CodeWarrior Development Studio for StarCore 3900FP DSPs Application Binary Interface (ABI) Reference Manual

Document Number: CWSCABIREF
Rev. 10.9.0, 06/2015

Contents

Section number **Title** **Page**

Chapter 1
Introduction

Chapter 2
Low-level Binary Interface

Chapter 3
High-level Language Issues

Chapter 4
Object File Format

Chapter 5
Assembler Syntax and Directives

Chapter 1
Introduction

The Application Binary Interface (ABI) defines a set of standards to ensure interoperability among conforming CodeWarrior for StarCore DSPs software components, such as, compilers, assemblers, linkers, debuggers, and assembly language code. These standards cover run-time aspects as well as object formats to be used by compatible tool chains from StarCore and third-party tools developers. A benefit of this standard definition is interoperability of conforming tools so that users can select the best tool for each phase of the application development cycle, rather than being constrained to using an entire tool chain. Another benefit is compatibility of conforming libraries. Programmers can build compatible binary libraries and assembly code libraries and be assured of their continued compatibility over time.

This chapter describes:

1.1 Standards Covered

This section lists the standards that are required to to ensure interoperability:

This document addresses the following types of standards:

- Low-level run-time binary interface standards
  - Processor-specific binary interface (the instruction set and representation of fundamental data types)
  - Function calling conventions (how arguments are passed and results returned, how registers are assigned, and how the calling stack is organized)
- Source-level standards

  - C language (preprocessor predefines, name mapping, and intrinsics)
  - Assembler syntax and directives
- Object-file binary interface standards
  - Header convention
  - Section layout
  - Relocation information format
  - Debugging information format
- Library standards
  - Compiler run-time libraries (integer routines and floating-point routines)

NOTE

Features defined in the *SC39xx* ABI are mandatory unless specifically stated otherwise. Optional features, if implemented, must conform to the ABI.

1.2 Accompanying Documentation

The Documentation Roadmap page describes the documentation included in this version of CodeWarrior Development Studio for StarCore DSPs.

You access Documentation Roadmap by:

- a shortcut link on the Desktop that the installer creates by default, or
- opening `START_HERE.html` in `CWInstallDir\SC\Help`.

1.3 Conventions

This document uses certain conventions to assist you in identifying, locating, and understanding information.

This section explains the following types of conventions used:

- Numbering Systems
- Typographic Notation
- Special Terms

1.3.1 Numbering Systems

The below table describes suffixes that identify different numbering systems.

Table 1-1. Different Numbering Systems

This suffix	Identifies a
b	Binary number. For example, the binary equivalent of the number 5 is written 101b.
d	Decimal number. Decimal numbers are followed by this suffix only when the possibility of confusion exists. In general, decimal numbers are shown without a suffix.
h	Hexadecimal number. For example, the hexadecimal equivalent of the number 60 is written 3Ch.

1.3.2 Typographic Notation

The following table lists the typographic notations that are used in this document.

Table 1-2. Different Typographic Notations

Example	Description
placeholder	Items in italics are placeholders for information that you provide. Italicized text is also used for the titles of publications and for emphasis.
code	Fixed-width type indicates text that must be typed exactly as shown. It is used for instruction mnemonics, symbols, subcommands, parameters, and operators. Fixed-width type is also used for example code.

1.3.3 Special Terms

The following table describes the terms that have special meanings.

Table 1-3. Special Terms

Term	Meaning
`byte`	An 8-bit data object
`double-long`	A 64-bit data object

Table continues on the next page...

Table 1-3. Special Terms (continued)

Term	Meaning
`long`	A 32-bit data object
`word`	A 16-bit data object

Chapter 2
Low-level Binary Interface

The low-level binary interface defines low-level system standards for the StarCore architectures of processor cores, including:

- processor-specific binary interface (the instruction set and representation of fundamental data types) and
- function calling conventions that define:
  - how arguments are passed and results returned,
  - how registers are assigned, and
  - how the calling stack is organized.

This chapter describes:

2.1 StarCore Architectures

To conform to the ABI, the processor must execute the architecture's instructions and produce the expected results.

This ABI does not define requirements for the services provided by an operating system, nor does it specify what instructions must be implemented in hardware. A software emulation of the architecture could conform to the ABI.

Programs that use non-StarCore instructions or capabilities do not conform to the StarCore ABI. Such programs may produce unexpected results when run on machines lacking the non-StarCore capability.

The StarCore SC3900FP architecture is a departure from previous StarCore architectures, namely the SC1200, SC1400, SC2200, SC2400, SC3400, and SC3850. The StarCore SC3900FP is not assembly level compatible with StarCore devices based on SC3850 and earlier processor cores.

2.2 Endian Support

The StarCore SC3900FP architecture currently supports only big-endian implementations.

Note that program binaries that run on a big-endian implementation are not portable to any future little-endian implementation (should one exist). The same applies to the data generated by these programs, as well as to the layout of data used by these programs (such as the layout of data generated by compilation tools).

NOTE

The SC3900FP StarCore compiler only supports big endian and the libraries are only compatible with big endian.

The bytes that form the supported data types are ordered in memory according to the following: the most significant byte (MSB) is located in the lowest address (byte 0).

2.3 Fundamental Data Types

This section lists the data types supported by StarCore architecture and also describes the mapping between the fundamental data types and the C language data types.

The StarCore architecture defines the following data types:

- An 8-bit `byte`
- A 16-bit `word`
- A 32-bit `long word`
- A 64-bit `long long word`

The mapping of these data types does not depend on whether this data is mapped to registers or memory.

The below table describes the mapping between these fundamental data types and the C language data types. Note that fundamental data is always naturally aligned; that is, a `double long word` is 8-byte aligned, a `long word` 4-byte aligned, and a `word` 2-byte aligned.

Table 2-1. Mapping of C Data Types to the StarCore Architecture

Type	C Type	Size (Bits)	Alignment (Bits)	Limits	StarCore
Character	`char, signed char`	8	8	-2^7 through 2^7-1	`Signed byte`
	`unsigned char`	8	8	0 through 2^8-1	`Unsigned byte`
	`short, signed short`	16	16	-2^{15} through $2^{15}-1$	`Signed word`
	`unsigned short`	16	16	0 through $2^{16}-1$	`Unsigned word`
Integral	`int, signed int, enum, long, signed long`	32	32	-2^{31} through $2^{31}-1$	`Signed long word`
	`unsigned int, unsigned long`	32	32	0 through $2^{32}-1$	`Unsigned long word`
	`long long, signed long long`	64	64	-2^{63} through $2^{63}-1$	`Signed double-long word`
	`unsigned long long`	64	64	0 through $2^{64}-1$	`Unsigned double-long word`
Pointer	pointer to data, pointer to function	32	32	0 through $2^{32}-1$	`Unsigned long word`
Floating-point (Floating-point types conform to the IEEE® 754™ format)	`float`	32	32	$-3.402e^{38}$ through $-1.175e^{-38}$ or $1.175e^{-38}$ through $3.402e^{38}$	`Unsigned long word`
	`double, long double`	64	64	$2.225e^{-308}$ through $1.797e^{308}$	`Unsigned double-long word`

NOTE

`_Bool, long long, signed long long, unsigned long long` data types are specified in the ISO C definition (ISO/IEC 9899:1999). Support of the bool data type is optional. If used, this data type must be implemented with the size and alignment shown. Support of `long long`, `signed long long`, `unsigned long` is mandatory. These types must be implemented as described above.

Fractional types are supported in the C language using intrinsic functions. The following table describes the fractional types that are supported.

Table 2-2. Mapping of C Fractional Types to the StarCore Architecture

C Type	C Type Definition	Size (Bits)	Alignment (Bits)	Limits
Fractional	`Word16`	16	16	-1 through $(2^{15}-1)/2^{15}$
Long fractional	`Word32`	32	32	-1 through $(2^{31}-1)/2^{31}$

Long fractional with extension bits	Little-endian:
	`typedef __int40 Word40;`
	Big-endian:
	`typedef struct { unsigned long body; char gap[3]; char ext;} Word40;`
		64	32	-256 through $(2^{39}-1)/2^{31}$

Double precision fractional	`typedef struct {`
	`long msb;`
	`unsigned long lsb;`
	`} Word64;`
		64	64 (A "word64" is 8-byte aligned)	-1 through $(2^{63}-1)/2^{63}$

2.4 Aggregates and Unions

The alignment of aggregates (structures and unions) is the largest alignment of its members. For example, a structure containing a `char`, a `short`, and an `int` must have a 4-byte alignment to match the alignment of the int.

Arrays have the same alignment as their individual elements.

An implementation may provide a user option that overrides the minimum structure alignment. Modules compiled with different minimum alignments cannot interoperate if they interface using structures or unions.

The size of any structure, array, or union must be an integral multiple of its alignment. Structure and unions may require padding to meet size and alignment constraints:

- An entire structure or union is aligned on the same boundary as its most strictly aligned member.

- Each member is allocated starting at the next byte that satisfies the alignment requirement for that member. This may require internal padding.
- 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.

The aggregate members are allocated starting with the low order (lowest addressed) byte of the structure or union, as shown in the following figures.

The below figure displays the internal padding where the first short(s1) starts at a word boundary. Tail padding makes the structure size a multiple of the int member's 4-byte alignment.

NOTE

Alignment and padding may be controlled with `#pragmas` and the `__attribute__` keyword. For more details, refer to *StarCore C-C ++ Compiler User Guide*.

```
struct {   /* 12 bytes, 4-byte aligned */
    char  c;
    short s1;
    int   i;
    short s2;
};
```

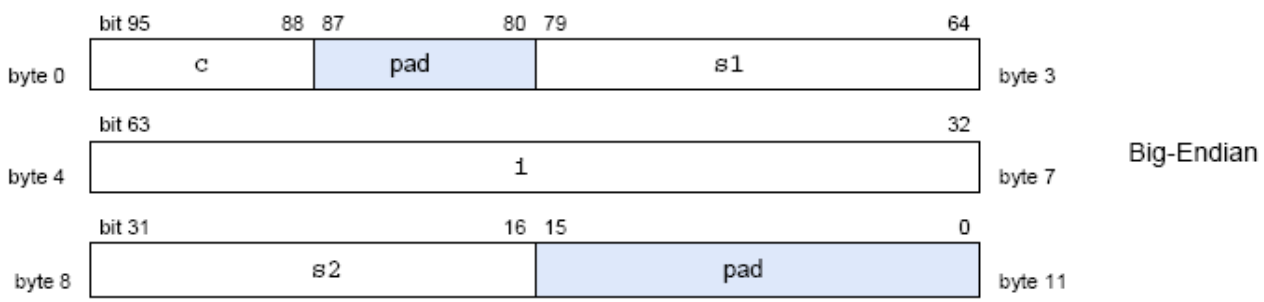

Figure 2-1. Structure with Internal and Tail Padding

The below figure shows the union allocation.

```
union {   /* 4 bytes, 4-byte aligned */
    short s;
    char  c;
    long  l;
};
```

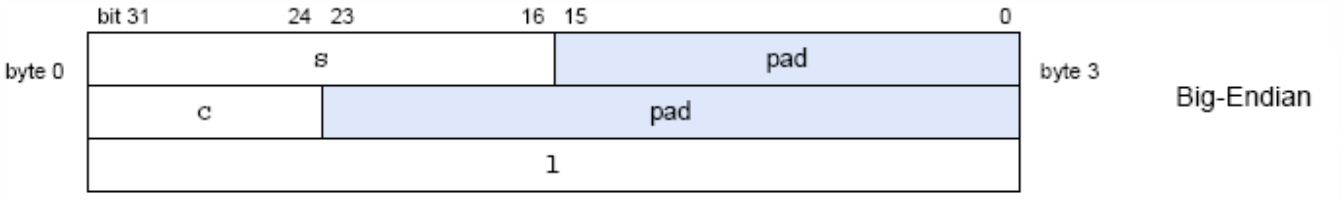

Figure 2-2. Union Allocation

2.5 Bit Fields

The structure and union definitions may have bit fields associated with them.

This section lists the bit fields of the structures and unions in the following table.

Table 2-3. C Bit Field Types

C Type	Maximum width (bits)
`_Bool` `char` (not required for ISO C conformance, but is required for ABI conformance) `signed char` (not required for ISO C conformance, but is required for ABI conformance) `unsigned char` (not required for ISO C conformance, but is required for ABI conformance)	1 to 8

`short`
`signed short`
`unsigned short`
(these bit field types are not required for ISO C conformance, but are required for ABI conformance)
	1 to 16

`int`
`signed int`
`enum` (not required for ISO C conformance, but is required for ABI conformance)
`long` (not required for ISO C conformance, but is required for ABI conformance)
`signed long` (not required for ISO C conformance, but is required for ABI conformance)
	1 to 32

Table 2-3. C Bit Field Types

C Type	Maximum width (bits)
`unsigned int` `unsigned long` (not required for ISO C conformance, but is required for ABI conformance)	

Support of `_Bool` is optional, but all other types described in the above table must be supported. If implemented, `_Bool` must be implemented with the width and range shown. This ABI does not have requirements for `long long` bit fields.

Unsigned bit-field values range from 0 to 2^{w-1}, where `w` is bit field's width in bits. Signed bit-field values range from -2^{w-1} to 2^{w-1-1}.

A *plain* bit field, the one that is not explicitly declared signed or unsigned, is signed. Although they may have type char, short, int, or long (which can have negative values), bit fields of these types have the same range as bit fields of the same size with the corresponding signed type. The same size and alignment rules that apply to other structure and union members also apply to bit fields. The following rules additionally apply to bit fields:

- Bit fields are allocated left to right. The first bit field occupies the MSBs while subsequent bit fields occupy less significant bits.
- A bit field may not cross a boundary for its type. For example, a signed `char` bit field cannot exceed 8 bits in width, and cannot cross a byte 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.
- An unnamed bit field does not affect the alignment of its enclosing structure or union, although an individual bit field's member offsets obey the alignment constraints. An unnamed, zero-width bit field prevents any further member (either bit field or non-bit field) from residing in the storage unit corresponding to the type of the zero-width bit field.

In the following figures, alignments are driven not by the widths of the bit fields but by the underlying types. The following figure shows a structure that is 4-byte aligned and has a 4-byte size because of the int bit fields. There is internal padding so that the `char` bit field does not cross a byte boundary, and so that the short member starts at a word boundary. All members share a `long` word.

```
struct {   /* 4 bytes, 4-byte aligned */
    int   a : 3;
    int   b : 4;
    char  c : 5;
    short d;
};
```

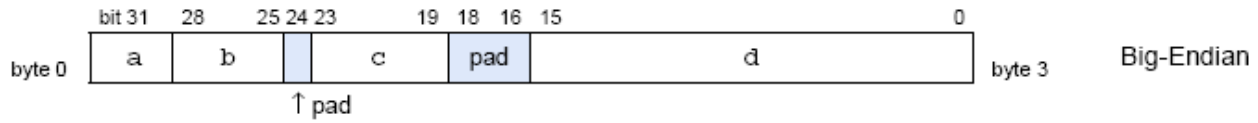

Figure 2-3. Bit Field Alignment and Padding

The below figure shows the structure that is at least 2-byte aligned because the unnamed `long` bit field does not affect structure alignment.

The actual alignment depends on the type alignment-2 in this example-and the minimum structure alignment. Refer to Aggregates and Unions for more information.

The zero-width short bit field pads to the next word boundary.

```
struct {   /* 6 bytes, 2-byte aligned */
    short a : 9;
    short   : 0;
    char  b : 5;
    long    : 15;
};
```

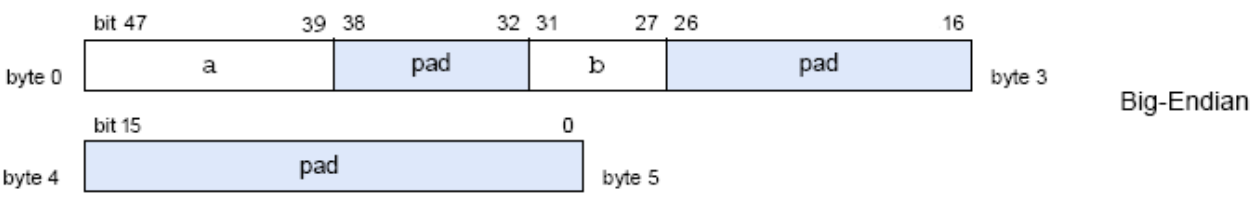

Figure 2-4. Unnamed and Zero-Width Bit Fields

2.6 Function Calling Sequence

This section lists all the calling conventions supported by StarCore DSP Architecture compilers.

Compilers must support these calling conventions:

- Argument Passing
- Return Values
- Processor Mode Bits

2.6.1 Argument Passing

This section lists the calling conventions required for passing arguments.

The following calling conventions must be supported:

- For passing arguments, 16 registers are used: D0-D7 and R0-R7.
- The first 8 parameters (left most) of integral data type or pointer type are passed in R registers.
- Other arguments, from left to right, use D registers up to the point that they are all used (D0-D7), at which time further arguments of these types go on the stack:
  - If a parameter is of type `float`, it is passed in the lowest available D register (up to the first 8 parameters)
  - If a parameter is of type `Word40`, it is passed in the lowest available D register (up to the first 8 parameters of this type)
  - If a parameter is of type long long or double, it is passed in the lowest available pair of D registers (D0D1, D2D3, D4D5, or D6D7), where the first register contains the MSB and second register the LSB part (up to the first 4 parameters of this type)
  - If a parameter is a structure with size lesser than 32 bits, it is passed in the lowest available D register (up to the first 8 of this type)
  - If a parameter is a structure of size 32 to 64 bits, it is passed in pairs of D registers (up to the first 4 of this type)
  - Structures or unions larger than 64-bits are pushed on the stack.
- The remaining parameters are pushed on the stack. Long parameters are pushed on the stack using big-endian mode. Arguments are passed via the stack, in order, from higher addresses to lower addresses.
- When an argument is passed in D register, all the register bytes that are part of the argument are defined by extension to the corresponding type. For example, a first

argument of type `short` is passed in D0[15-0], and the contents of D0[31-16] and D0.e contain the sign of the argument.
- Functions with a variable number of arguments pass all variable arguments on the stack. Such arguments of fewer than 4 bytes are located on the stack as if the argument had been promoted to 32 bits. All fixed arguments are passed according to the preceeding rules.
- All stack arguments are allocated using their alignment constraints.

2.6.2 Return Values

This section provides description about the return values.

- An integral return value, other than a `long long`, is sign or zero extended to 32 bits and returned in R0. A pointer return value is returned in R0.
- A `long long, double`, or `long double` return value is returned in D0 and D1.
- Structures with size between 32 and 64 bits are returned in D0 and D1; D0 containing the most significant long word and D1 containing the least significant long word, regardless of the endian mode. The extension (upper 8 bits of the 40-bit register) of D0 and D1 are undefined.
- The `float` type, `Word40` type, and structures with size lesser than 32 bits are returned in D0.
- Functions returning large structures, meaning structures that do not fit in a single register or double, receive and return the returned structure address in R7. The space for the returned object is allocated by the caller.
- Registers are saved as shown in Listing 2-1 on page 21.

2.6.3 Processor Mode Bits

This section provides description about the operating control bits.

Compilers make the following assumptions about operating control bits:

- Rounding mode (RM) default is 1 (SR[3]=1), which means two's complement rounding.
- Saturation mode bit (SM) default is 0 (SR[2]=[0]), which means no SM-dependent saturation. Note that SC3900FP instructions that saturate do not depend on this bit.
- Scaling mode (SCM) bits default is 0 (SR[4,5] = [00]), which means no scaling.
- Wide 20-bit mode default is 0 and saturation mode 2 defaults to 0 `(SR[W20:SM2]=[00])`.

- Scaling status flag (S) default is 0.
- The hardware loop nesting may occur while entering a function. A function is assumed to use up all hardware loops unless compiler knows otherwise (analysis or calling convention). When compiler knows a function and it's callees do not use all hardware loops, it may generate a hardware loop around the call to that function.

The following listing shows two function calls and the arguments that are allocated for each call.

Listing 2-1. Function Call and Allocation of Parameters

```
 # Function call:
int alpha(int a1, struct fourbytes a2, struct eightbytes a3, int *a4)

# Parameters for the preceding function call:

# a1 - in R0

# a2 - in D0

# a3 - in D2D3

# a4 - R1

# return in R0

# Function call:

beta(long *b1, int b2, int b3[])

# Parameters for the preceding function call:

# b1 - in R0

# b2 - in R1

# b3 - in R2

# Function call:

long long gamma(Word40 c1, long long c2, fourbytes c3, int c4,
eightbytes *c5, int *c6, int c7, short c8, eightbytes c9, Word64 c10,
unsigned c11, int *c12, unsigned long long c13, short c14, int c15)

# c1 - in D0

# c2 - in D2D3

# c3 - in D1

# c4 - in R0

# c5 - in R1

# c6 - in R2

# c7 - in R3

# c8 - in R4

# c9 - in D4D5

# c10 - in D6D7
```

```
# c11 - in R5
# c12 - in R6
# c13 - on stack
# c14 - in R7
# c15 - on stack
# return in D0D1
```

The below table summarizes register usage in the calling convention.

Table 2-4. Register Usage in the Calling Convention

Register	Caller Saved	Callee Saved	Used As
D0-D7	+		Argument passing (`long long`, `double`, `Word40`, structures with size less than 64 bits) - D0 and/or D1 return
D8-D27	+		
D28-D31		+	
D32-D63	+		
R0-R7	+		Argument passing (numeric, pointer) - R0 return
R8-R27	+		
R28-R30		+	
R31		+	Optional frame pointer

2.6.4 Variable Argument Lists

On the SC3900FP and other architectures, C programs intended to be portable use the header files `stdarg.h` or `varargs.h` to deal with variable argument lists.

In some cases, C programs intended to be portable rely on argument passing schemes that assume the following:

- All arguments are passed on the stack
- Arguments appear on the stack in increasing order

In reality, programs that make these assumptions are not portable but still work on many implementations. They do not work with this standard, however, because some arguments are passed in registers.

ANSI C requires that before a function with a variable argument list is called, it must be declared with a prototype containing a trailing ellipsis (...).

2.6.5 Stack

The SP register serves as the stack pointer. SP points to the first available location, with the stack direction being towards higher addresses (that is, a push is implemented as "(sp) +").

Initially, a long word with value -1 is pushed at offset 0 on the stack to serve as a top-of-stack marker. The stack pointer must be 8-byte aligned.

2.6.6 Stack Frame Layout

This section provides details on the stack fram 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.

The stack pointer value must always be a multiple of eight. The below figure shows a typical stack frame for a function and indicates the relative position of local variables, arguments, and return addresses. The stack grows upward from low addresses. 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 within 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.

A "return address" value of FFFFFFFFh (-1) is used to denote the current frame as the outermost (oldest) frame on the current call stack. This convention requires that the outermost frame be manually constructed and that sufficient object file details are available to determine the sizes of all frames on the current call stack. The sole purpose of this convention is to stop stack unwinding while debugging.

Beyond these requirements, a function is free to manage its stack frame in any way desired.

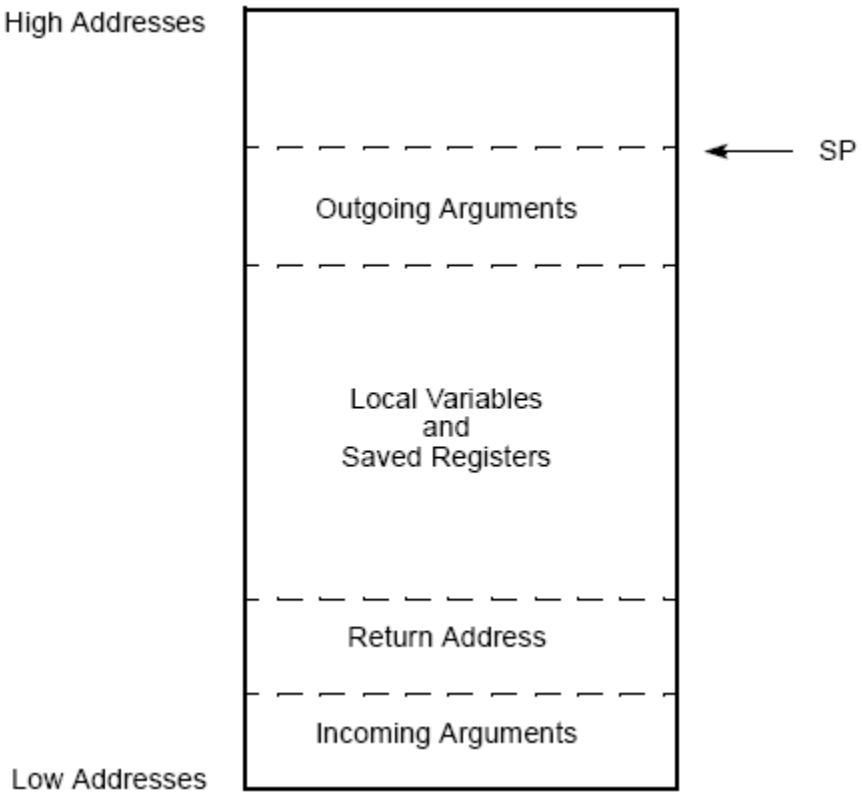

Figure 2-5. Stack Frame Layout

2.6.7 Stack Unwinding

The compiler creates special symbols when a module is compiled without debug enabled - for example, when the `-g` compiler option is not used.

These symbols appear as local symbols in the `.symtab ELF` section and have the following syntax:

```
TextStart_<module_name> : module's low PC
TextEnd_<module_name> : module's high PC
StackOffset_<label> : size of stack at label
FuncEnd_<function_name> : function's high PC
```

where:

- <module_name> is the base name of the source file. The base name must follow the same conventions as assembly language labels. These conventions are documented in Symbol Names.
- <label> is a program label within the function. The value of StackOffset_label is the size of the stack frame at the label. The size is in 2-byte words and does not include an implied JSR/BSR two-word stack push.
- <function_name> is the function name, without a leading underscore.

For example, a hello.c program might generate the ELF symbol sequence as shown below.

Listing 2-2. ELF Symbol Sequence

```
Value Size Binding Type Section Name
----- ---- ------- ---- ------- ----

10120h 0 LOCAL NOTYPE .text TextStart_hello

0h 0 LOCAL NOTYPE ABS StackOffset__main

2h 0 LOCAL NOTYPE ABS StackOffset_DW_2

0h 0 LOCAL NOTYPE ABS StackOffset_DW_5

1012Ah 0 LOCAL NOTYPE .text DW_2

10136h 0 LOCAL NOTYPE .text DW_5

10138h 0 LOCAL NOTYPE .text FuncEnd_main

10138h 0 LOCAL NOTYPE .text TextEnd_hello
```

In the above figure the binding LOCAL means an ELF symbol binding of STB_LOCAL, the type NOTYPE means a symbol type of STT_NOTYPE, and the section ABS means a symbol table entry of SHN_ABS.

The listing below illustrates how these symbols might be defined in an assembly-language program.

Listing 2-3. Generating Stack Unwinding Symbols in Assembly Code

```
section .text local
TextStart_hello

;*********************************************************

; Example function _main

;*********************************************************

global _main

_main type func

[

 push.4x d27:d28:d29:d31

 push.4l r27:r28:r30:r31
```

```
]
DW_2
...
[
 pop.4l r27:r28:r30:r31
 pop.4x d27:d28:d29:d31
]
DW_5
rts
FuncEnd__main
StackOffset__main equ 0 ; At _main sp = 0 words
StackOffset_DW_2 equ 96 ; At DW_2 sp = 96 words
StackOffset_DW_5 equ 0 ; At DW_5 sp = 0 words
TextEnd_hello
endsec
```

2.6.8 Register Saving and Restoring Functions

This section explains the register saving and restoring functions.

The register saving and restoring functions described in this section save and restore the callee-saved registers defined by Table 2-4. These functions are provided to save and restore these registers with a minimal increase in static code size. The functions use nonstandard calling conventions that require them to be statically linked into any executable or shared object modules in which they are used.

Thus their interfaces are private, within module interfaces, and therefore are not part of the ABI. They are defined here only to encourage uniformity among compilers in the code used to save and restore registers.

After calling the saving function `___CW_SC100_callee_save`, the stack frame values relative to the address in the stack pointer (SP) as shown in the below figure:

Figure 2-6. Stack Pointer

The restoring function `___CW_SC100_callee_restore` assumes the stack frame layout above. It restores the callee-saved registers and returns through the caller return address stored at SP - 24. There is no need for an RTS after calling the restoring function, since it returns automatically for the caller. A compiler implementation may also use different names for the saving and restoring, namely `___CW_SC100_callee_save` and `___CW_SC100_callee_restore`.

The below listing shows an example use of the saving and restoring functions. The functions do not modify any caller-saved registers.

Listing 2-4. Saving and Restoring Functions Usage Example

```
_foo:
   bsr ___Qabi_callee_save ; Save callee-saved registers.

  adda #frame_size_foo,sp ; Adjust SP by frame size.

_foo_body:

  ...

_foo_body_end:

  suba #frame_size_foo,sp ; Adjust SP by frame size.

  bra ___Qabi_callee_restore ; Restore callee-saved registers
                             ; and return to caller of foo.
```

2.6.9 Layout of setjmp and longjmp

The layout preserves the callee-saved registers, which is needed to restore the state when `longjmp` is called.

The layout for the `jmp_buf` used by `setjmp` and `longjmp` is shown in the below figure.

typedef int jnp_buf[7];

Offset	Saved Register
+0	D28
+8	D29
+16	D30
+24	D31
+32	R28
+36	R29
+40	R30
+44	R31
+48	SP
+52	PC (Return Address)

Figure 2-7. Layout for jmp_buf

2.6.10 Frame and Global Pointers

This ABI standard does not require the use of a frame pointer or a global pointer. If, however, the use of a frame pointer is necessary, a compiler may allocate R31 as a frame pointer.

When these registers are allocated for this purpose, they should be saved and restored as part of the function prologue/epilogue code.

2.6.11 Dynamic Memory Allocation

Dynamic allocations are implemented using a heap structure managed by the standard library functions `malloc()` and `free()`. The heap shall be allocated statically by the linker. All addresses returned by `malloc()` shall be at least 8-byte aligned.

2.6.12 Hardware Loops

All hardware loop resources are available for the compiler's use. As it is assumed that no nesting occurs when entering a function, a function may use all four nesting levels for its own use.

2.7 Address Modifier Modes

This section list the assumptions that compilers make about the address modifier modes.

Following are the assumptions:

- The default C runtime state of the MCTL register is 0, which identifies the memory address calculation methods for R0-R7 as linear.
- If the MCTL register is changed local to a function, then MCTL must be restored to 0 prior to calling any other function or returning from the original function.

2.8 Compatibility with SC3850

The SC3850-based intrinsics are seamlessly mapped to SC3900FP equivalents via the `prototype.h` header, which is by-default included in the SC3900FP-based tools.

In addition, you can change the following mode bits for the SC3850-based code.

- Rounding mode (RM)
- Saturation mode bit (SM)
- Scaling mode (SCM) bits
- Wide 20-bit mode (W20)
- Saturation mode 2 (SM2)

2.9 Data Addressing Models

The SC3900FP-based compiler uses only one memory model, which was referred to as *huge memory model* in previous architectures.

Chapter 3
High-level Language Issues

This chapter explains:

3.1 C Preprocessor Predefines

All C/C++ language compilers must have the predefined macros.

These predefined macros as described in the below table, in addition to the predefined macros required by the C and C++ language standards.

Table 3-1. Predefined Macros

Macro	Description
_SC3900_	For use with all compilers based on the SC3900 architecture.
_SC3900FP_	For use with all compilers based on the SC3900FP architecture.
BIG_ENDIAN	Defined by default since big-endian mode is the only mode.

3.2 C Name Mapping

Externally visible names in the C language are prefixed by an underscore (_) when generating assembly language symbol names.

For example, the below list fragment

```
void testfunc() {
  return;
}
```

generates assembly code similar to the following fragment:

```
_testfunc:

rts
```

3.3 C++ Name Mapping (Name Mangling)

C++ names are formed by prefixing an underscore to the C++ mangled name.

3.4 C System Calls

There are several typedefs specified in POSIX.1 that are required for system call wrappers.

These types are described in the below listing for the StarCore architecture.

Listing 3-1. Typedefs Required for System Call Wrappers

```
typedef unsigned int mode_t;
typedef long int off_t;

typedef unsigned int size_t;

typedef int ssize_t;

typedef unsigned long clock_t;

typedef unsigned long time_t;
```

The below listing shows system calls that must also be supported.

Listing 3-2. System Calls Required for System Call Wrappers

```
int open(const char *, int, ...); // Third arg is mode_t if present.
int close(int);

ssize_t read(int, void *, size_t);

ssize_t write(int, const void *, size_t);
```

```
off_t lseek(int, off_t, int);

int unlink(const char *);

int rename(const char *, const char *);

int access(const char *, int);

clock_t clock(void);

time_t time(time_t *);
```

3.5 Compiler Assist Libraries

The StarCore architecture does not provide hardware support for floating-point data types, nor for divide functionality for integer types. Compilers should provide the functionality for some of these operations through the use of support library routines.

Compilers that generate in-line code to provide these functions must make no reference to the library functions. Compilers that provide these functions by generating function calls to the support libraries must use the calling convention when calling them. To ensure the ability to link code produced by different compilers into a single executable, it is required that names of compiler support library functions match those listed in the tables that follow.

Routines in support libraries must satisfy the following constraints:

- Identical results must be returned when a routine is re-invoked with the same input arguments.
- Multiple calls with the same input arguments can be collapsed into a single call with a cached result.

These properties permit a compiler to make assumptions about variable lifetimes across library function calls: values in memory will not change, and previously de-referenced pointers need not be referenced again.

The functions to be provided through support library routines include the following:

- Floating-Point Routines
- Integer and Fractional Arithmetic Routines
- Optional Integer Routines

3.5.1 Floating-Point Routines

Conformant library support must include the floating-point routines listed in this section (the routine interfaces are shown as C function prototypes).

These floating point routines must comply with the calling conventions described in Function Calling Sequence.

The data formats are as specified in IEEE®-754™. The math routines are not required to compute results as specified in IEEE®-754™. Implementation of these routines must document the degree to which operations conform to the IEEE® standard. Not all users of floating point require IEEE®-754™ precision and exception handling, and may not want to incur the overhead that complete conformance requires.

Table 3-2. Summary of Floating-Point Routines

32-bit Floating point routines	___QFAbs	___QFDiv
	___QFCmpeq	___QFCmpge
	___QFCmpgt	___QFCmple
	___QFCmplt	___QFSub
	___QFMul	___QFCmpne
	___QFAdd	___QInt16sToFloat
	___QFloatToInt16s	___QInt16uToFloat
	___QFloatToInt16u	___QInt32sToFloat
	___QFloatToInt32s	___QInt32uToFloat
	___QFloatToInt32u	__d_utof
	__d_ftou	__d_itof
	__d_ftoi	__d_lltof
	__d_ftoll	__d_ulltod
	__d_ftoull	__d_dtof
	__d_ftod	
64-bit Floating point routines	__d_abs	__d_div
	__d_feq	__d_fge
	__d_fgt	__d_fle
	__d_flt	__d_sub
	__d_mul	__d_fne
	__d_add	__d_usub
	__d_itod	__d_dtoi
	__d_utod	__d_dtou
	__d_lltod	__d_dtoll
	__d_ulltod	__d_dtoull

NOTE

The routines, `_d_ftoi`, `_d_itof`, `_d_ftou`, and `_d_utof` exist under two aliased names. See the below table for further description. The routines, `_d_feq`, `_d_fge`, `_d_fgt`, `_d_fle`, `_d_flt`, `_d_fne`,

_f_feq, _f_fge, _f_fgt, _f_fle, _f_flt, and _f_fne use a modified calling convention (defined in Argument Passing) for the return value. This value is returned in the True bit of SR instead of D0.

Table 3-3. Description of Float-Point Routines

Type	Routine	Description
32-bitFloating point routines	___QFAbs	Returns absolute value of float/ double
	___QFMul	Computes a * b
	___QFAdd	Computes a + b
	___QFloatToInt16s	Converts float to int16
	___QFloatToInt16u	Converts float to unsigned int16
	___QFloatToInt32s	Converts float to int32
	___QFloatToInt32u	Converts float to unsigned int32
	__d_ftou	Converts float to unsigned int32
	__d_ftoi	Converts float to int32
	__d_ftoll	Converts float to long long
	__d_ftoull	Converts float to unsigned long long
	__d_ftod	Converts float to double
	___QFDiv	Computes a/b
	___QFSub	Computes a-b
	___QInt16sToFloat	Converts int16 to float
	___QInt16uToFloat	Converts unsigned int16 to float
	___QInt32sToFloat	Converts int32 to float
	___QInt32uToFloat	Converts unsigned int32 to float
	__d_utof	Converts unsigned int32 to float
	__d_itof	Converts int32 to float
	__d_lltof	Converts long long to float
	__d_ulltod	Converts unsigned long long to float
	__d_dtof	Converts double to float
64-bitFloating point routines	__d_abs	Returns the absolute value
	__d_mul	Computes a * b
	__d_add	Computes a + b
	__d_itod	Converts int32 to double
	__d_utod	Converts unsigned int32 to double
	__d_div	Computes a/b
	__d_sub	Computes a-b
	__d_usub	Returns unary minus
	__d_dtoi	Converts double to int32

Table continues on the next page...

Table 3-3. Description of Float-Point Routines (continued)

Type	Routine	Description
	__d_dtou	Converts double to unsigned int32
	__d_lltod	Converts long long to double
	__d_ulltod	Converts unsigned long long to double
	__d_dtoll	Converts double to long long
	__d_dtoull	Converts double to unsigned long long

The following routines shown in the table below use a modified calling convention concerning the return value. This value is returned in the *true* bit of register SR instead of D0. These routines are supposed to assist the compiler and cannot be directly called from the C language.

Table 3-4. Routines with Modified Calling Conventions

Type	Routine	Description
32-bit Floating point routines	___QFCmpeq	Performs equality test between the single-precision values of a and b. Returns 1 if true and a 0 otherwise.
	___QFCmpge	Performs >= test between the single-precision values of a and b. Returns 1 if true and a 0 otherwise.
	___QFCmpgt	Performs > test between the single-precision values of a and b. Returns 1 if true and a 0 otherwise.
	___QFCmple	Performs <= test between the single-precision values of a and b. Returns 1 if true and a 0 otherwise.
	___QFCmplt	Performs < test between the single-precision values of a and b. Returns 1 if true and a 0 otherwise.
	___QFCmpne	Performs inequality test between the single-precision values of a and b. Returns 1 if true and a 0 otherwise.
64-bit Floating point routines	__d_feq	Performs equality test between the double-precision values of a and b. Returns 1 if true and a 0 otherwise.
	__d_fge	Performs >= test between the double-precision values of a and b. Returns 1 if true and a 0 otherwise.
	__d_fgt	Performs > test between the double-precision values of a and b. Returns 1 if true and a 0 otherwise.
	__d_fle	Performs <= test between the double-precision values of a and b. Returns 1 if true and a 0 otherwise.

Table continues on the next page...

Table 3-4. Routines with Modified Calling Conventions (continued)

Type	Routine	Description
	`__d_flt`	Performs < test between the double-precision values of `a` and `b`. Returns 1 if true and a 0 otherwise.
	`__d_fne`	Performs inequality test between the double-precision values of `a` and `b`. Returns 1 if true and a 0 otherwise.

3.5.2 Integer and Fractional Arithmetic Routines

Conformant library support must include the integer and fractional arithmetic routines as listed in this section (the routine interfaces are shown as C function prototypes).

These integer routines must comply with the calling conventions described in Function Calling Sequence. These routines have no side effects.

Table 3-5. Integer Routines and Fractional Routines

__div16	__div32	__rem16	__rem32	_div_s
__udiv16	__udiv32	__urem16	__urem32	

The following table lists the description of integer and fractional arithmetic routines.

Table 3-6. Descriptions of Integer and Fractional Arithmetic Routines

Routine	Description
`int __div16(short a, short b)`	Returns the value of a/b. If the divisor has the value zero, the behavior is undefined.
`int __udiv16(unsigned short a, unsigned short b)`	Returns the unsigned value of a/b. If the divisor has the value zero, the behavior is undefined.
`int __div32(long a, long b)`	Returns the value of a/b. If the divisor has the value zero, the behavior is undefined.
`int __udiv32(unsigned long a, unsigned long b)`	Returns the unsigned value of a/b. If the divisor has the value zero, the behavior is undefined.
`int __rem16(short a, short b)`	Returns the remainder upon dividing a by b. If the divisor has the value zero, the behavior is undefined.
`int __urem16(unsigned short a, unsigned short b);`	Returns the unsigned remainder upon dividing a by b. If the divisor has the value zero, the behavior is undefined.
`int __rem32(long a, long b)`	Returns the remainder upon dividing a by b. If the divisor has the value zero, the behavior is undefined.
`int __urem32(unsigned long a, unsigned long b)`	Returns the unsigned remainder upon dividing a by b. If the divisor has the value zero, the behavior is undefined.

Table continues on the next page...

Table 3-6. Descriptions of Integer and Fractional Arithmetic Routines (continued)

Routine	Description
`short _div_s(short a, short b)`	Returns the value of the fractional divide a/b. If the divisor has the value zero, the behavior is undefined.

3.5.3 Optional Integer Routines

If the optional C long long data type is supported, library support must also include the long long integer routines as described in this section.

These routines must comply with the calling conventions described in Function Calling Sequence.

Table 3-7. Summary of Optional Integer Routines

_SDiv64	_d_dtoll	_d_ftoll
_UDiv64	_d_dtoull	_d_ftoull
_SRem64	_d_lltod	_d_lltof
_URem64	_d_ulltod	_d_ulltof

The following table lists the description of optional integer routines.

Table 3-8. Descriptions of Optional Integer Routines

Routine	Description
`long long _SDiv64(long long a, long long b)`	Computes the quotient a/b, truncating any fractional part, and returns the signed long long result. If the divisor has the value zero, the behavior is undefined.
`unsigned long long _UDiv64(unsigned long long a, unsigned long long b)`	Computes the quotient a/b, truncating any fractional part, and returns the unsigned long long result. If the divisor has the value zero, the behavior is undefined.
`long long _SRem64(long long a, long long b)`	Computes the remainder upon dividing a by b, and returns the signed long long result. If the divisor has the value zero, the behavior is undefined.
`unsigned long long _URem64(unsigned long long a, unsigned long long b)`	Computes the remainder upon dividing a by b, and returns the unsigned long long result. If the divisor has the value zero, the behavior is undefined.
`long long _d_dtoll(double a)`	Converts the double-precision value of a to a signed long long by truncating any fractional part, and returns the signed long long value.
`unsigned long long _d_dtoull(double a)`	Converts the double-precision value of a to an unsigned long long by truncating any fractional part, and returns the unsigned long long value.

Table continues on the next page...

Table 3-8. Descriptions of Optional Integer Routines (continued)

Routine	Description
`double _d_lltod(long long a)`	Converts the signed long long value of `a` to a double-precision value, and returns the double-precision value.
`double _d_ulltod(unsigned long long a)`	Converts the unsigned `long long` value of `a` to a double-precision value, and returns the double-precision value.
`long long _d_ftoll(float a)`	Converts the single-precision value of `a` to a signed `long long` by truncating any fractional part, and returns the `signed long long` value.
`unsigned long long _d_ftoull(float a)`	Converts the single-precision value of `a` to an `unsigned long long` by truncating any fractional part, and returns the `unsigned long long` value.
`float _d_lltof(long long a)`	Converts the `signed long long` value of `a` to a single-precision value, and returns the single-precision value.
`float _d_ulltof(unsigned long long a)`	Converts the `unsigned long long` value of `a` to a single-precision value, and returns the single-precision value.

3.6 Access to Architectural Features

The set of intrinsics listed in the StarCore C-C++ Compiler Intrinsics Reference Manual must be supported. These intrinsics allow access to hardware resources from a C-based application without using assembly inserts. The intrinsics from SC3850 and earlier processors must also be supported.

Chapter 4
Object File Format

The executable and linking format (ELF) is used for representing the binary application to the system. For a complete description of ELF, refer to the Tools Interface Standards (TIS) ELF Specification, Version 1.1.

In addition, the chapter highlights the differences between the ELF version 1.1 definition and the StarCore implementation, and elaborates the interface for relocatable and executable programs. A relocatable program contains code suitable for linking to create another relocatable program or executable program. An executable program contains binary information suitable for loading and execution on a target processor.

This chapter explains:

4.1 Interface Descriptions

This section explains the interface description in detail.

ELF presents two views of binary data, as shown in the figure below.

- The linking view provides data in a format suitable for incremental linking into a relocatable file or final linking to an executable file.
- The execution view provides binary data in a format suitable for loading and execution.

An ELF header is always present in either view of the ELF file. For the linking view, sections are the main entity in which information is presented. A section header table provides information for interpretation and navigation for each section. For 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. For exact details, see the ELF version 1.1 specification.

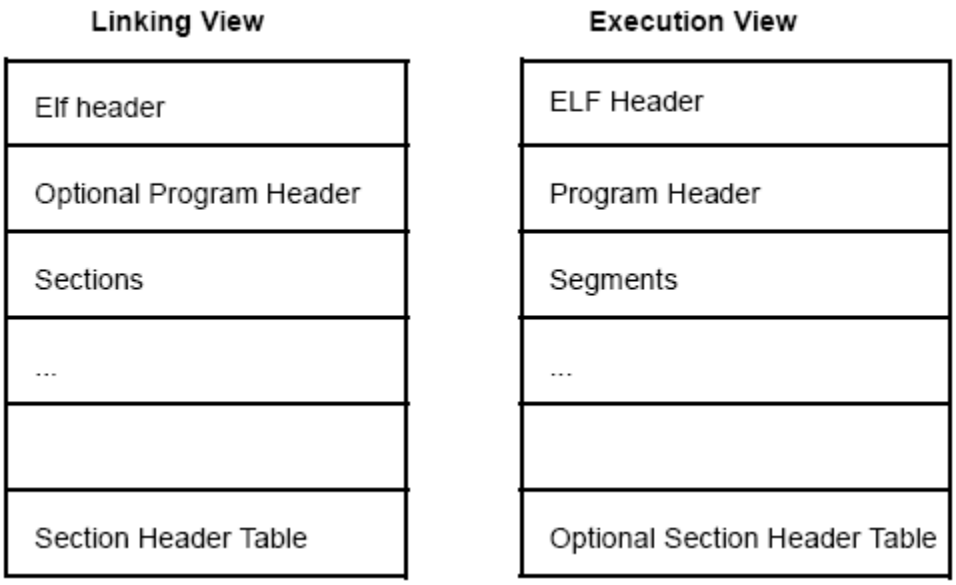

Figure 4-1. Object File Format

4.2 ELF Header

The listing below shows the ELF header structure.

The structure and its fields are defined by the ELF version 1.1 specification.

Listing 4-1. ELF Header Structure

```
typedef struct {
  unsigned char e_ident[EI_NIDENT];

  Elf64_Half    e_type;

  Elf64_Half    e_machine;

  Elf64_Word    e_version;

  Elf64_Addr    e_entry;

  Elf64_Off     e_phoff;

  Elf64_Off     e_shoff;
```

```
  Elf64_Word    e_flags;
  Elf64_Half    e_ehsize;
  Elf64_Half    e_phentsize;
  Elf64_Half    e_phnum;
  Elf64_Half    e_shentsize;
  Elf64_Half    e_shnum;
  Elf64_Half    e_shstrndx;
} Elf64_Ehdr;
```

The below listing shows an example of the StarCore-specific code.

Listing 4-2. StarCore Specifics

```
e_ident[EI_CLASS] = ELFCLASS64
e_ident[EI_DATA] = ELFDATA2LSB (little-endian memory mode)

e_ident[EI_DATA] = ELFDATA2MSB (big-endian memory mode)

e_machine: 0x3a (EM_STARCORE)
```

The `e_flags` field is used to distinguish object files translated for different cores, features, different core revisions, and different ABI versions. The `e_flags` field is split into three parts:

- Bits 0-5: The core features. The defined core features are: `#define EF_STARCORE_CORE_4_MAC 0`

  Mixing object files with `EF_STARCORE_CORE_2_MAC` and `EF_STARCORE_CORE_4_MAC` result in an object file with `EF_STARCORE_CORE_4_MAC`.

- Bits 6-11: The revision of the used core architecture. The defined core revisions are: `#define EF_STARCORE_CORE_REV_UNKNOWN 0 #define EF_STARCORE_CORE_REV_SC140E_V3 3 #define EF_STARCORE_CORE_REV_SC3000_V5 5 #define EF_STARCORE_CORE_REV_SC3000_V6D 7 #define EF_STARCORE_CORE_REV_SC3900_V7 8`
- Bits 12-17: The ABI version. The defined ABI versions are: `#define EF_STARCORE_ABI_PREABI 0 #define EF_STARCORE_ABI_NONCONFORMING 1 #define EF_STARCORE_ABI_2_0 2 #define EF_STARCORE_ABI_3_0 3`

  Linking with different ABI versions or with non-conforming object files may result in linker errors or undetermined output.

- Bits 18-31: Zero. Reserved for future use.

The following list shows the definition of macros for accessing the `e_flag` parts.

Listing 4-3. Definition of Macros for Accessing e_flag Parts

```
#define ELF64_EF_STARCORE_CORE(e_flags) ((e_flags) & 0x3f)
#define ELF64_EF_STARCORE_REV(e_flags) (((e_flags) >> 6) & 0x3f))
```

```
#define ELF64_EF_STARCORE_ABI(e_flags) (((e_flags) >> 12) & 0x3f))

#define ELF64_EF_STARCORE(core,rev,abi) \

      (((core) & 0x3f) | (((rev) & 0x3f) << 6) | (((abi) & 0x3f) << 12))
```

4.3 Sections

Sections are the main components of the ELF file. Section headers define all the information about a section.

A section header is shown in the listing below. It is identical to the ELF version 1.1 definition.

Listing 4-4. Section Header Structure

```
typedef struct {
    Elf64_Word    sh_name;

    Elf64_Word    sh_type;

    Elf64_Xword   sh_flags;

    Elf64_Addr    sh_addr;

    Elf64_Off     sh_offset;

    Elf64_Xword   sh_size;

    Elf64_Word    sh_link;

    Elf64_Word    sh_info;

    Elf64_Xword   sh_addralign;

    Elf64_Xword   sh_entsize;

} Elf64_Shdr;
```

Sections used in StarCore ELF binaries are listed in the below table. The section names listed in this table are case sensitive and are reserved for the system.

Table 4-1. StarCore ELF Sections

Name (sh_name)	Type (sh_type)	Flags (sh_flags)	Purpose
.text	SHT_PROGBITS	SHF_ALLOC, SHF_EXECINSTR	Executable instructions
.data	SHT_PROGBITS	SHF_ALLOC, SHF_WRITE	Initialized data
.rodata	SHT_PROGBITS	SHF_ALLOC	Read-only, initialized data
.zdata	SHT_PROGBITS	SHF_ALLOC, SHF_WRITE	Zero Data Area initialized data

Table continues on the next page...

Table 4-1. StarCore ELF Sections (continued)

Name (sh_name)	Type (sh_type)	Flags (sh_flags)	Purpose
.bss	SHT_NOBITS	SHF_ALLOC, SHF_WRITE	Uninitialized data (The contents of the .bss and .zbss sections are zeroed when loaded)
.zbss	SHT_NOBITS	SHF_ALLOC, SHF_WRITE	Zero Data Area uninitialized data (The contents of the .bss and .zbss sections are zeroed when loaded)
.relasection	SHT_RELA	None	Relocation info for section (see Relocation)
.symtab	SHT_SYMTAB	None	Symbol table
.shstrtab	SHT_STRTAB	None	Section name string table
.strtab	SHT_STRTAB	None	General purpose string table
.note	SHT_NOTE	None	File identification (see Note Section)
.debug_abbrev	SHT_PROGBITS	None	Abbreviation tables (This information is in DWARF2 format)
.debug_aranges	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)
.SC100.delay_slots	SHT_PROGBITS	None	Static delay slot information

4.4 Relocation

Each section that contains relocatable data has a corresponding relocation section of type SHT_RELA. This section also provide detail description on the Relocation Types and Relocation Stack.

The sh_info field of the relocation section defines the section header index of the section (henceforth referred to 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, .rela is prefixed to the name of the data section.

A relocation entry is defined by the Elf32_Rela structure and associated macros as shown in the below listing. 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.

Listing 4-5. Relocation Entry Defined with Elf64_Rela

```
typedef struct {
  Elf64_Addr    r_offset;

  Elf64_Xword   r_info;

  Elf64_Sxword  r_addend;

} Elf64_Rela;

#define ELF64_R_SYM(i)      ((i)>>32)

#define ELF64_R_TYPE(i)     ((i)&UINT64_C(0xffffffff))

#define ELF64_R_INFO(s,t)   (((s)<<32)|((t)&UINT64_C(0xffffffff)))
```

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.

In the remainder of this section, 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). For a relocation type in Relocation Types, 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, in which case the relocation value is simply the value of the r_addend field. This degenerate case is also often used by the extended relocation types defined in Relocation Stack, particularly R_STARCORE_OPER and R_STARCORE_POP, for which a symbol value is rarely useful.

Further in this topic:

- Relocation Types
- Relocation Stack
- Instruction Address Versus VLES Address

4.4.1 Relocation Types

This section provide details on the relocation type definitions.

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 below table lists and defines the relocation types. The following fields appear in each definition:

- Type-The value extracted using ELF64_R_TYPE, both as a number and as a standard C preprocessor symbol. A brief abstract of the relocation follows in parentheses.
- Size-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.
- Signedness-Whether the relocation value is treated as signed or unsigned.
- Alignment-Alignment requirement in bits of the relocation value. This is the number of LSBs in the relocation value that must be zero.
- Shift-The number of bits the relocation value is right-shifted before it is encoded. The shift count subtracted from the size is the number of bits used to encode the relocation value.
- Overflow-Whether the relocation type checks the relocation value for overflow. A calculated relocation value can be larger than the intended field, and the relocation type can verify the value range or truncate the result. For verifying the value range, a relocation value must be in range or an error message is issued. Truncation simply cuts off the exceeding bits and fits the result into the intended field, while issuing a warning message.
- Special-Any other special processing performed during relocation. For instructions that compute the PC, the value of the PC is the address of the instruction to which the relocation applies (computed by adding the relocation's `r_offset` field and the data section's `sh_addr` field), not the machine's runtime PC value (refer to Instruction Address Versus VLES Address for more information).
- Encoding-The way the relocation value is encoded in the target memory locations. The order of the bits in the instruction operand or data value encoding does not necessarily match the order of the bits in the relocation value. Upper case letters indicate relocation value bits that are more significant than bits indicated by lower case letters. The `s` and `S` denote bits of a signed relocation value, `u` and `U` denote bits of an unsigned relocation value, and `x` denotes a bit of a relocation value that is either signed or unsigned. Dashes indicate bits not changed by the relocation. All encodings

for instructions are shown in groups of 16 bits; the bits within each group are subject to byte-swapping depending on target endianness. Relocation types 2 and 3 (16-bit and 32-bit direct) are also endianness-sensitive.

- Applies To-Which instructions or directives generate this relocation. For example:
- `----S---sss--SSS ---sssssssssssS`

```
    1   111  111   1198765432101

    9   432  876  10  5
```

Left to right, the `Ss` represent bits 19-15 and the `ss` represent bits 14-0 of the relocation value.

```
byte 0 byte 1 byte 2 byte 3

----S--- sss--SSS ---sssss ssssssssS

   1    111  111    11987 65432101

   9    432  876    10    5
```

Table 4-2. Relocation Type Definitions

Type	1, `R_STARCORE_DIRECT_8` (8-bit direct)
Size	8
Signedness	Either
Alignment	0
Shift	0
Overflow	Truncate
Encoding	`xxxxxxxx`
Applies to	`76543210` DCB
Type	2, `R_STARCORE_DIRECT_16` (16-bit direct)
Size	16
Signedness	Either
Alignment	0
Shift	0
Overflow	Truncate
Encoding	`xxxxxxxxxxxxxxxx`

Applies to	`1111119876543210`
	`543210`
	DCW

Type	3, `R_STARCORE_DIRECT_32` (32-bit direct)

Table continues on the next page...

Table 4-2. Relocation Type Definitions (continued)

Size	32
Signedness	Either
Alignment	0
Shift	0
Overflow	Truncate
Encoding	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Applies to	33222222222211111111119876543210 109876543210987654321 0 DCL
Type	50, R_STARCORE_DIRECT_64 (64-bit direct)
Size	64
Signedness	Either
Alignment	0
Shift	0
Overflow	Truncate
Encoding	xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
Applies to	66665555555555444444444433333333332222222222111111111987 6543210 32109876543210987654321098765432109876543210987654321 0 DCLL
Type	80, R_SC3900_u4_0_0
Size	4
Signedness	Unsigned
Alignment	0
Shift	0
Overflow	Verify
Encoding	-------uuuu----- ----------------
Applies to	. 3210 ash.lft.4w #u4,Dab,Dmn; ash.lft.s.4w #u4,Dab,Dmn; ash.rgt.4w #u4,Dab,Dmn; lsh.rgt.4w #u4,Dab,Dmn
Type	81, R_SC3900_u5_0_0
Size	5
Signedness	Unsigned
Alignment	0
Shift	0
Overflow	Verify
Encoding	------uuuuu----- ----------------

Table continues on the next page...

Table 4-2. Relocation Type Definitions (continued)

Applies to	43210 add.leg.x #u5,Da,Dn; add.x #u5,Da,Dn; adda #u5,Ra,Rn; adda.lin #u5,Ra,Rn; addc.wo.l #u5,Da,Dn; addc.wo.leg.x #u5,Da,Dn; addc.wo.x #u5,Da,Dn; ash.lft.2l #u5,Dab,Dmn; ash.lft.2t #u5,Da,Dn; ash.lft.2w #u5,Da,Dn; ash.lft.4t #u5,Dab,Dmn; ash.lft.l #u5,Da,Dn; ash.lft.leg.x #u5,Da,Dn; ash.lft.s.2l #u5,Dab,Dmn; ash.lft.s.2t #u5,Da,Dn; ash.lft.s.2w #u5,Da,Dn; ash.lft.s.4t #u5,Dab,Dmn; ash.lft.s.l #u5,Da,Dn; ash.lft.s20.2t #u5,Da,Dn; ash.lft.s20.4t #u5,Dab,Dmn; ash.rgt.2l #u5,Dab,Dmn; ash.rgt.2t #u5,Da,Dn; ash.rgt.2w #u5,Da,Dn; ash.rgt.4t #u5,Dab,Dmn; ash.rgt.l #u5,Da,Dn; ash.rgt.leg.x #u5,Da,Dn; ash.rgt.s.2t #u5,Da,Dn; ash.rgt.s.4t #u5,Dab,Dmn; asha.lft #u5,Ra,Rn; asha.rgt #u5,Ra,Rn; clip.t.u.2b #u5,Da,Dn; clip.w.u.2b #u5,Da,Dn; clip.x.u.b #u5,Da,Dn; cmp.gt.u.l #u5,Da,Pmn; cmp.gt.u.l #u5,Da,Pn; cmp.le.u.l #u5,Da,Pmn; cmp.le.u.l #u5,Da,Pn; extc.lft #u5,Da; extc.rgt #u5,Da; extracta #U5,#u5,Ra,Rn; extracta.u #U5,#u5,Ra,Rn; lsh.rgt.2l #u5,Dab,Dmn; lsh.rgt.2w #u5,Da,Dn; lsh.rgt.l #u5,Da,Dn; lsha.rgt #u5,Ra,Rn; sub.leg.x #u5,Da,Dn; sub.x #u5,Da,Dn; suba #u5,Ra,Rn; suba.lin #u5,Ra,Rn; subc.wo.l #u5,Da,Dn; subc.wo.leg.x #u5,Da,Dn; subc.wo.x #u5,Da,Dn; cmpa.gt.u #u5,Ra,Pmn; cmpa.gt.u #u5,Ra,Pn; cmpa.le.u #u5,Ra,Pmn; cmpa.le.u #u5,Ra,Pn; fix2flt.l.2sp #u5,Dab,Dmn; fix2flt.l.sp #u5,Da,Dn; fix2flt.ul.2sp #u5,Dab,Dmn; fix2flt.ul.sp #u5,Da,Dn; flt2fix.sp.2x #u5,Dab,Dmn; flt2fix.sp.l #u5,Da,Dn; flt2fix.sp.x #u5,Da,Dn
Type	83, R_SC3900_u5_t2_0_0
Size	5
Signedness	Unsigned
Alignment	0
Shift	0
Overflow	Verify
Encoding	-uuuuu---------- ----------
Applies to	43210 clric #u5_t2; rte.cic #u5_t2
Type	84, R_SC3900_u5_t3_0_0
Size	5
Signedness	Unsigned
Alignment	0
Shift	0
Overflow	Verify
Encoding	---------- ----uuuuu--------
Applies to	43210 extracta #U5,#u5,Ra,Rn; extracta.u #U5,#u5,Ra,Rn; inserta #U5,#u5_t3,Ra,Rn
Type	88, R_SC3900_u4dyn_0_0
Size	4
Signedness	Unsigned

Table continues on the next page...

Table 4-2. Relocation Type Definitions (continued)

Alignment	0
Shift	0
Overflow	Verify
Encoding	`------uuuu------ ----------------` `3210`
Applies to	extract.u.x #U6,#u6,Da,Dn; extract.x #U6,#u6,Da,Dn; insert.x #U6,#u6,Da,Dn
Type	91, `R_SC3900_u10dyn_0_0`
Size	5
Signedness	Unsigned
Alignment	0
Shift	0
Overflow	Verify
Encoding	`-uuuuuuuuuu----- ----------------` `9876543210`
Applies to	debuge.0 #u10; debuge.1 #u10; debuge.2 #u10; debuge.3 #u10; trap.0 #u10; trap.1 #u10
Type	105, `R_SC3900_RelAdd19_t1_1_1` (PC -relative)
Size	20
Signedness	Signed
Alignment	1
Shift	1
Overflow	Verify

Encoding	`-ssss---------- -sssssssssssss ----------------`
	`1111 111111987654321`
	`9876 543210`

Applies to	break.0 RelAdd19_t1; break.1 RelAdd19_t1; break.2 RelAdd19_t1; break.3 RelAdd19_t1; cont.0 RelAdd19_t1,Ra; cont.1 RelAdd19_t1,Ra; cont.2 RelAdd19_t1,Ra; cont.3 RelAdd19_t1,Ra; skip.0 RelAdd19_t1; skip.0.u RelAdd19_t1; skip.1 RelAdd19_t1; skip.1.u RelAdd19_t1; skip.2 RelAdd19_t1; skip.2.u RelAdd19_t1; skip.3 RelAdd19_t1; skip.3.u RelAdd19_t1
Type	108, `R_SC3900_U5_t1_0_0`
Size	5
Signedness	Unsigned
Alignment	0
Shift	0
Overflow	Verify
Encoding	`----------uuuuu ----------------`

Table continues on the next page...

Table 4-2. Relocation Type Definitions (continued)

Applies to	43210 inserta #U5,#u5_t3,Ra,Rn
Type Size Signedness Alignment Shift Overflow Encoding Applies to	111, R_SC3900_s16_t2_0_0 16 Signed, biextended 0 0 Verify --------------ss --sssssssssssss ---------- ---------- 11 11119876543210 54 3210 mac.leg.x #s16_t2,Da.h,Dn; mac.leg.x #s16_t2,Da.l,Dn; mac.s.x #s16_t2,Da.h,Dn; mac.s.x #s16_t2,Da.l,Dn; mac.x #s16_t2,Da.h,Dn; mac.x #s16_t2,Da.l,Dn; mpy.i.x #s16_t2,Da.l,Dn
Type Size Signedness Alignment Shift Overflow Encoding Applies to	112, R_SC3900_s32V2_0_0 32 Signed 0 0 Verify -sssssssssssssss ---------- ---------- ---------- 33222222222211111111119876543 210 109876543210987 6543210 and.x #s32,Da,Dn; cmp.eq.l #s32,Da,Pmn; cmp.eq.l #s32,Da,Pn; cmp.eq.x #s32,Da,Pmn; cmp.eq.x #s32,Da,Pn; cmp.gt.l #s32,Da,Pmn; cmp.gt.l #s32,Da,Pn; cmp.gt.x #s32,Da,Pmn; cmp.gt.x #s32,Da,Pn; cmp.le.l #s32,Da,Pmn; cmp.le.l #s32,Da,Pn; cmp.le.x #s32,Da,Pmn; cmp.le.x #s32,Da,Pn; cmp.ne.l #s32,Da,Pmn; cmp.ne.l #s32,Da,Pn; cmp.ne.x #s32,Da,Pmn; cmp.ne.x #s32,Da,Pn; cmpd.eq.x #s32,Da,Dn; cmpd.ge.x #s32,Da,Dn; cmpd.gt.x #s32,Da,Dn; cmpd.le.x #s32,Da,Dn; cmpd.lt.x #s32,Da,Dn; cmpd.ne.x #s32,Da,Dn; tfr.2x #s32,Dmn; tfr.x #s32,Dn
Type Size Signedness Alignment Shift Overflow Encoding	117, R_SC3900_u32V2_0_0 32 Unsigned 0 0 Verify -uuuuuuuuuuuuuuu --uuuuuuuuuuuuuu ------uuu------- ----------

Table continues on the next page...

Table 4-2. Relocation Type Definitions (continued)

Applies to	332222222222111 11111119876543 210 109876543210987 6543210 and.l #u32,Da,Dn; cmp.gt.u.l #u32,Da,Pmn; cmp.gt.u.l #u32,Da,Pn; cmp.le.u.l #u32,Da,Pmn; cmp.le.u.l #u32,Da,Pn; eor.l #u32,Da,Dn; or.l #u32,Da,Dn; tstbm.c.l #u32,Da,Pmn; tstbm.c.l #u32,Da,Pn; tstbm.s.l #u32,Da,Pmn; tstbm.s.l #u32,Da,Pn; cmp.gt.u.x #u32,Da,Pmn; cmp.gt.u.x #u32,Da,Pn; cmp.le.u.x #u32,Da,Pmn; cmp.le.u.x #u32,Da,Pn
Type	119, R_SC3900_u3_2_2_2
Size	5
Signedness	Unsigned
Alignment	2
Shift	2
Overflow	Verify
Encoding	---------- ----uuu----------
Applies to	432 cneg.n.4b #u3_2,Da,Db,Dn; cneg.n.4t #u3_2,Da,Dcd,Dmn; cneg.n.4w #u3_2,Da,Dcd,Dmn; cneg.n.rev.4b #u3_2,Da,Db,Dn; cneg.n.rev.4t #u3_2,Da,Dcd,Dmn; cneg.n.rev.4w #u3_2,Da,Dcd,Dmn; cnegadd.n.2x #u3_2,Da,Dcd,Dmn; cnegadd.n.rev.2x #u3_2,Da,Dcd,Dmn
Type	121, R_SC3900_ebit3_0_0
Size	3
Signedness	Unsigned
Alignment	0
Shift	0
Overflow	Verify
Encoding	----------u----- ------uu---------
Applies to	2 10 extract.u.x #U6,#u6,Da,Dn; extract.x #U6,#u6,Da,Dn; insert.x #U6,#u6,Da,Dn
Type	122, R_SC3900_sbit3_0_0
Size	3
Signedness	Unsigned
Alignment	0
Shift	0
Overflow	Verify
Encoding	---------- ---------uuu-----
Applies to	210 extract.u.x #U6,#u6,Da,Dn; extract.x #U6,#u6,Da,Dn; insert.x #U6,#u6,Da,Dn

Table continues on the next page...

Table 4-2. Relocation Type Definitions (continued)

Type	124, `R_SC3900_ux5_0_0`
Size	5
Signedness	Signed
Alignment	0
Shift	0
Overflow	Verify
Encoding	`------sssss----- ----------------` `43210`
Special	
Applies to	Value is computed before applying conform to the table: -1 -> 0b11111 0 -> 0b00000 1 -> 0b00001 2 -> 0b00010 3 -> 0b00011 4 -> 0b00100 5 -> 0b00101 6 -> 0b00110 7 -> 0b00111 8 -> 0b01000 9 -> 0b01001 10 -> 0b01010 11 -> 0b01011 12 -> 0b01100 13 -> 0b01101 14 -> 0b01110 15 -> 0b01111 16 -> 0b10000 17 -> 0b10001 18 -> 0b10010 19 -> 0b10011 20 -> 0b10100 21 -> 0b10101 22 -> 0b10110 23 -> 0b10111 24 -> 0b11000 25 -> 0b11001 26 -> 0b11010

Table continues on the next page...

Table 4-2. Relocation Type Definitions (continued)

	27 -> 0b11011 28 -> 0b11100 29 -> 0b11101 30 -> 0b11110 cmp.eq.l #ux5,Da,Pmn; cmp.eq.l #ux5,Da,Pn; cmp.eq.x #ux5,Da,Pmn; cmp.eq.x #ux5,Da,Pn; cmp.gt.l #ux5,Da,Pmn; cmp.gt.l #ux5,Da,Pn; cmp.gt.x #ux5,Da,Pmn; cmp.gt.x #ux5,Da,Pn; cmp.le.l #ux5,Da,Pmn; cmp.le.l #ux5,Da,Pn; cmp.le.x #ux5,Da,Pmn; cmp.le.x #ux5,Da,Pn; cmp.ne.l #ux5,Da,Pmn; cmp.ne.l #ux5,Da,Pn; cmp.ne.x #ux5,Da,Pmn; cmp.ne.x #ux5,Da,Pn; cmpa.eq #ux5,Ra,Pmn; cmpa.eq #ux5,Ra,Pn; cmpa.gt #ux5,Ra,Pmn; cmpa.gt #ux5,Ra,Pn; cmpa.le #ux5,Ra,Pmn; cmpa.le #ux5,Ra,Pn; cmpa.ne #ux5,Ra,Pmn; cmpa.ne #ux5,Ra,Pn
Type	126, `R_SC3900_ue5_0_0`
Size	5
Signedness	Unsigned
Alignment	0
Shift	0
Overflow	Verify
Encoding	`------sssss----- ----------------` `43210`
Special:	Value is computed before applying conform to the table: 32 -> 0b00000 1 -> 0b00001 2 -> 0b00010 3 -> 0b00011 4 -> 0b00100 5 -> 0b00101 6 -> 0b00110 7 -> 0b00111 8 -> 0b01000 9 -> 0b01001 10 -> 0b01010 11 -> 0b01011 12 -> 0b01100 13 -> 0b01101 14 -> 0b01110 15 -> 0b01111 16 -> 0b10000 17 -> 0b10001
Applies to	

Table continues on the next page...

Table 4-2. Relocation Type Definitions (continued)

	18 -> 0b10010 19 -> 0b10011 20 -> 0b10100 21 -> 0b10101 22 -> 0b10110 23 -> 0b10111 24 -> 0b11000 25 -> 0b11001 26 -> 0b11010 27 -> 0b11011 28 -> 0b11100 29 -> 0b11101 30 -> 0b11110 31 -> 0b11111 ash.lft.2x #ue5,Dab,Dmn; ash.lft.s.2x #ue5,Dab,Dmn; ash.lft.s.x #ue5,Da,Dn; ash.lft.x #ue5,Da,Dn; ash.rgt.2x #ue5,Dab,Dmn; ash.rgt.s.2x #ue5,Dab,Dmn; ash.rgt.s.x #ue5,Da,Dn; ash.rgt.x #ue5,Da,Dn; lsh.rgt.2x #ue5,Dab,Dmn; lsh.rgt.x #ue5,Da,Dn

Type
Size
Signedness
Alignment
Shift
Overflow
Encoding
Applies to
	128, `R_SC3900_u6_t2_0_0`
	6
	Unsigned
	0
	0
	Verify
	`------uuuuuu---- ----------------`
	`543210`
	pinita #uena,#uval; prda.rst #uena,#u12_t1,Ra; prda.swp #uena,#u12_t1,Rn; prda.sav #uena,#u12_t1,Rn

Type
Size
Signedness
Alignment
Shift
Overflow
Encoding
Applies to
	129, `R_SC3900_u6_t3_0_0`
	6
	Unsigned
	0
	0
	Verify
	`------uuuuu----- ----u-----------`
	`54321 0`
	ash.lft.ll #u6_t3,Dab,Dmn; ash.rgt.ll #u6_t3,Dab,Dmn; lsh.rgt.ll #u6_t3,Dab,Dmn

Table continues on the next page...

Table 4-2. Relocation Type Definitions (continued)

Type	134, R_SC3900_AbsAdd31_t2_1_1
Size	32
Signedness	Unsigned
Alignment	1
Shift	1
Overflow	Verify
Encoding	-uuuuuuuuuuuuuuu -uuuuuuuuuuuuuuu ------u---------- 332222222222111 111111198765432 1 109876543210987 6543210
Applies to	jmp AbsAdd31_t2; jmp.nobtb AbsAdd31_t2; jmprf AbsAdd31_t2; jsr AbsAdd31_t2; jsr.nobtb AbsAdd31_t2
Type	138, R_SC3900_s16_t6_0_0
Size	16
Signedness	Signed, biextended
Alignment	0
Shift	0
Overflow	Verify
Encoding	-sssssssssssssss ------s--------- ----------------- 111119876543210 1 43210 5
Applies to	adda #s16_t7,Ra,Rn; adda.lin #s16_t7,Ra,Rn; cmpa.eq #s16_t7,Ra,Pmn; cmpa.eq #s16_t7,Ra,Pn; cmpa.gt #s16_t7,Ra,Pmn; cmpa.gt #s16_t7,Ra,Pn; cmpa.le #s16_t7,Ra,Pmn; cmpa.le #s16_t7,Ra,Pn; cmpa.ne #s16_t7,Ra,Pmn; cmpa.ne #s16_t7,Ra,Pn; ld.16l (Rn+s16_t7),Da__Dp; ld.2b (Rn+s16_t7),Dab; ld.2b (Rn+s16_t7),Rab; ld.2bf (Rn+s16_t7),Dab; ld.2f (Rn +s16_t7),Dab; ld.2l (Rn+s16_t7),Dab; ld.2l (Rn+s16_t7),Dac; ld.2l (Rn +s16_t7),Rab; ld.2l (sp+s16_t7),Dab; ld.2l (sp+s16_t7),Rab; ld.2w (Rn +s16_t7),Dab; ld.2w (Rn+s16_t7),Rab; ld.2x (Rn+s16_t7),Dab; ld.4b (Rn +s16_t7),Dabcd; ld.4b (Rn+s16_t7),Rabcd; ld.4bf (Rn+s16_t7),Dabcd; ld.4f (Rn+s16_t7),Dabcd; ld.4l (Rn+s16_t7),Dabcd; ld.4l (Rn+s16_t7),Dacbd; ld.4l (Rn+s16_t7),Daceg; ld.4l (Rn+s16_t7),Rabcd; ld.4w (Rn+s16_t7),Dabcd; ld. 4w (Rn+s16_t7),Rabcd; ld.4x (Rn+s16_t7),Dabcd; ld.8b (Rn +s16_t7),Da__Dh; ld.8bf (Rn+s16_t7),Da__Dh; ld.8f (Rn+s16_t7),Da__Dh; ld.8l (Rn+s16_t7),Da__Dh; ld.8l (Rn+s16_t7),Dacegbdfh; ld.8l (Rn +s16_t7),Daceghjln; ld.8w (Rn+s16_t7),Da__Dh; ld.8x (Rn+s16_t7),Da__Dh; ld.b (Rn+s16_t7),Da; ld.b (Rn+s16_t7),Ra; ld.b (sp+s16_t7),Da; ld.b (sp +s16_t7),Ra; ld.bf (Rn+s16_t7),Da; ld.bf (sp+s16_t7),Da; ld.f (Rn +s16_t7),Da; ld.f (sp+s16_t7),Da; ld.l (Rn+s16_t7),Da; ld.l (Rn+s16_t7),Ra; ld.l (sp+s16_t7),Da; ld.l (sp+s16_t7),Ra; ld.par.w (Rn+s16_t7),Da.h; ld.par.w (Rn+s16_t7),Da.l; ld.par.w (sp+s16_t7),Da.h; ld.par.w (sp+s16_t7),Da.l; ld.u. 2b (Rn+s16_t7),Dab; ld.u.2b (Rn+s16_t7),Rab; ld.u.2l (Rn+s16_t7),Dab; ld.u. 2l (sp+s16_t7),Dab; ld.u.2w (Rn+s16_t7),Dab; ld.u.2w (Rn+s16_t7),Rab; ld.u.4b (Rn+s16_t7),Dabcd; ld.u.4b (Rn+s16_t7),Rabcd; ld.u.4l (Rn +s16_t7),Dabcd; ld.u.4w (Rn+s16_t7),Dabcd; ld.u.4w (Rn+s16_t7),Rabcd; ld.u.8b (Rn+s16_t7),Da__Dh; ld.u.8l (Rn+s16_t7),Da__Dh; ld.u.8w (Rn +s16_t7),Da__Dh; ld.u.b (Rn+s16_t7),Da; ld.u.b (Rn+s16_t7),Ra; ld.u.b (sp

Table continues on the next page...

Table 4-2. Relocation Type Definitions

	+s16_t7),Da; ld.u.b (sp+s16_t7),Ra; ld.u.l (Rn+s16_t7),Da; ld.u.l (sp +s16_t7),Da; ld.u.w (Rn+s16_t7),Da; ld.u.w (Rn+s16_t7),Ra; ld.u.w (sp +s16_t7),Da; ld.u.w (sp+s16_t7),Ra; ld.w (Rn+s16_t7),Da; ld.w (Rn +s16_t7),Ra; ld.w (sp+s16_t7),Da; ld.w (sp+s16_t7),Ra; ld.x (Rn +s16_t7),Da; ld.x (sp+s16_t7),Da; ld2.16b (Rn+s16_t7),Da__Dh; ld2.16bf (Rn+s16_t7),Da__Dh; ld2.16f (Rn+s16_t7),Da__Dh; ld2.16f (Rn +s16_t7),Da__DhMod8; ld2.16f (Rn+s16_t7),Da__DhMod8p; ld2.16f (Rn +s16_t7),Da__DhR3; ld2.16f (Rn+s16_t7),Da__DhR5; ld2.2b (Rn +s16_t7),Da; ld2.2bf (Rn+s16_t7),Da; ld2.2f (Rn+s16_t7),Da; ld2.32f (Rn +s16_t7),Da__Dp; ld2.32f (Rn+s16_t7),Da__DpMod16; ld2.4b (Rn +s16_t7),Dab; ld2.4bf (Rn+s16_t7),Dab; ld2.4f (Rn+s16_t7),Dab; ld2.8b (Rn +s16_t7),Dabcd; ld2.8bf (Rn+s16_t7),Dabcd; ld2.8f (Rn+s16_t7),Dabcd; ld2.8f (Rn+s16_t7),Dacdb; ld2.u.16b (Rn+s16_t7),Da__Dh; ld2.u.2b (Rn+s16_t7),Da; ld2.u.4b (Rn+s16_t7),Dab; ld2.u.8b (Rn +s16_t7),Dabcd; ldsh2.dn.2f (Rn+s16_t7),Dab; ldsh2.dn.2f (sp+s16_t7),Dab; ldsh2.up.2f (Rn+s16_t7),Dab; ldsh2.up.2f (sp+s16_t7),Dab; st.16l Da__Dp, (Rn+s16_t7); st.2b Dab,(Rn+s16_t7); st.2b Rab,(Rn+s16_t7); st.2bf Dab,(Rn +s16_t7); st.2f Dab,(Rn+s16_t7); st.2l Dab,(Rn+s16_t7); st.2l Dab,(sp +s16_t7); st.2l Dac,(Rn+s16_t7); st.2l Rab,(Rn+s16_t7); st.2l Rab,(sp +s16_t7); st.2w Dab,(Rn+s16_t7); st.2w Rab,(Rn+s16_t7); st.2x Dab,(Rn +s16_t7); st.4b Dabcd,(Rn+s16_t7); st.4b Rabcd,(Rn+s16_t7); st.4bf Dabcd, (Rn+s16_t7); st.4f Dabcd,(Rn+s16_t7); st.4l Dabcd,(Rn+s16_t7); st.4l Rabcd, (Rn+s16_t7); st.4w Dabcd,(Rn+s16_t7); st.4w Rabcd,(Rn+s16_t7); st.4x Dabcd,(Rn+s16_t7); st.8b Da__Dh,(Rn+s16_t7); st.8bf Da__Dh,(Rn +s16_t7); st.8f Da__Dh,(Rn+s16_t7); st.8l Da__Dh,(Rn+s16_t7); st.8w Da__Dh,(Rn+s16_t7); st.8x Da__Dh,(Rn+s16_t7); st.b Da,(Rn+s16_t7); st.b Da,(sp+s16_t7); st.b Ra,(Rn+s16_t7); st.b Ra,(sp+s16_t7); st.bf Da,(Rn +s16_t7); st.bf Da,(sp+s16_t7); st.f Da,(Rn+s16_t7); st.f Da,(sp+s16_t7); st.l Da,(Rn+s16_t7); st.l Da,(sp+s16_t7); st.l Ra,(Rn+s16_t7); st.l Ra,(sp +s16_t7); st.srs.2bf Dab,(Rn+s16_t7); st.srs.2f Dab,(Rn+s16_t7); st.srs.2l Dab,(Rn+s16_t7); st.srs.4bf Dabcd,(Rn+s16_t7); st.srs.4f Dabcd,(Rn +s16_t7); st.srs.4l Dabcd,(Rn+s16_t7); st.srs.8bf Da__Dh,(Rn+s16_t7); st.srs.8f Da__Dh,(Rn+s16_t7); st.srs.8l Da__Dh,(Rn+s16_t7); st.srs.bf Da, (Rn+s16_t7); st.srs.f Da,(Rn+s16_t7); st.srs.l Da,(Rn+s16_t7); st.srs.wh.8f Da__Dh,(Rn+s16_t7); st.srs.wl.8f Da__Dh,(Rn+s16_t7); st.w Da,(Rn +s16_t7); st.w Da,(sp+s16_t7); st.w Ra,(Rn+s16_t7); st.w Ra,(sp+s16_t7); st.x Da,(Rn+s16_t7); st.x Da,(sp+s16_t7); st2.16b Da__Dh,(Rn+s16_t7); st2.16bf Da__Dh,(Rn+s16_t7); st2.2b Da,(Rn+s16_t7); st2.2bf Da,(Rn +s16_t7); st2.4b Dab,(Rn+s16_t7); st2.4bf Dab,(Rn+s16_t7); st2.8b Dabcd, (Rn+s16_t7); st2.8bf Dabcd,(Rn+s16_t7); st2.srs.16bf Da__Dh,(Rn+s16_t7); st2.srs.16f Da__Dh,(Rn+s16_t7); st2.srs.16f Da__DhMod8,(Rn+s16_t7); st2.srs.16f Da__DhMod8p,(Rn+s16_t7); st2.srs.2bf Da,(Rn+s16_t7); st2.srs. 2f Da,(Rn+s16_t7); st2.srs.32f Da__Dp,(Rn+s16_t7); st2.srs.4bf Dab,(Rn +s16_t7); st2.srs.4f Dab,(Rn+s16_t7); st2.srs.8bf Dabcd,(Rn+s16_t7); st2.srs.8f Dabcd,(Rn+s16_t7); adda.lin #s16_t7,sp,Rn; ld.16l (sp +s16_t7),Da__Dp; ld.2b (sp+s16_t7),Dab; ld.2b (sp+s16_t7),Rab; ld.2bf (sp +s16_t7),Dab; ld.2f (sp+s16_t7),Dab; ld.2l (sp+s16_t7),Dac; ld.2w (sp +s16_t7),Dab; ld.2w (sp+s16_t7),Rab; ld.2x (sp+s16_t7),Dab; ld.4b (sp +s16_t7),Dabcd; ld.4b (sp+s16_t7),Rabcd; ld.4bf (sp+s16_t7),Dabcd; ld.4f (sp+s16_t7),Dabcd; ld.4l (sp+s16_t7),Dabcd; ld.4l (sp+s16_t7),Dacbd; ld.4l (sp+s16_t7),Daceg; ld.4l (sp+s16_t7),Rabcd; ld.4w (sp+s16_t7),Dabcd; ld.4w (sp+s16_t7),Rabcd; ld.4x (sp+s16_t7),Dabcd; ld.8b (sp+s16_t7),Da__Dh; ld. 8bf (sp+s16_t7),Da__Dh; ld.8f (sp+s16_t7),Da__Dh; ld.8l (sp +s16_t7),Da__Dh; ld.8l (sp+s16_t7),Dacegbdfh; ld.8l (sp+s16_t7),Daceghjln; ld.8w (sp+s16_t7),Da__Dh; ld.8x (sp+s16_t7),Da__Dh; ld.u.2b (sp

Table continues on the next page...

Table 4-2. Relocation Type Definitions (continued)

	+s16_t7),Dab; ld.u.2b (sp+s16_t7),Rab; ld.u.2w (sp+s16_t7),Dab; ld.u.2w (sp +s16_t7),Rab; ld.u.4b (sp+s16_t7),Dabcd; ld.u.4b (sp+s16_t7),Rabcd; ld.u.4l (sp+s16_t7),Dabcd; ld.u.4w (sp+s16_t7),Dabcd; ld.u.4w (sp+s16_t7),Rabcd; ld.u.8b (sp+s16_t7),Da__Dh; ld.u.8l (sp+s16_t7),Da__Dh; ld.u.8w (sp +s16_t7),Da__Dh; ld2.16b (sp+s16_t7),Da__Dh; ld2.16bf (sp +s16_t7),Da__Dh; ld2.16f (sp+s16_t7),Da__Dh; ld2.16f (sp +s16_t7),Da__DhMod8; ld2.16f (sp+s16_t7),Da__DhMod8p; ld2.16f (sp+s16_t7),Da__DhR3; ld2.16f (sp+s16_t7),Da__DhR5; ld2.2b (sp +s16_t7),Da; ld2.2bf (sp+s16_t7),Da; ld2.2f (sp+s16_t7),Da; ld2.32f (sp +s16_t7),Da__Dp; ld2.32f (sp+s16_t7),Da__DpMod16; ld2.4b (sp +s16_t7),Dab; ld2.4bf (sp+s16_t7),Dab; ld2.4f (sp+s16_t7),Dab; ld2.8b (sp +s16_t7),Dabcd; ld2.8bf (sp+s16_t7),Dabcd; ld2.8f (sp+s16_t7),Dabcd; ld2.8f (sp+s16_t7),Dacdb; ld2.u.16b (sp+s16_t7),Da__Dh; ld2.u.2b (sp +s16_t7),Da; ld2.u.4b (sp+s16_t7),Dab; ld2.u.8b (sp+s16_t7),Dabcd; st.16l Da__Dp,(sp+s16_t7); st.2b Dab,(sp+s16_t7); st.2b Rab,(sp+s16_t7); st.2bf Dab,(sp+s16_t7); st.2f Dab,(sp+s16_t7); st.2l Dac,(sp+s16_t7); st.2w Dab, (sp+s16_t7); st.2w Rab,(sp+s16_t7); st.2x Dab,(sp+s16_t7); st.4b Dabcd,(sp +s16_t7); st.4b Rabcd,(sp+s16_t7); st.4bf Dabcd,(sp+s16_t7); st.4f Dabcd, (sp+s16_t7); st.4l Dabcd,(sp+s16_t7); st.4l Rabcd,(sp+s16_t7); st.4w Dabcd, (sp+s16_t7); st.4w Rabcd,(sp+s16_t7); st.4x Dabcd,(sp+s16_t7); st.8b Da__Dh,(sp+s16_t7); st.8bf Da__Dh,(sp+s16_t7); st.8f Da__Dh,(sp+s16_t7); st.8l Da__Dh,(sp+s16_t7); st.8w Da__Dh,(sp+s16_t7); st.8x Da__Dh,(sp +s16_t7); st.srs.16l Da__Dp,(Rn+s16_t7); st.srs.16l Da__Dp,(sp+s16_t7); st.srs.2bf Dab,(sp+s16_t7); st.srs.2f Dab,(sp+s16_t7); st.srs.2l Dab,(sp +s16_t7); st.srs.4bf Dabcd,(sp+s16_t7); st.srs.4f Dabcd,(sp+s16_t7); st.srs.4l Dabcd,(sp+s16_t7); st.srs.8bf Da__Dh,(sp+s16_t7); st.srs.8f Da__Dh,(sp +s16_t7); st.srs.8l Da__Dh,(sp+s16_t7); st.srs.bf Da,(sp+s16_t7); st.srs.f Da, (sp+s16_t7); st.srs.l Da,(sp+s16_t7); st.srs.wh.8f Da__Dh,(sp+s16_t7); st.srs.wl.8f Da__Dh,(sp+s16_t7); st2.16b Da__Dh,(sp+s16_t7); st2.16bf Da__Dh,(sp+s16_t7); st2.2b Da,(sp+s16_t7); st2.2bf Da,(sp+s16_t7); st2.4b Dab,(sp+s16_t7); st2.4bf Dab,(sp+s16_t7); st2.8b Dabcd,(sp+s16_t7); st2.8bf Dabcd,(sp+s16_t7); st2.srs.16bf Da__Dh,(sp+s16_t7); st2.srs.16f Da__Dh,(sp+s16_t7); st2.srs.16f Da__DhMod8,(sp+s16_t7); st2.srs.16f Da__DhMod8p,(sp+s16_t7); st2.srs.2bf Da,(sp+s16_t7); st2.srs.2f Da,(sp +s16_t7); st2.srs.32f Da__Dp,(sp+s16_t7); st2.srs.4bf Dab,(sp+s16_t7); st2.srs.4f Dab,(sp+s16_t7); st2.srs.8bf Dabcd,(sp+s16_t7); st2.srs.8f Dabcd, (sp+s16_t7);
Type	139, `R_SC3900_s32_t5_0_0`
Size	32
Signedness	Signed
Alignment	0
Shift	0
Overflow	Verify
Encoding	`-sssssssssssssss -sssssssss----- ----sssssss------`

Applies to	`111119876543210 3222222223 2211111`
	`43210 0987654321 1098765`
	tfra.l #s32_t5,Rn

Type	140, `R_SC3900_s32_t6_0_0`

Table continues on the next page...

Table 4-2. Relocation Type Definitions (continued)

Size	32
Signedness	Signed
Alignment	0
Shift	0
Overflow	Verify
Encoding	`-sssssssssssssss -sssssssss------ ----sssssssss-----`

Applies to	`111119876543210 322222222 22111113`
	`43210 098765432 10987651`
	move.l #s32_t6,Dn

Type	143, `R_SC3900_u12_t1_0_0`
Size	12
Signedness	Unsigned
Alignment	0
Shift	0
Overflow	Verify
Encoding	`----uuuuuuuuuuuu ---------------- -----------------`

Applies to	`119876543210`
	`10`
	prda.rst #uena,#u12_t1,Ra; prda.swp #uena,#u12_t1,Rn; prda.sav #uena,#u12_t1,Rn

Type	144, `R_SC3900_u16_t4_0_0`
Size	16
Signedness	Unsigned
Alignment	0
Shift	0
Overflow	Verify
Encoding	`-uuuuuuuuuuuuuuu ------u--------- -----------------`

Applies to	`111119876543210 1`
	`43210 5`
	bmchga #u16,C4.h; bmchga #u16,C4.l; bmchga #u16,Rn.h; bmchga #u16,Rn.l; bmclra #u16,C4.h; bmclra #u16,C4.l; bmclra #u16,Rn.h; bmclra #u16,Rn.l; bmseta #u16,C4.h; bmseta #u16,C4.l; bmseta #u16,Rn.h; bmseta #u16,Rn.l; bmtsta.c #u16,C4.h,Pmn; bmtsta.c #u16,C4.h,Pn; bmtsta.c #u16,C4.l,Pmn; bmtsta.c #u16,C4.l,Pn; bmtsta.c #u16,Ra.h,Pmn; bmtsta.c #u16,Ra.h,Pn; bmtsta.c #u16,Ra.l,Pmn; bmtsta.c #u16,Ra.l,Pn; bmtsta.s #u16,C4.h,Pmn; bmtsta.s #u16,C4.h,Pn; bmtsta.s #u16,C4.l,Pmn; bmtsta.s #u16,C4.l,Pn; bmtsta.s #u16,Ra.h,Pmn; bmtsta.s #u16,Ra.h,Pn; bmtsta.s #u16,Ra.l,Pmn; bmtsta.s #u16,Ra.l,Pn; cmpa.gt.u #u16,Ra,Pmn; cmpa.gt.u #u16,Ra,Pn; cmpa.le.u #u16,Ra,Pmn; cmpa.le.u #u16,Ra,Pn

Type	146, `R_SC3900_u16_t6_0_0`

Table continues on the next page...

Table 4-2. Relocation Type Definitions (continued)

Size	16
Signedness	Unsigned
Alignment	0
Shift	0
Overflow	Verify
Encoding	-uuuuuuuuuuuuuuu ---------- ----u----------
Applies to	111119876543210 1 43210 5 mpyadda.i #u16_t6,Ra,Rb,Rn
Type	147, R_SC3900_u1_t2_0_0
Size	1
Signedness	Unsigned
Alignment	0
Shift	0
Overflow	Verify
Encoding	----------u----- ----------
Applies to	0 chcf #u1_t2,SR.RM; chcf #u1_t2,SR.SM
Type	148, R_SC3900_u2_t2_0_0
Size	2
Signedness	Unsigned
Alignment	0
Shift	0
Overflow	Verify
Encoding	---------uu----- ----------
Applies to	10 chcf #u2_t2,SR.SWM2; chcf #u2_t2,SR2.ASP
Type	149, R_SC3900_u3_t3_0_0
Size	3
Signedness	Unsigned
Alignment	0
Shift	0
Overflow	Verify
Encoding	--------uuu----- ----------
Applies to	210 chcf #u3_t3,SR.SCM

Table continues on the next page...

Table 4-2. Relocation Type Definitions (continued)

Type	153, R_SC3900_u8_t2_0_0
Size	8
Signedness	Unsigned
Alignment	0
Shift	0
Overflow	Verify
Encoding	-uuuu----------- ----uuuu--------
Applies to	7654 3210 pcalc #u8_t2,Pabc,Pmn; pcalc #u8_t2,Pabc,Pn; pcalca #u8_t2,Pabc,Pmn; pcalca #u8_t2,Pabc,Pn
Type	154, R_SC3900_u9_t2_0_0
Size	9
Signedness	Unsigned
Alignment	0
Shift	0
Overflow	Verify
Encoding	-uuuuuuuuu------ ----------------
Applies to	087654321 ld.b (sp-u9),Da; ld.b (sp-u9),Ra; ld.bf (sp-u9),Da; ld.u.b (sp-u9),Da; ld.u.b (sp-u9),Ra; st.b Da,(sp-u9); st.b Ra,(sp-u9); st.bf Da,(sp-u9)
Type	155, R_SC3900_u9_1_t2_1_1
Size	10
Signedness	Unsigned
Alignment	1
Shift	1
Overflow	Verify
Encoding	-uuuuuuuuu------ ----------------
Applies to	198765432 ld.f (sp-u9_1),Da; ld.u.w (sp-u9_1),Da; ld.u.w (sp-u9_1),Ra; ld.w (sp-u9_1),Da; ld.w (sp-u9_1),Ra; st.f Da,(sp-u9_1); st.w Da,(sp-u9_1); st.w Ra,(sp-u9_1)
Type	156, R_SC3900_u9_2_t2_2_2
Size	11
Signedness	Unsigned
Alignment	2
Shift	2
Overflow	Verify
Encoding	-uuuuuuuuu------ ----------------

Table continues on the next page...

Table 4-2. Relocation Type Definitions (continued)

Applies to	219876543 0 ld.l (sp-u9_2),Da; ld.l (sp-u9_2),Ra; ld.u.l (sp-u9_2),Da; st.l Da,(sp-u9_2); st.l Ra,(sp-u9_2)
Type	157, R_SC3900_u9_3_t2_3_3
Size	12
Signedness	Unsigned
Alignment	3
Shift	3
Overflow	Verify
Encoding	-uuuuuuuuu------ ----------------
Applies to	311987654 10 ld.x (sp-u9_3),Da; st.x Da,(sp-u9_3)
Type	158, R_SC3900_uval_0_0
Size	6
Signedness	Unsigned
Alignment	0
Shift	0
Overflow	Verify
Encoding	-uuuuu------0--- ----------------
Applies to	54321 pinita #uena,#uval
Type	166, R_SC3900_s16_t7_0_0
Size	16
Signedness	Signed, biextended
Alignment	0
Shift	0
Overflow	Verify
Encoding	-sssssssssssssss -------s-------- -----------------
Applies to	111119876543210 1 43210 5 adda.lin #s16,Ra,Rn; cmpa.eq #s16,Ra,Pmn; cmpa.eq #s16,Ra,Pn; cmpa.gt #s16,Ra,Pmn; cmpa.gt #s16,Ra,Pn; cmpa.le #s16,Ra,Pmn; cmpa.le #s16,Ra,Pn; cmpa.ne #s16,Ra,Pmn; cmpa.ne #s16,Ra,Pn; mpy32a.i #s16,Ra,Rn; adda.lin #s16,sp,Rn
Type	167, R_SC3900_u16_t7_0_0
Size	16

Table continues on the next page...

Table 4-2. Relocation Type Definitions (continued)

Signedness	Unsigned
Alignment	0
Shift	0
Overflow	Verify
Encoding	`-uuuuuuuuuuuuuuu -------u-------- -----------------`

Applies to	`111119876543210 1`
	`43210 5`
	cmpa.gt.u #u16_t7,Ra,Pmn; cmpa.gt.u #u16_t7,Ra,Pn; cmpa.le.u #u16_t7,Ra,Pmn; cmpa.le.u #u16_t7,Ra,Pn; move.par.w #u16_t7,Dn.h; move.par.w #u16_t7,Dn.l

Type	168, `R_SC3900_URRelAdd4_t3_1_1`
Size	5
Signedness	Unsigned
Alignment	1
Shift	1
Overflow	Verify
Encoding	`------------uuuu ---------------- -----------------`
Applies to	`4321` lpskip.0 RelAdd19_t3,URRelAdd4_t3; lpskip.0.u RelAdd19_t3,URRelAdd4_t3; lpskip.1 RelAdd19_t3,URRelAdd4_t3; lpskip.1.u RelAdd19_t3,URRelAdd4_t3; lpskip.2 RelAdd19_t3,URRelAdd4_t3; lpskip.2.u RelAdd19_t3,URRelAdd4_t3; lpskip.3 RelAdd19_t3,URRelAdd4_t3; lpskip.3.u RelAdd19_t3,URRelAdd4_t3; lpskip.sq0 RelAdd19_t3,URRelAdd4_t3; lpskip.sq0.u RelAdd19_t3,URRelAdd4_t3; lpskip.sq1 RelAdd19_t3,URRelAdd4_t3; lpskip.sq1.u RelAdd19_t3,URRelAdd4_t3; lpskip.sq2 RelAdd19_t3,URRelAdd4_t3; lpskip.sq2.u RelAdd19_t3,URRelAdd4_t3; lpskip.sq3 RelAdd19_t3,URRelAdd4_t3; lpskip.sq3.u RelAdd19_t3,URRelAdd4_t3
Type	170, `R_SC3900_u16_t8_0_0`
Size	16
Signedness	Unsigned
Alignment	0
Shift	0
Overflow	Verify
Encoding	`-uuuuuuuuuuuuuu-- ----uuu---------`

Applies to	`9876543210111 111`
	`210 543`
	doen.0 #u16_t8; doen.1 #u16_t8; doen.2 #u16_t8; doen.3 #u16_t8

Type	171, `R_SC3900_RelAdd19_t2_1_1` (PC - relative)
Size	20

Table continues on the next page...

Table 4-2. Relocation Type Definitions (continued)

Signedness	Signed
Alignment	1
Shift	1
Overflow	Verify
Encoding	`-uuuuuuuuuuuuuuu ------uuuu------`

Applies to	`111111987654321 1111`
	`543210 9876`
	bra RelAdd19_t2; bra.nobtb RelAdd19_t2; bsr RelAdd19_t2; bsr.nobtb RelAdd19_t2; lpend.0 RelAdd19_t2; lpend.1 RelAdd19_t2; lpend.2 RelAdd19_t2; lpend.3 RelAdd19_t2; lpend.sq0 RelAdd19_t2; lpend.sq1 RelAdd19_t2; lpend.sq2 RelAdd19_t2; lpend.sq3 RelAdd19_t2; lpst.0 RelAdd19_t2; lpst.1 RelAdd19_t2; lpst.2 RelAdd19_t2; lpst.3 RelAdd19_t2; lpst.sq0 RelAdd19_t2; lpst.sq1 RelAdd19_t2; lpst.sq2 RelAdd19_t2; lpst.sq3 RelAdd19_t2

Type	172, `R_SC3900_u6_t4_0_0`
Size	6
Signedness	Unsigned
Alignment	0
Shift	0
Overflow	Verify
Encoding	`-uuuuuu--------- ----------------`
Applies to	`543210` acator #u6_t4; acator.p #u6_t4,#u4_t6
Type	173, `R_SC3900_u4_t6_0_0`
Size	4
Signedness	Unsigned
Alignment	0
Shift	0
Overflow	Verify
Encoding	`-------uuuu--------------------`
Applies to	`3210` acator.p #u6_t4,#u4_t6; decor #u4_t6
Type	174, `R_SC3900_s9_3_3_3`
Size	12
Signedness	Signed, biextended
Alignment	3
Shift	3
Overflow	Verify
Encoding	`-sssssssss------ ----------------`

Table continues on the next page...

Table 4-2. Relocation Type Definitions (continued)

Applies to	119876543 10 adda.lin #s9_3,sp
Type Size Signedness Alignment Shift Overflow Encoding Applies to	175, R_SC3900_s9_0_0_0 9 Signed, biextended 0 0 Verify -sssssssss------ ---------------- 876543210 adda.lin #s9,sp,Rn; move.l #s9,Dn; tfra.l #s9,Rn
Type Size Signedness Alignment Shift Overflow Encoding Applies to	176, R_SC3900_u2_t3_0_0 2 Unsigned 0 0 Verify ---------------- ----uu---------- 10 fnd.max.4t #u2_t3,Da,Db,Dmn; fnd.max.4w #u2_t3,Da,Db,Dmn; fnd.maxm.4t #u2_t3,Da,Db,Dmn; fnd.maxm.4w #u2_t3,Da,Db,Dmn; fnd.min.4t #u2_t3,Da,Db,Dmn; fnd.min.4w #u2_t3,Da,Db,Dmn
Type Size Signedness Alignment Shift Overflow Encoding Applies to	177, R_SC3900_RelAdd19_t3_1_1 (PC - relative) 20 Signed 1 1 Verify ------ssss------ -sssssssssssssss ----------------- 1111 111111987654321 9876 543210 lpskip.0 RelAdd19_t3,URRelAdd4_t3; lpskip.0.u RelAdd19_t3,URRelAdd4_t3; lpskip.1 RelAdd19_t3,URRelAdd4_t3; lpskip.1.u RelAdd19_t3,URRelAdd4_t3; lpskip.2 RelAdd19_t3,URRelAdd4_t3; lpskip.2.u RelAdd19_t3,URRelAdd4_t3; lpskip.3 RelAdd19_t3,URRelAdd4_t3; lpskip.3.u RelAdd19_t3,URRelAdd4_t3; lpskip.sq0 RelAdd19_t3,URRelAdd4_t3; lpskip.sq0.u RelAdd19_t3,URRelAdd4_t3; lpskip.sq1 RelAdd19_t3,URRelAdd4_t3; lpskip.sq1.u RelAdd19_t3,URRelAdd4_t3; lpskip.sq2

Table continues on the next page...

Table 4-2. Relocation Type Definitions (continued)

	RelAdd19_t3,URRelAdd4_t3; lpskip.sq2.u RelAdd19_t3,URRelAdd4_t3; lpskip.sq3 RelAdd19_t3,URRelAdd4_t3; lpskip.sq3.u RelAdd19_t3,URRelAdd4_t3

Type
Size
Signedness
Alignment
Shift
Overflow
Encoding
Applies to
	178, `R_SC3900_u2_1_1_1`
	3
	Unsigned
	1
	1
	Verify
	`---------uu----- ----------------`
	`21`
	bit.expnd.2t #u2_1,Da,Dn; bit.expnd.2x #u2_1,Da,Dmn; bit.expnd.rev.2t #u2_1,Da,Dn; bit.expnd.rev.2x #u2_1,Da,Dmn

Type
Size
Signedness
Alignment
Shift
Overflow
Encoding
Applies to
	179, `R_SC3900_u2_2_2_2`
	4
	Unsigned
	2
	2
	Verify
	`---------uu----- ----------------`
	`32`
	bit.expnd.4t #u2_2,Da,Dmn; bit.expnd.rev.4t #u2_2,Da,Dmn

4.4.2 Relocation Stack

A relocation stack is a standard last-in-first-out data structure containing 32-bit values. A hosted environment must not place any arbitrary limit on the depth of the stack.

An embedded environment may impose any limit on stack depth or omit the relocation stack entirely (effectively, a maximum stack depth of zero).

When the relocation value cannot be expressed as a simple symbol value plus an addend, there are three special relocation types to evaluate an arbitrary expression on a relocation stack. These relocation types are referred to as extended relocations. Other relocation types are ordinary relocations.

A relocation type of 252 (R_STARCORE_PUSH_PC) indicates that the sum of the symbol value (the value of symbol number zero is zero) plus the signed r_addend value plus the current location counter value should be pushed onto the relocation stack.

A relocation type of 253 (R_STARCORE_PUSH) indicates that the sum of the symbol value (the value of symbol number zero is zero) plus the signed r_addend value should be pushed onto the relocation stack.

A relocation type of 254 (R_STARCORE_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) plus the signed r_addend value. Operations are shown in the below table. 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 4-3. Relocation Stack Operations

Relocation Value	Before		After	Operation
	Stack0	Stack1	Stack0	
0	x		X	No operation
1	x		-X	Negation (2s complement)
2	x		~X	Bitwise NOT (1s complement)
3	x		!X	Boolean NOT (zero -> 1, nonzero -> 0)
4	Y	X	X*Y	Multiplication
5	Y	X	X/Y	Division
6	Y	X	X%Y	Remainder
7	Y	X	X+Y	Addition
8	Y	X	X-Y	Subtraction
9	Y	X	X<<<Y	Logical shift left
10	Y	X	X>>>Y	Logical shift right
11	Y	X	X<<Y	Arithmetic shift left
12	Y	X	X>>Y	Arithmetic shift right
13	Y	X	X<Y	1 if X < Y, otherwise 0
14	Y	X	X<=Y	1 if X <= Y, otherwise 0
15	Y	X	X>Y	1 if X > Y, otherwise 0
16	Y	X	X>=Y	1 if X >= Y, otherwise 0
17	Y	X	X==Y	1 if X equals Y, otherwise 0
18	Y	X	X!=Y	1 if X does not equal, otherwise 0
19	Y	X	X&Y	Bitwise AND
20	Y	X	X^Y	Bitwise OR
21	Y	X	X|Y	Bitwise XOR

Table continues on the next page...

Table 4-3. Relocation Stack Operations (continued)

Relocation Value	Before		After	Operation
	Stack0	Stack1	Stack0	
22	Y	X	X&&Y	1 if X and Y both nonzero, otherwise 0
23	Y	X	X||Y	1 if X or Y or both nonzero, otherwise 0

Note that in most cases, the stack values are treated as unsigned. However, arithmetic shifts and logical shifts are treated differently.

- Logical shift left: Zeroes are shifted to the right.
- Logical shift right: Zeroes are shifted to the left.
- Arithmetic shift left: Zeroes are shifted to the right, and the most significant bit is always unaffected.
- Arithmetic shift right: Copies of the most significant bit are shifted to the left.

A relocation type of 255 (R_STARCORE_POP) indicates the end of a relocation expression, to be relocated using an ordinary relocation type from Relocation Types. The relocation type is specified by the sum of the symbol value (the value of symbol number zero is zero) plus the signed r_addend value.

When the R_STARCORE_POP operation is encountered, there should be exactly one value on the stack. This value, which is consumed by this operation, becomes the new relocation value for the ordinary relocation type specified in the R_STARCORE_POP relocation.

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

4.4.3 Instruction Address Versus VLES Address

Within a variable-length execution set (VLES), all instructions share a common value of the PC register, specifically the starting address of the VLES itself. The `r_offset` field of a relocation points to the instruction address, not the VLES address.

Consider a 2-instruction VLES that is described in the table below:

Table 4-4. 2-Instruction VLES

VLES Offset	Instruction
0 6	[tfra.l #>_symbol1,r0 tfra.l #>_symbol1,r1]

In table, the `r_offset` field for the `tfra.l #>_symbol1,r0` instruction is 0 and for the `tfra.l #>_symbol1,r1` instruction, the `r_offset` field is 6.

The generated relocation sequence is described in the below table.

Table 4-5. Generated Relocation Sequence

Offset	Addend	Type	Symbol
0x00000000	0x0000000000000000	253	_symbol1
0x00000000	0x000000000000008b	255	(symbol 0)
0x00000006	0x0000000000000000	253	_symbol1
0x00000006	0x000000000000008b	255	(symbol 0)

4.5 Note Section

The note section is optional. It contains object file vendor identification and application-specific object file comments.

If included, it follows the described format. Vendor identification format is shown in the figure below. It consists of the following:

- namesz: The string length (not counting null terminator) of the name. It is a 4-byte unsigned integer.
- descz: The size of the description entries. This is 12 bytes for the vendor ID note. The description fields contain the version, revision, and minor revision numbers of the producing entity (assembler or linker). Data is an unsigned 4-byte integer.
- type: Type equals 2 for the vendor identification note. It is a 4-byte unsigned integer in little-endian order.
- name: Null terminated string and padded, if necessary, to achieve a 4-byte boundary alignment which represents the vendor's identification.

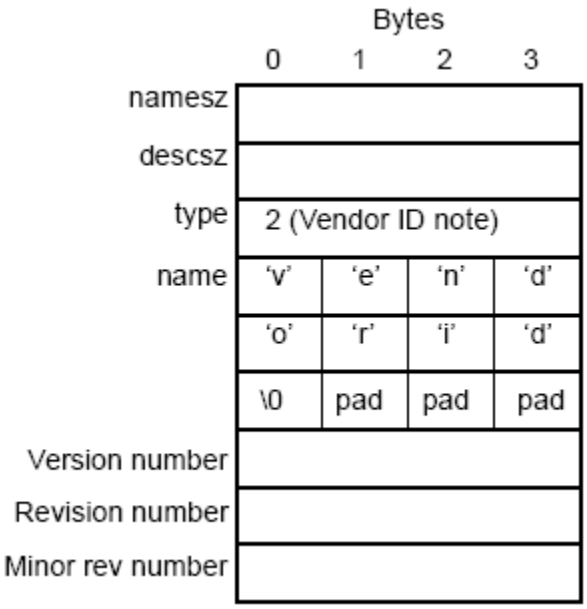

Figure 4-2. Vendor Identification Note Format

Object file comments generated by the user through an assembler directive are placed in the note section. This is typically for users to identify their object code. The same string termination and padding restrictions apply to object file comments as apply to vendor identification notes. The field contains a user-specified comment. A null comment (\0) is not a valid comment. The object file comment format is shown in the figure below.

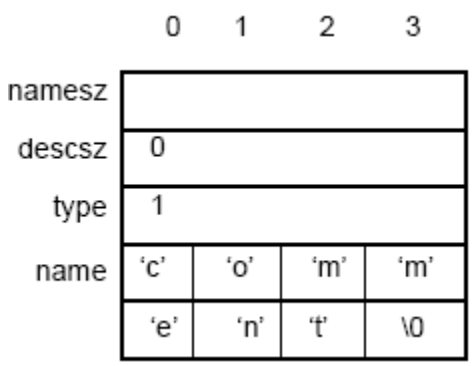

Figure 4-3. User (Application-Specific) Note Format

4.6 Program Headers

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

Refer the listing below for a sample program header.

Listing 4-6. Program Header

```
typedef struct {
  Elf64_Word    p_type;

  Elf64_Word    p_flags;

  Elf64_Off     p_offset;

  Elf64_Addr    p_vaddr;

  Elf64_Addr    p_paddr;

  Elf64_Xword   p_filesz;

  Elf64_Xword   p_memsz;

  Elf64_Xword   p_align;

} Elf64_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 types.
- `p_offset`: Offset from beginning of file to first byte of segment.
- `p_vaddr`: Virtual address in memory of the first byte of the segment.
- `p_paddr`: Physical address in memory of the first byte of the segment.
- `p_filesz`: Number of bytes in segment's file image. (May be zero.)
- `p_memsz`: Number of bytes in segment's memory image. (May be zero.)
- `p_flags`: Flags relevant to the segment. Defined flags are `PF_R`, `PF_W`, and `PF_X`.
- `p_align`: Segment alignment requirements in file and memory.

4.7 Debugging

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

This topic explains:

- DWARF Register Number Mapping

4.7.1 DWARF Register Number Mapping

The following table outlines the register number mapping for the StarCore processor cores.

Table 4-6. StarCore Register Number Mapping

Register Name	Number	Abbreviation
Stack Pointer	0	SP
General Data Registers	1-64	D0-D63
Address Registers	65-96	R0-R31
Data Registers-extension portion	97-160	D0_e-D63_e
Data Registers-high portion	161-224	D0_h-D63_h
Data Registers-low portion	225-288	D0_l-D63_l
Loop Counter Registers	289-292	LC0-LC3
Program Counter	293	PC
Modifier Control Register	294	MCTL
Back Trace Registers	295-296	BTR0-1
Implementation Dependent Configuration Register	297	IDCR
General Configuration Register	298	GCR

Chapter 5
Assembler Syntax and Directives

Assemblers must support the directives, special characters, and syntax identified in this chapter:

NOTE

For more information on StarCore Assembler, refer to *StarCore Assembler User Guide*.

This chapter explains:

- Assembler Significant Characters
- Assembler Directives
- Assembler Syntax
- Rule Checking

5.1 Assembler Significant Characters

This section list the assembler significant characters along with their meanings.

Several one- and two-character sequences are significant to the assembler and must be supported. Some have multiple meanings depending on context. These characters are listed in the table below.

Table 5-1. Assembler Significant Characters

Character	Meaning
;	Comment delimiter
;;	Unreported comment delimiter
\	Line continuation character or macro dummy argument concatenation operator
"	Quoted string DEFINE expansion character

Table continues on the next page...

Table 5-1. Assembler Significant Characters (continued)

Character	Meaning
@	Function delimiter
*	Location counter substitution
++	String concatenation operator
[ ]	Substring delimiter or instruction grouping delimiter
<<	I/O short addressing mode force operator
<	Short addressing mode force operator
>	Long addressing mode force operator
#	Immediate addressing mode operator
#<	Immediate short addressing mode force operator
#>	Immediate long addressing mode force operator

5.2 Assembler Directives

This section list the assembler directives that must be supported and their description.

The following table lists the assembler directives.

Table 5-2. Assembler Directives

Type	Directive	Description
Assembly Control	COMMENT	Start comment lines
	DEFINE	Define substitution string
	END	End of source program
	FAIL	Programmer generated error message
	HIMEM	Set high memory bounds
	INCLUDE	Include secondary file
	LOMEM	Set low memory bounds
	MSG	Programmer generated message
	ORG	Initialize memory space and location counters
	RADIX	Change input radix for constants
	UNDEF	Undefine DEFINE symbol
	WARN	Programmer generated warning
	ENDSEC	End section
	EQU	Equate symbol to a value
	GLOBAL	Global section symbol declaration
Symbol definition	GSET	Set global symbol to a value

Table continues on the next page...

Table 5-2. Assembler Directives (continued)

Type	Directive	Description
	SECFLAGS	Set ELF section flags
	SECTION	Start section
	SECTYPE	Set ELF section type
	SET	Set symbol to a value
	SIZE	Set size of symbol in the ELF symbol table
	TYPE	Set symbol type in the ELF symbol table
Data definition and storage allocation	ALIGN	Set address to modulo boundary
	BADDR	Set buffer address
	BSB	Block storage bit-reverse
	BSC	Block storage of constant
	BUFFER	Start buffer
	DC, DCW	Define constant (16-bits)
	DCB	Define constant byte (8-bits)
	DCL	Define constant long word (32-bits)
	DS	Define storage
	DSR	Define reverse carry storage
	ENDBUF	End buffer
	FALIGN	Align hardware loop
Conditional assembly	DUP	Duplicate sequence of source lines
	ENDIF	End of conditional assembly
	ENDM	End of duplicate sequence
	ELSE	Conditional assembly directive
	IF	Conditional assembly directive

5.3 Assembler Syntax

The symbol names, strings and statement format required in the assembler syntax are described in this section.

The following sections provide details on assembler syntax:

- Symbol Names
- Strings
- Source Statement Format

5.3.1 Symbol Names

This section lists the conventions followed by symbol names.

Following are the conventions:

- From 1 to 4000 characters long.
- Cannot begin with a number (0-9). Symbol names can otherwise be any combination of alphanumeric characters (A-Z, a-z, 0-9) and the underscore character (_).
- Symbol names and other identifiers containing a period (.) are legal but are reserved for the system.
- Upper and lower case letters in symbols are considered distinct.
- The upper or lower case names of StarCore processor core registers are reserved by the assembler and cannot be used.

The below table describes examples of symbol names.

Table 5-3. Symbol Name Examples

Valid Names	Invalid Names	Reserved Names

`loop_1`
`ENTRY`
`_a_B_c`
	`1_loop`	`loop.e`
		`.loop`

5.3.2 Strings

One or more ASCII characters enclosed by single quotes (') constitute a literal ASCII string. To specify an apostrophe within a literal string, two consecutive apostrophes must appear where the single apostrophe is intended.

Strings are used as operands for some assembler directives and also can be used to a limited extent in expressions.

A string can also be enclosed in double quotes ("), and any DEFINE directive symbols contained in the string would be expanded.

Two strings separated by the string concatenation operator (++) will be recognized by the assembler as equivalent to the concatenation of the two strings. For example, the following two strings are equivalent:

```
'ABC'++'DEF' = 'ABCDEF'
```

The assembler has a substring extraction capability using brackets ([]). Refer to the following example:

```
['abcdefg',1,3] = 'bcd'
```

Substrings can be used wherever strings are valid and can be nested. There are also functions for determining the length of a string and the position of one string within another.

5.3.3 Source Statement Format

This section explains the assembly language source statement format in detail.

As shown in the below figure, an assembly language source statement may include four fields in its most basic form: label, operation, operand, and comment.

Figure 5-1. Basic Source Statement

Fields must be separated by one or more spaces or tabs. Fields other than the comment field cannot contain embedded whitespace characters because these characters are used as field delimiters. An exception is spaces and tabs in quoted strings. Only fields preceding the comment field are considered significant to the assembler; the comment field is ignored. Anything beginning in column 1 is considered a label.

A source statement can be extended to multiple lines by including the line continuation character (\) as the last character on the line to be continued. An exception to this is instruction groups, which can span multiple lines as long as the instruction group is surrounded by brackets ([]). (See Instruction Groups) A source statement (first line and any continuation lines) can be a maximum of 4000 characters long. Upper and lower case letters are equivalent for assembler mnemonics and directives, but are distinct for labels, symbols, directive arguments, and literal strings.

Further in this topic:

- Instruction Groups

5.3.3.1 Instruction Groups

The StarCore architecture includes instruction groups that allow multiple instructions to execute in parallel. Rules governing how instructions can be grouped are discussed in each core's respective reference manual. The assembler interprets each line containing instructions as an instruction group. Instructions must be separated by tabs or spaces, as shown in the below listing.

Listing 5-1. Single-line instruction group

```
st.f (r2)+,d0 st.f (r2)+,d8 add.x d0,d1,d2 ; Instruction group with 3 instructions
```

When delimited with brackets ([]), an instruction group can span multiple lines, as shown in the following listing.

Listing 5-2. Multiple-line instruction group (SC1400)

```
[ mac.x d0.h,d1.h,d2
  mac.x d3.l,d4.h,d5

  add.x d0,d1,d3

]
```

5.3.3.2 Labels

Labels begin in column 1 of a source statement. A space or tab as the first character on a line ordinarily indicates that the label field is empty. Labels are subject to the following rules:

- Label names must follow the same conventions as symbol names.
- A label may be indented if it is immediately followed by a colon (:) with no intervening spaces. All characters preceding the label on the line must be whitespace characters-spaces or tabs.
- A label can occur only once in the label field of an individual source file unless it is used as a local label, a label local to a section, or is used with the SET directive. If a

non-local label occurs more than once in a label field, each reference to that label after the first is flagged as an error.
- A line consisting of a label only is valid and has the effect of assigning the value of the location counter to the label. With the exception of some directives, a label is assigned the value of the location counter of the first word of the instruction or data being assembled.

5.3.3.3 Operation Field

The operation field appears after the label field, and must be preceded by at least one space or tab. Entries in the operation field can be one of three types:

- Opcode: Mnemonics that correspond directly to DSP machine instructions.
- Directive: Special operation codes known to the assembler that control the assembly process.
- Macro call: Invocation of a previously defined macro to be inserted in place of the macro call.

5.3.3.4 Operand Field

The interpretation of the operand field depends on its contents. The operand field, if present, must follow the operation field, and must be preceded by at least one space or tab. The operand field can contain a symbol, an expression, or a combination of symbols and expressions separated by commas with no intervening spaces.

5.3.3.5 Comment Field

The assembler ignores comments, but they can be included in the source file for documentation purposes. A comment field is composed of any characters (not part of a literal string) that are preceded by a semicolon (;).

5.4 Rule Checking

Every core architecture has a set of programming rules to ensure correct code execution. Each core's reference manual defines the instructions that an assembly programmer or compiler uses.

The assembler and simulator ensure that the instructions are legally used.

To ensure ABI conformance, third-party assemblers and simulators must follow the requirements defined in the corresponding *SC3900 Core Reference Manuals*.

This documentation defines which rules must be validated statically by the assembler. In some cases, a rule can be dynamically validated only through the simulator. These cases are also documented.

Index

T

V

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number CWSCABIREF
Revision 10.9.0, 06/2015