4.
- 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 may require internal padding, depending on the previous member.
- If necessary, a structure's size is increased to make it a multiple of the structure's alignment. This may require tail padding, depending on the last member.

```
struct X1 { short x; };            // sizeof(X1) == 2; alignof(X1) == 2
struct X2 { short x, y; };        // sizeof(X2) == 4; alignof(X2) == 4
struct X3 { short x, y, z; };    // sizeof(X3) == 8; alignof(X3) == 4
```

2.4 Bit Fields

C struct and union definitions may have bit fields, defining integral objects with a specified number of bits. Bit fields exhibit the property of same signedness as their underlying data type. In addition to the following rules, bit fields follow the same size and alignment rules as other structure and union members:

- The bit fields are allocated from right to left (least to most significant).
- A bit field must entirely reside in a storage unit appropriate for its declared type. Thus, a bit field never crosses its unit boundary.
- The bit fields share a storage unit with other structure and union bit field members if and only if there is sufficient space within the storage unit.
- Unnamed bit fields' types do not affect the alignment of a structure or union, although an individual bit field's member offsets obeys the alignment constraints. An unnamed, zero-width bit field shall prevent any further member, bit field or other, from residing in the storage unit corresponding to the type of the zero-width bit field.

2.5 Function Calling Conventions

Compilers must support the conventions described in following sections:

- Argument Passing and Return Values
- Using Registers in Calling Convention
- Stack Frame Layout

2.5.1 Argument Passing and Return Values

The standard calling sequence requirements apply only to global functions. Local functions that are not reachable from other compilation units may use different conventions. Nonetheless, it is recommended that all functions use the standard calling sequences when possible. The C language programs follow the given conventions:

- The registers g0, g1, and a0 are available for return values.
- A function returning a function-pointer type uses g0.
- A function returning a non-function pointer type uses the a0 register to return the result.
- A function returning a non-pointer type T where `sizeof(T) == 1` (8-bit word) or sizeof(T) == 2 (16-bit word) or sizeof(T) == 4 (32-bit word) , uses the g0 register to return the result. 8-bit results are returned in bits 7-0, bits 31-8 are undefined. 16-bit results are returned in bits 15-0, bits 31-16 are undefined.
- A function returning a non-pointer type T where `sizeof(T) == 8` (two 32-bit words), uses the g0 and g1 registers to return the result. The lower address word is returned in the g0 register and the higher address word is returned in g1 register.
- All other functions use a caller allocated stack area to return the result in memory. The address of the stack area is passed as an invisible parameter using the a0 register.

- The registers g0 - g5 and a0 - a5 (a1 - a5 for functions returning results in memory) are available for parameter passing. Parameter registers are assigned in left-to-right order.
- Arguments that are part of a variable argument list are always passed by-value on the stack.
- A non-static C++ member function has an invisible class pointer parameter before all other parameters.
- A function-pointer type argument is passed in the first available "gX" register.
- A non-function pointer type argument is passed in the first available register. If there are no "aX" registers left, it is passed on the stack.
- An argument with the non-pointer type T where `sizeof(T) == 1` (8 -bit word) or sizeof(T) == 2 (16-bit word), is passed in the first available "gX" register. If there are no "gX" registers left, it is passed on the stack. 16-bit arguments are passed in bits 15-0, bits 31-16 are undefined. 8-bit arguments are passed in bits 7-0, bits 31-8 are undefined.
- An argument with the non-pointer type T where `sizeof(T) == 8` (two 32-bit words) is passed in the first available "gX" register pair (g0:g1, g1:g2, g2:g3, g3:g4, g4:g5). If there are no "gX" register pairs left, it is passed on the stack. The lower address word is passed in the lower register and the higher address word is passed in the higher register.
- Stack arguments are passed by-value in a caller allocated argument stack area. This argument stack area starts at SP upon entry of a function. The stack arguments are stored at properly aligned addresses, as if they had been pushed on the stack in right-to-left order. For more details on Stack arguments, see Stack Frame Layout.

2.5.2 Using Registers in Calling Convention

The following registers are saved by the caller:

- g0- g7
- a0- a11

The following registers are saved by the callee, if actually used:

- g8- g11
- a12- a19

The register `SP` is used as Stack Pointer, whereas, the register `a19` is used as Frame Pointer. In a function that does not require a frame pointer, the frame pointer register is allocated for ordinary usage. Currently, the CodeWarrior compiler does not use frame pointers.

2.5.3 Stack Frame Layout

The stack pointer points to the top (high address) of the stack frame. Space at higher addresses than the stack pointer is considered invalid and may actually be un-addressable. Pushing a word onto the stack moves the stack pointer to a higher address. The stack pointer value must always be aligned to a DMEM line. The outgoing arguments area is located at the top (higher addresses) of the frame. The caller puts argument variables that do not fit in registers into the outgoing arguments area. If all arguments fit in registers, this area is not required. A caller may allocate outgoing arguments space sufficient for the worst-case call, use portions of it as necessary, and not change the stack pointer between calls. Local variables that do not fit into the local registers are allocated space in the local variables area of the stack. If there are no such variables, this area is not required. The caller must reserve stack space for return variables that do not fit in registers. This return buffer area is typically located with the local variables, but it may be the address of a global variable. This space is typically allocated only in functions that make calls returning structures. The following figure shows the stack frame layout.

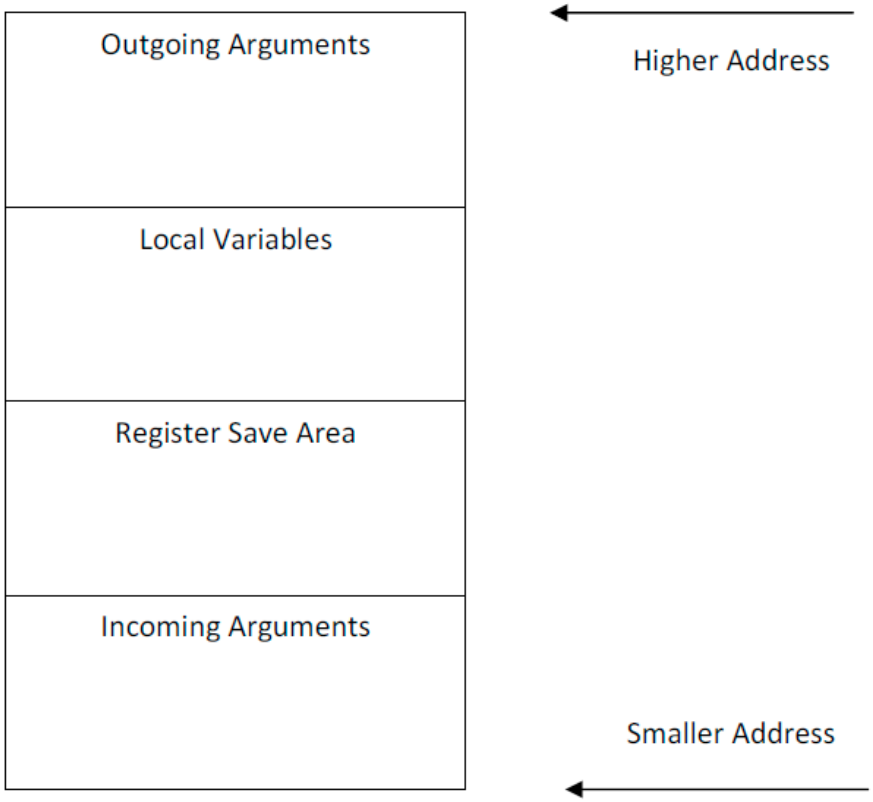

Figure 1. Stack Frame Layout

2.6 VSPA Modes

CC update mode is disabled at startup (`set.creg 4, 0`) and assumed to be off while executing C code. The VCPU unit can run in either 8-bit word addressing or 16-bit word addressing. The IPPU unit can also run in either 16-bit word addressing or 32-bit word addressing. The only supported mode for VSPA3 is 8-bit addressing for VCPU and 16-bit addressing for IPPU. This mode is enabled at startup and assumed to be on while executing C code. For VCPU the mode is enabled by setting the vspa_mode creg field (set.creg 22, 0b10). For IPPU the mode is enabled by setting the ippu_legacy_mem_addr field of the IPPU_CONTROL register (`mv g0, 0x1000000`; `mvip 0x1c0, g0, 0x1000000`). In order to take effect, this operation should be done while the IPPU is not in the busy state.

NOTE

The A0/A1/A2/A3 ranges cannot be used while executing C code. They must be set to the full data memory range.

Other VSPA modes are undefined.

2.7 VSPA Syscall values

It lists the syscall values that are defined and accepted in the VSPA environment.

Following syscall values are defined and accepted:

- SYSCALL_NONE 0
- SYSCALL_open 1
- SYSCALL_close 2
- SYSCALL_read 3

- SYSCALL_write 4
- SYSCALL_lseek 5
- SYSCALL_unlink 6
- SYSCALL_rename 7
- SYSCALL_clock 9
- SYSCALL_time 10
- SYSCALL_system 14

Chapter 3
High-Level Language Issues

3.1 C Preprocessor Predefines

The VSPA3 compiler supports the following macros:

- __VSPA__

  An implicitly defined macro that expands to 1 if the compiler is generating the object code for VSPA family targets.

- __VSPA3__

  An implicitly defined macro that expands to 1 if the compiler is generating the object code for VSPA3 targets.

- __ASSEMBLER__

  A Preprocessor macro that gets defined to value 1 when the compiler is preprocessing an assembly file, i.e `.sx` files.

- __AU_COUNT__

  A Preprocessor macro that specifies the number of VSPA AUs (2-64).

- __VSPA_SP__

  An implicitly defined macro that expands to 1 if the compiler is generating the object code for VSPA target with a single precision core.

- __VSPA_DP__

  An implicitly defined macro that expands to 1 if the compiler is generating the object code for VSPA target with a double precision core.

3.2 Access to Architectural Features

The supported intrinsic functions allows access to the hardware resources from a C application without using assembly language instructions.

NOTE

To use the supported intrinsic features, include the standard header file `vspa/intrinsics.h` in the source files `#include <vspa/intrinsics.h>.`.

NOTE

For more details on intrinsic functions, see *VSPA Intrinsics Manual* .

Chapter 4
Object File Format

The ELF is used for representing the binary application to the system.

NOTE

For more information on ELF, see *Tools Interface Standards (TIS) Executable and Linking Format (ELF) Specification*, Version 1.1.

This chapter explains:

- Interface Descriptions
- ELF Header
- Sections
- Symbol Table
- Relocation
- Program Headers
- Debugging
- VSPA Memory spaces

The chapter focuses on the interface for relocatable and executable programs. A relocatable program contains the code suitable for linking and to create another relocatable program or executable program. An executable program contains binary information suitable for loading and execution on a target processor.

4.1 Interface Descriptions

ELF represents two views of binary data, as shown in .

- Linking View: Provides data in a format suitable for incremental linking into a relocatable file or final linking to an executable file.
- Execution View: Provides binary data in a format suitable for loading and execution.

An ELF header is always present in both the views of the ELF file. In the linking view, sections are the main entity in which information is presented. A section header table provides information for interpretation and navigation through each section.

In the execution view, segments are the primary sources of information. Sections may be present but are not required. A program header table provides information for interpretation and navigation through each segment.

NOTE

For more details, see *ELF version 1.1 specifications*.

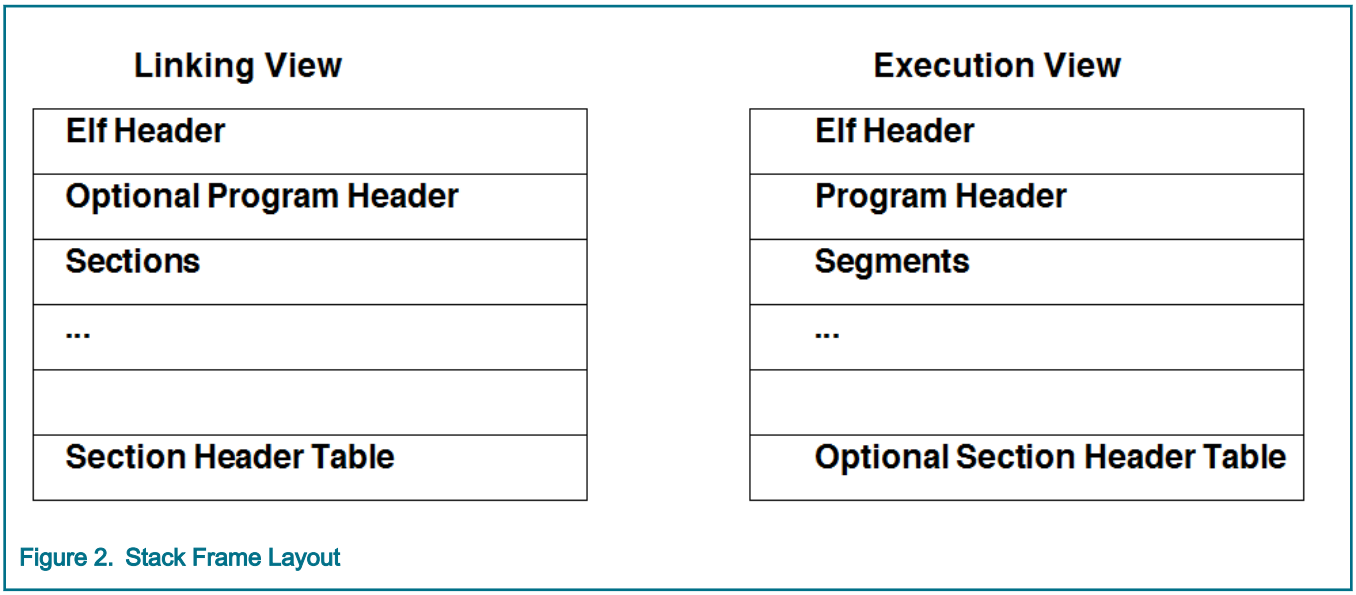

Figure 2. Stack Frame Layout

4.2 ELF Header

The following listing shows the ELF header structure.

Listing: ELF Header Structure

```
typedef struct {
unsigned char e_ident[EI_NIDENT];
Elf32_Half e_type;
Elf32_Half e_machine;
Elf32_Word e_version;
Elf32_Addr e_entry;
Elf32_Off e_phoff;
Elf32_Off e_shoff;
Elf32_Word e_flags;
Elf32_Half e_ehsize;
Elf32_Half e_phentsize;
Elf32_Half e_phnum;
Elf32_Half e_shentsize;
Elf32_Half e_shnum;
Elf32_Half e_shstrndx;
} Elf32_Ehdr;
```

The following listing shows an example of the VSPA -specific code:

Listing: VSPA Specifics

```
e_ident[EI_CLASS] = ELFCLASS32
e_ident[EI_DATA] = ELFDATA2LSB (little-endian memory mode)
e_machine: 16584 (0x40c8)
```

4.3 Sections

Sections are the main components of an ELF file. The section headers define all the information about a section. The following listing shows an example of the section header.

Listing: Section Header Structure

```
typedef struct {
Elf32_Word sh_name;
Elf32_Word sh_type;
Elf32_Word sh_flags;
Elf32_Addr sh_addr;
Elf32_Off sh_offset;
Elf32_Word sh_size;
Elf32_Word sh_link;
Elf32_Word sh_info;
Elf32_Word sh_addralign;
Elf32_Word sh_entsize;
} Elf32_Shdr;
```

lists the sections used in VSPA ELF binaries.

NOTE

The section names listed in this table are case sensitive and are reserved for the system.

Table 3. VSPA ELF sections

Name (sh_name)	Type (sh_type)	Flags (sh_flags)	Purpose
`.text`	SHT_PROGBITS	SHF_ALLOC, SHF_EXECINSTR	VCPU Executable instructions in VCPU program RAM
`.rom`	SHT_PROGBITS	SHF_ALLOC,SHF_EXECINSTR	VCPU Executable instructions in VCPU program ROM
`.data`	SHT_PROGBITS	SHF_ALLOC,SHF_WRITE	Initialized data on VCPU
`.rodata`	SHT_PROGBITS	SHF_ALLOC	Read-only, Initialized data on VCPU
`.bss`	SHT_NOBITS	SHF_ALLOC, SHF_WRITE	Uninitialized data on VCPU (The contents of the `.bss` section are zeroed when loaded.)
`.vbss`	SHT_NOBITS	SHF_ALLOC, SHF_WRITE	Uninitialized data on VCPU (The contents of the `.vbss` section are zeroed when loaded.)
`.ibss`	SHT_NOBITS	SHF_ALLOC, SHF_WRITE	Uninitialized data on IPPU (The contents of the `.ibss` section are zeroed when loaded.)
`.relasection`	SHT_RELA	None	Relocation information for section (see "")
`.symtab`	SHT_SYMTAB	None	Symbol Table
`.shstrtab`	SHT_STRTAB	None	Section name string table
`.strtab`	SHT_STRTAB	None	General purpose string table

Table continues on the next page...

Table 3. VSPA ELF sections (continued)

Name (sh_name)	Type (sh_type)	Flags (sh_flags)	Purpose
.debug_abrev	SHT_PROGBITS	None	Abbreviation tables (This information is in DWARF2 format)
.debug_arranges	SHT_PROGBITS	None	Address range tables (This information is in DWARF2 format)
.debug_frame	SHT_PROGBITS	None	Call frame information (This information is in DWARF2 format)
.debug_info	SHT_PROGBITS	None	Debugging information entries (This information is in DWARF2 format)
.debug_line	SHT_PROGBITS	None	Line number information (This information is in DWARF2 format)
.debug _loc	SHT_PROGBITS	None	Location lists (This information is in DWARF2 format)
.debug_macinfo	SHT_PROGBITS	None	Macro information (This information is in DWARF2 format)
.debug_pubnames	SHT_PROGBITS	None	Global name tables (This information is in DWARF2 format)

4.3.1 Special Sections

The sections .vbss and .ibss are generated by the linker. All VCPU_COMMON symbols will be allocated within .vbss and all IPPU_COMMON symbols will be allocated within .ibss. For more details, see Symbol Table

4.4 Symbol Table

An object file's symbol table holds information needed to locate and relocate a program's symbolic definitions and references.

The following listing shows a symbol table entry.

Listing: Symbol Table Entry

```
typedef struct {
Elf32_Word st_name;
Elf32_Addr st_value;
Elf32_Word st_size;
unsigned char st_info;
unsigned char st_other;
Elf32_Half st_shndx;
} Elf32_Sym;
```

The symbols in ELF object files convey specific information to the linker and loader. The VSPA ELF uses two additional common symbol types.

Table 4. VSPA Additional Symbol Types

Symbol Type	Description
VCPU_COMMON	The symbol labels a common block, which is used by the part of the program running on the VCPU that has not been allocated. The symbol's value gives alignment constraints, similar to a section's `sh_addralign` member. The link editor will allocate the storage for the symbol at an address that is a multiple of `st_value` in the '`.vbss`' section. The symbol's size specifies how many bytes are required.
IPPU_COMMON	The symbol labels a common block, which is used by the part of the program running on the IPPU that has not been allocated. The symbol's value gives alignment constraints, similar to a section's `sh_addralign` member. The link editor will allocate the storage for the symbol at an address that is a multiple of `st_value` in the '`.ibss`' section. The symbol's size specifies how many bytes are required.

4.5 Relocation

Each section that contains relocatable data has a corresponding relocation section of type `SHT_RELA`. The `sh_info` field of the relocation section defines the section header index of the section (hence referred as the "data section") to which the relocations apply. The `sh_link` field of the relocation section defines the section header index of the associated symbol table. If section names are used, the name of the relocation section is `.rela` prepended to the name of the data section.

A relocation entry is defined by the `Elf32_Rela` structure and associated macros as shown in the code listed below. The `r_offset` field defines an offset into the data section to which the individual relocation applies. The `r_info` field specifies both the type of the relocation and the symbol used in computation of the relocation data.

The relocation type is extracted from the `r_info` field using the `ELF32_R_TYPE` macro and the symbol number is extracted using the `ELF32_R_SYM` macro. The `r_info` field is synthesized from the relocation type and symbol number using the `ELF32_R_INFO` macro.

The " **Relocation Value** " is the value to be stored at the location defined by the `r_offset` field in the format specified by the relocation type. In the relocation value is computed by adding the signed value of the `r_addend` field to the value of the symbol indicated by the symbol number. Symbol number **zero** is treated as absolute zero where the relocation value is simply the value of the `r_addend` field.

Listing: Relocation Entry Defined with Elf32_Rela

```
typedef struct {
Elf32_Addr r_offset;
Elf32_Word r_info;
Elf32_Sword r_addend;
} Elf32_Rela;
#define ELF32_R_SYM(i) ((i)>>8)
#define ELF32_R_TYPE(i) ((i)&0xff)
#define ELF32_R_INFO(s,t) (((s)<<8)|((t)&0xff))
```

4.5.1 Relocation Types

Device-specific relocations describe how a memory location should be patched by the linker. An ordinary relocation encodes exactly one instruction operand (or, for data relocations, exactly one data value). The linker must ensure that the operand meets the range and alignment requirement specified by the relocation. The following fields appear in each definition:

- **Type** : Specifies the value extracted using `ELF32_R_TYPE`, both as a number and as a standard C preprocessor symbol. A brief abstract of the relocation follows in parentheses.

- **Size** : Specifies the number of bits used to represent the relocation value. If the operand range is a subset of the values that can be represented in these bits, that restriction is indicated in parentheses.
- **Bit position** : Specifies the bit position of the relocation value within the full VLIW instruction of the address location that is being relocated.

  For example, a 17 bit relocation with LSB bit offset 2 with the VLIW instruction will appear as:

  ```
  0000 0000 0000 0000 0000 0000 0000 0000 0000 0XXX XXXX XXXX XXXX XX00
  ```

- **Shift** : Specifies the number of bits the relocation value is right-shifted before it is encoded.
- **Applies To** : Specifies the instructions or directives that generate this relocation.

The following table lists and defines the relocation types.

Table 5. Relocation Type Definitions for VSPA3

Type	Size	LSB bit offset (in total bits)	Shift	Instructions/Directives Generating This Relocation
R_VSPA_8	8	0 (in 8 bit field)	0	.1byte (debug section relocations)
R_VSPA_16	16	0 (in 16 bit field)	0	.2byte (debug section relocations)
R_VSPA_32	32	0 (in 32 bit field)	0	.4byte or `.word` (debug section relocations)
R_VSPA_HW_LO_19	19	6 (in 64 bit field)	0	lower OpS ld h,lu19 st lu19,h
R_VSPA_HW_HI_19	19	35 (in 64 bit field)	0	upper OpS ld h,lu19 st lu19,h
R_VSPA_PMEM_25	25	24 (in 64 bit field)	2	jmp(.cc) lu25 jsr(.cc) lu25 loop_break(.cc) lu25 swi(.cc) lu16
R_VSPA_PMEM_25_OFST	25	24 (in 64 bit field)	2	jmp(.cc) lu25 jsr(.cc) lu25
R_VSPA_LAB_32	32	2 (in 64 bit field)	0	add(.ucc)(.cc) gX, gY, I32 (sp allowed) add(.ucc)(.cc) gX, I32 (sp allowed) and(.cc) gX, gY, I32 (sp allowed) clrip lu9, I32 cmp(.cc) gX, I32 (sp allowed)

Table continues on the next page...

Table 5. Relocation Type Definitions for VSPA3 (continued)

Type	Size	LSB bit offset (in total bits)	Shift	Instructions/Directives Generating This Relocation
				fadd(.ucc)(.cc) gX, gY, I32 fdiv(.cc) gX, gY, I32 fdiv gX, I32 floatx2n gX, gY, I32 fmul(.cc) gX, gY, I32 fmul gX, I32 fsub(.ucc)(.cc) gX, gY, I32 lfsr gX, gY, I32 mpy(.cc) gX, gY, I32 mpy(.cc) gX, I32 mv(.cc) gX, I32 (sp allowed) mvip Iu9, I32 mvip Iu9, gX, I32 mvip gX, Iu9, I32 or(.cc) gX, gY, I32 or(.cc) gX, I32 push I32 setip Iu19, I32 st Iu19, I32 stw Iu19, I32 sub(.ucc)(.cc) gX, gY, I32 (sp allowed) sub(.ucc) gX, I32 (sp allowed) xor(.cc) gX, gY, I32 xor(.cc) gX, I32
R_VSPA_LAB_IND_32	32	0 (in 64 bit field)	0	`.word` (for symbolic relocations)
R_OCRAM_LAB_32	64	0	0	This relocation type is internal to the linker to handle on-chip shared RAM access from VSPA through 'mv gX, I32' instruction
R_OCRAM_LAB_IND_32	32	0 (in 64 bit field)	0	This relocation type is internal to the linker to handle on-chip shared RAM access from VSPA through .word

Table continues on the next page...

Table 5. Relocation Type Definitions for VSPA3 (continued)

Type	Size	LSB bit offset (in total bits)	Shift	Instructions/Directives Generating This Relocation
				assembler directive with symbolic reference; for example '.word _ocram_sym_'
R_VSPA_LAB_22	22	24 (in 64 bit field)	0	cmp aX, I22 mv aX, I22 set.range range, aXg, I22
R_VSPA_SP_LO_20	20	2 (in 64 bit field)	2	mv sp, I20
R_VSPA_SP_HI_20	20	31 (in 64 bit field)	2	mv sp, I20

R_VSPA_LAB_S_19	19	24 (in 64 bit field)	0	add aX, I19
				add aY, aX, I19
				add aY, sp, I19

R_VSPA_DMEM_20	20	0 (in 64 bit field)	2	st I20, I32 stw I20, I32
R_VSPA_DMEM_21	21	34 (in 64 bit field)	1	sth I21, I16
R_B_VSPA_DMEM_22	22	2 (in 64 bit field)	0	ld gX, I22 ldb(.s) gX, I22 ldh(.s) gX, I22 ldw(.s) gX, I22 st I22, gX stb I22, gX sth I22, gX stw I22, gX
R_VSPA_DMEM_22	22	32 (in 64 bit field)	0	stb I22, I8
R_VSPA_LAB_18	18	24 (in 64 bit field)	0	ld(.u) aYg, sp({I18}) ld(.u) gY, (aXg({I18})) ld(.u) gY, (aXg)({I18}) ldb(.u)(.s) gY, aXg({I18}) ldh(.u)(.s) gY, aXg({I18}) ldh(.u)(.s) gY, sp({I18}) st(.u) (aXg({I18})), gY st(.w) (aXg({I18})) st(.u) (sp({I18})), aYg

Table continues on the next page...

Table 5. Relocation Type Definitions for VSPA3 (continued)

Type	Size	LSB bit offset (in total bits)	Shift	Instructions/Directives Generating This Relocation
				st(.u) (sp)({l18}), aYg stb (aXg)({l18}), gY stb(.u) (aXg({l18})), gY sth (aXg)({l18}), gY sth(.u) (aXg({l18})), gY

4.5.2 Relocation Stack

When the relocation value cannot be expressed as a simple symbolic value with an addend, then there are four special relocation types to evaluate an arbitrary expression on a relocation stack. These four relocation types (`push_pc, push, oper, pop`) are referred to as **extended relocations.** Other relocation types are ordinary relocations.

A relocation stack is a standard last-in-first-out data structure that contains 32-bit values. There is no arbitrary limit set on the depth of the relocation stack of a hosted environment. Where as, an embedded environment may impose any limit on the stack depth or omit the relocation stack entirely (effectively, a maximum stack depth of zero),

The following relocation types:

- `252 (R_VSPA_PUSH_PC)` - Indicates the sum of the symbol value (the value of symbol number zero is zero), the signed `r_addend` value, and the current location counter value should be pushed onto the relocation stack.
- `253 (R_VSPA_PUSH)`- Indicates the sum of the symbol value (the value of symbol number zero is zero), and the signed `r_addend` value should be pushed onto the relocation stack.
- `254 (R_VSPA_OPER)` - Defines an operation to be performed on one or more stack values. The operation is specified by the sum of the symbol value (the value of symbol number zero is zero), and the signed `r_addend` value should be pushed onto the relocation stack.
- `255 (R_VSPA_POP)` - Indicates the end of a relocation expression, to be relocated using an ordinary relocation type from the below table. The relocation type is specified by the sum of the symbol value (the value of symbol number zero is zero) and the signed `r_addend` value.

NOTE

Operations are shown in the table below. In the table, Stack0 indicates the value on the top of the stack, and Stack1 indicates the value one level beneath the top of the stack.

Table 6. Operations Performed on Relocation Values

Relocation Value	Before		After	Operation
	Stack0	Stack1	Stack0	
8	Y	X	X-Y	Subtraction

When the `R_VSPA_POP` operation is encountered, there should be exactly one value on the stack. This value becomes the new relocation value for the ordinary relocation type specified in the `R_VSPA_POP` relocation.

The relocation engine will ensure that the stack is empty after the `R_VSPA_POP`, before an ordinary relocation, and after the linking is complete. A sequence of relocations which causes a stack underflow does not conform to the ABI.

The symbolic relocation of "`.data`" section is:

```
.data

.word _a - _b
```

The following listing shows the relocations generated for the above section:

Listing: Program Header

```
Offset           Addend        Type     Symbol
-------------------------------------------------------
0x00000000      0x00000000     252      _a
0x00000000      0x00000000     252      _b
0x00000000      0x00000008     254      (symbol 0)}
0x00000000      0x0000000c     255      (symbol 0)
```

NOTE

The above relocations mean that the symbols "_a" and "_b" are pushed on the relocation stack, the operation executed is a subtraction (value 8 in the addend) and the result is a VSPA relocation of type `12 (R_VSPA_LAB_IND_32)`, specified in the addend.

4.6 Program Headers

Program headers are used to build an executable image in memory and are useful for executable files only. While section headers may or may not be included in executable files, program headers are always present.

The following listing shows an example of program header.

Listing: Program Header

```
typedef struct {
Elf32_Word p_type;
Elf32_Off p_offset;
Elf32_Addr p_vaddr;
Elf32_Addr p_paddr;
Elf32_Word p_filesz;
Elf32_Word p_memsz;
Elf32_Word p_flags;
Elf32_Word p_align;
}Elf32_Phdr;
```

The program header members are described as follows:

- `p_type`: Describes the type of program header. Only PT_LOAD and PT_NOTE are recognized as program header types.
- `p_offset`: Sets the offset from beginning of file to first byte of segment.
- `p_vaddr`: Describes the virtual address in memory of the first byte of the segment.
- `p_paddr`: Specifies the physical address in memory of the first byte of the segment.
- `p_filesz`: Shows the number of bytes in segment's file image.
- `p_memsz`: Sets the number of bytes in segment's memory image.
- `p_flags`: Shows the flags relevant to the segment. The defined flags are PF_R, PF_W, and PF_X.
- `p_align`: Describes the segment alignment requirements in file and memory.

4.7 Debugging

Tools for the VSPA architecture must use the **Debug With Arbitrary Record Format** (DWARF) debugging format, as defined in the *Tool Interface Standard (TIS) DWARF Debugging Information Format Specification, Version 2.0*.

4.7.1 DWARF Register Number Mapping

outlines the register number mapping for the VSPA processor.

Table 7. VSPA Register Number Mapping

Register Name	Number	Abbreviation
General purpose register	0-11	g0-g11
Address storage register	12-27	as0-as15
Address register	28-31	a0-a3
Stack pointer	32	sp
Dummy return register	36	ret

4.8 VSPA Memory Spaces

This sections explains the Memory Space encoding enhancements. The address encoding for each of the VSPA memories are no longer mixed with IDs, for identifying different physical regions.

The VSPA memory space is divided into the following:

- VCPU Program
- VSPA Data
- IPPU Program
- Physical (OCRAM)
- LUT

Each memory space is allocated with a standard ID. The user must specify the memory space ID while placing a section into it.

The memory space encoding takes into consideration the following aspects:

Table 8. Memory spaces

ID	Memory space	Start address	Size
0	VCPU Program: `vcpu_pram`	`0x0`	`0x08000000`
1	VSPA Data: `vcpu_dram`	`0x0`	`0x0080000`
2	IPPU Program: `ippu_pram`	`0x0`	`0x00001000`
4	Physical: `ocram`	`0x10000000`	`0xF0000000`
5	LUT	`0x0`	`0xF0000000`

Chapter 5
Assembler Syntax and Directives

- Assembler Significant Characters
- Assembler Directives

In addition to the standard assembler directives and syntax specified in the GNU assembler documentation, VSPA assembler also support the following assembler directives, special characters, and syntax specific to VSPA .

5.1 Assembler Significant Characters

VSPA assembler supports several significant one- and two-character sequences that can have multiple meanings depending on the context. These characters are listed in the following table.

Table 9. Assembler Significant Characters

Character	Meaning
`;`	Use this character as instruction delimiter in multi-instructions per line
`//`	Use this character as comment delimiter.
`{ }`	Use this character as instruction grouping delimiter

5.2 Assembler Directives

The VSPA assembler supports all the standard assembler directives listed in GNU assembler manual, *as.pdf*. The following table lists the various VSPA target specific assembler directives and their respective type.

Table 10. Assembler Directives

Type	Directive	Description
Assembly Control	`.vcpu`	Assembles the instructions for VCPU processing unit.
	`.ippu`	Assembles the instructions for IPPU processing unit.
Symbol Definition	`.vcomm symbol, length [, align]`	Defines common symbol named symbol for the VCPU core. The length parameter specifies the number of bytes allocated for the symbol. The optional align parameter specifies the desired alignment of the symbol expressed in bytes.
	`.icomm symbol, length [, align]`	Defines common symbol named symbol for the IPPU core. The length parameter specifies the number of bytes allocated for the symbol. The optional align parameter specifies the desired alignment of the symbol expressed in bytes.

Table continues on the next page...

Table 10. Assembler Directives (continued)

Type	Directive	Description
Data definition and Storage Allocation	`.hfixed flonums`	Assembles VSPA 16-bit signed magnitude fixed point constants (r format). Note that this directive accepts only either zero or even number of float numbers.
	`.hfloat flonums`	Assembles IEEE 16-bit binary floating point constants (h format). Note that this directive accepts only either zero or even number of float numbers.

NOTE

Use `.byte` directive in the data initialization of symbol's definition, as the size of all the data types in VSPA3 build tools is 8 -bit. Use .hword directive for 16-bit data. Use .word directive for 32-bit data.

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including “typicals,” must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

While NXP has implemented advanced security features, all products may be subject to unidentified vulnerabilities. Customers are responsible for the design and operation of their applications and products to reduce the effect of these vulnerabilities on customer’s applications and products, and NXP accepts no liability for any vulnerability that is discovered. Customers should implement appropriate design and operating safeguards to minimize the risks associated with their applications and products.

For more information, please visit: http://www.nxp.com
For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 07/2020
Document identifier: CWVSPA3ABIREF