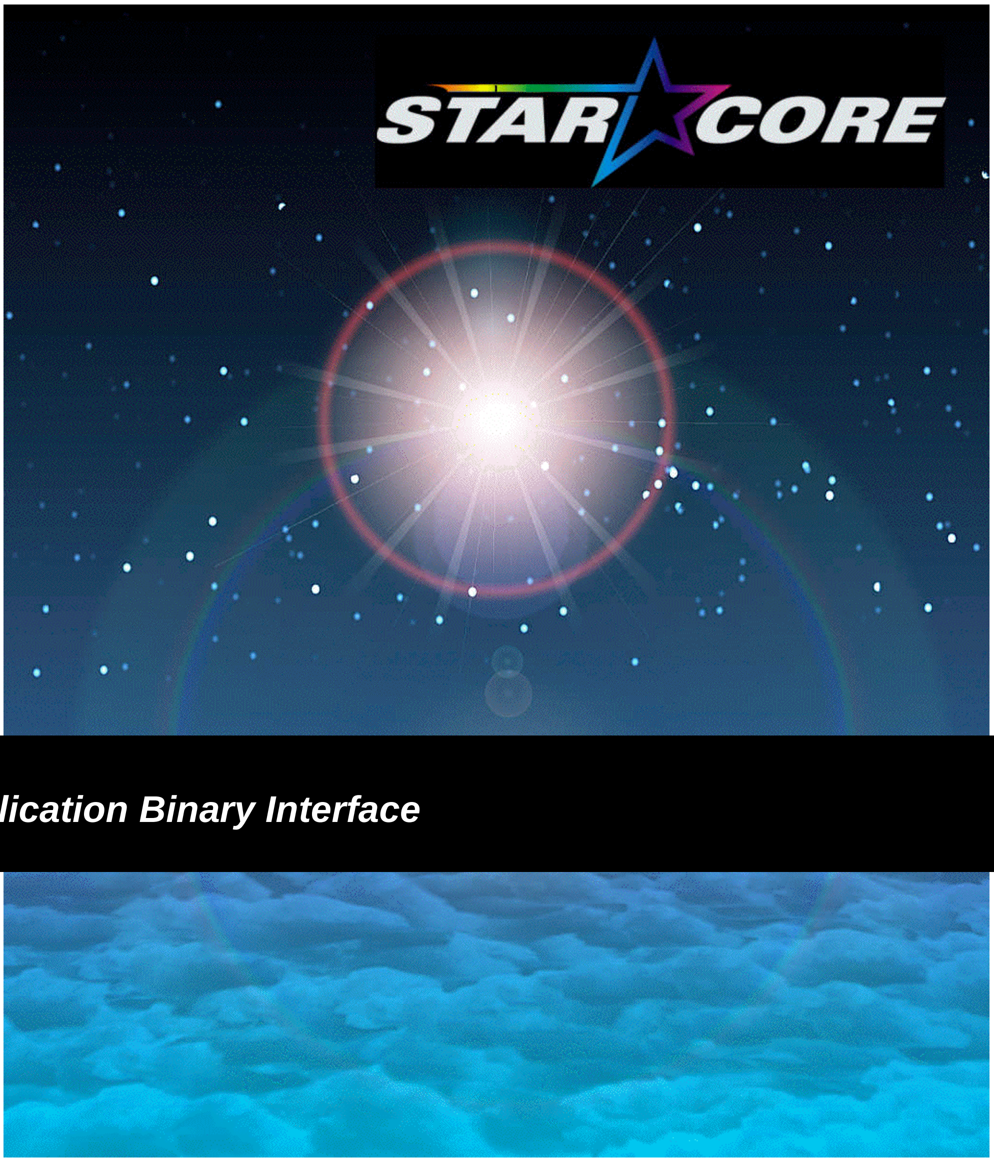
MNSC100ABI/D
Rev. 2.0, 06/2002



SC100 Application Binary Interface

MOTOROLA intelligence everywhere™ | digital dna™

agere systems

MNSC100ABI/D
Rev. 2.0, 06/2002

SC100 Application Binary Interface

Table of Contents

List of Tables

List of Figures

List of Examples

Chapter 1 Introduction

The SC100 application binary interface (ABI) defines a set of standards intended to ensure interoperability between conforming 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 the StarCore Technology Center, Agere Systems, Motorola, and third party tools developers.

A benefit of this standard definition is interoperability of conforming tools. This allows users to 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.

1.1 Overview

This ABI 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 are 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)

1.2 Conformance

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

1.3 References

The following standards provide useful reference information:

- *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*, Version 1.1, UNIX Systems Laboratories, Portable Formats Specification, 1995
- *DWARF Debugging Information Format*, Revision: Version 2.0.0, Industry Review Draft, UNIX International, Program Languages SIG, July 27, 1993
- *ANSI/IEEE Std 754-1985*, IEEE standard for binary floating-point arithmetic data types
- *ISO/IEC 9899:1999(E), International Standard - Programming Languages—C*, 2nd Edition, International Organization for Standardization, December 1, 1999

The following StarCore documents are included by reference into this ABI. With the exception of the design specification listed below, these documents are available through the StarCore web site at http://www.starcore-dsp.com.

- *SC100 Assembly Language Tools User's Manual* (MNSC100ALT/D)

  Describes the SC100 assembler syntax and directives listed in Chapter 5 of this ABI.

- *SC110 DSP Core Reference Manual* (MNSC110CORE/D)

  Describes the SC110 core architecture and programming model, including the SC110 instruction set.

- *SC140 DSP Core Reference Manual* (MNSC140CORE/D)

  Describes the SC140 core architecture and programming model, including the SC140 instruction set.

- *Support in the Assembler and Simulator Required for Correct Reporting of SC100 Restrictions* (design specification)

  Defines which instruction set programming rules must be validated by the assembler and simulator, and specifies the identifier that must be included in the error or warning message that is generated when a given rule is violated. This document is an internal design specification that is available to third parties under a non-disclosure agreement with the StarCore Technology Center.

The SC100 generation of core architectures currently includes two cores: the StarCore SC110 and the StarCore SC140. As future cores become available, their respective core reference manuals should also be considered part of this ABI.

1.4 Revision History

This Rev. 2.0 of the ABI supersedes the previous edition, Rev. 1.8, dated 04/2000. Major changes from the previous edition include:

Chapter 2, "Low-Level Binary Interface."

- Updated discussion of fundamental data types, aggregates, and bit fields, with little-endian and big-endian differences noted.
- Added sections on stack unwinding, register saving and restoring functions, function call modes, address modifier modes, saturation mode, and data addressing models.
- Removed the section, "Interrupt Handlers."
- Updated the calling conventions with these notable changes:
  - If the first argument is a `long long` (where implemented), `double`, or `long double`, it is passed in D0 and D1, as if it were first stored in an 8-byte aligned memory area and then the low-addressed word were loaded into D0 and the high-addressed word into D1.
  - Each argument on the stack is passed in the byte order appropriate for the endian mode.
  - A function with a variable number of arguments passes the last fixed argument and all subsequent variable arguments on the stack.
  - An argument that is 8-byte aligned is passed 8-byte aligned on the stack. All other arguments are passed 4-byte aligned on the stack.
  - Arguments are passed on the stack, in order, from higher addresses to lower addresses. Each argument on the stack is passed in the byte order appropriate for the endian mode.
  - A `long long`, `double`, or `long double` return value is returned in D0 and D1, as if it were first stored in an 8-byte aligned memory area and then the low-addressed word were loaded into D0 and the high-addressed word into D1.
  - A function returning a structure or union of any size receives in R2 the address of space in which to return the structure or union. The function does not return that address in R2.
  - The extension registers, D6.e and D7.e, are callee saved; the remaining extension registers are caller saved.
  - The MCTL register is caller saved.
  - A compiler assumes the rounding mode default is two's complement rounding, and the scaling mode default is no scaling.

Chapter 3, "High-Level Languages Issues."

- Added new C preprocessor predefines: `__SC110__`, `__SC140__`, `__LITTLE_ENDIAN__`, and `__BIG_ENDIAN__`.
- Removed requirement for support of C in-line assembly syntax.
- Changed names of existing floating-point routines and integer routines, and added `double` and `long long` routines. Also added descriptions of all routines.
- Added new section on intrinsics for accessing architectural features.

Chapter 4, "Object File Format."

- Updated the list of SC100 ELF sections.
- Added sections on SC100 special sections and debugging information.
- Replaced the relocation section with a new relocation scheme.

Removed original Chapter 5, “Endian Support.” This revision of the ABI incorporates endian information in individual sections, as appropriate, throughout the document.

Chapter 5, “Assembler Syntax and Directives” (originally Chapter 6 in Rev 1.8).

- Removed requirement for support of object file control directives, in addition to the individual directives MODE, DUPA, DUPC, DUPF, EXITM, MACLIB, MACRO, and PMACRO.
- Added requirement for support of ELSE and FALIGN directives.
- Added requirements for checking SC100 programming rules.

1.5 Acknowledgements

The SC100 Application Binary Interface team included representatives from the following companies:

Agere Systems Inc.	Metrowerks, Inc.
Altium Limited	Motorola, Inc.
Green Hills Software, Inc.	WindRiver Systems, Inc.
Lineo, Inc.	

We gratefully thank all participants for devoting their time and effort to create this standard.

Chapter 2
Low-Level Binary Interface

This chapter defines low-level system standards for the SC100 generation of DSP cores, including:

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

2.1 Core Architecture

The SC100 generation of core architectures currently includes three cores: the StarCore SC110, the StarCore SC140, and the StarCore SC140E. The architecture and instruction set for each core is defined in that core's respective reference manual, as listed in Section 1.3, "References." Programs written for these cores use their instruction sets, as well as the instruction encodings and semantics of their architecture. Programmers may assume that the instructions for these cores work as documented. Note that while an ABI-conforming SC110 program will run on an ABI-conforming SC140 processor, the reverse is not always true.

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-SC100 instructions or capabilities do not conform to the SC100 ABI. Such programs may produce unexpected results when run on machines lacking the non-SC100 capability.

2.2 Endian Support

The SC100 architecture supports both big-endian and little-endian implementations. This standard defines a binary interface for each. Note that program binaries that run on a big-endian implementation are not portable to a little-endian implementation, and vice versa. 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).

The bytes that form the supported data types are ordered in memory according to the following:

- In a big-endian implementation, the most significant byte (MSB) is located in the lowest address (byte 0).
- In a little-endian implementation, the least significant byte (LSB) is located in the lowest address (byte 0).

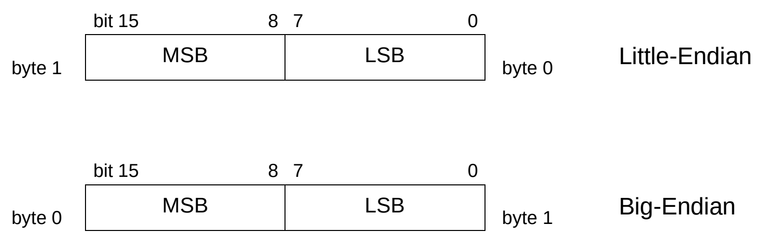
2.3 Fundamental Data Types

The SC100 architecture defines the following data types:

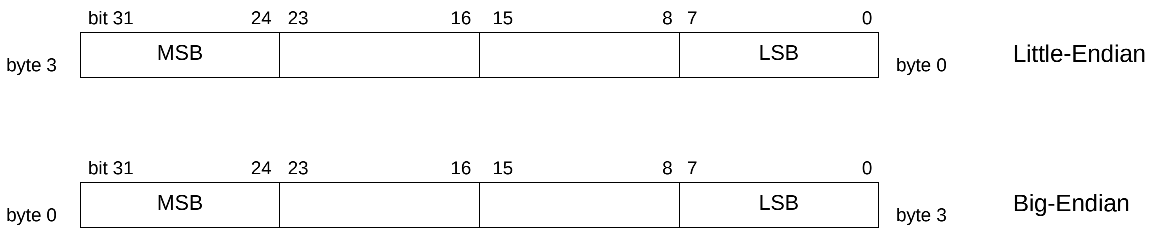
- An 8-bit byte
- A 16-bit word
- A 32-bit long word
- A 64-bit double-long word

The following examples illustrate the bit and byte numbering for these data types.

Example 2-1. Word Bit and Byte Numbering



Example 2-2. Long Word Bit and Byte Numbering



Example 2-3. Double-Long Word Bit and Byte Numbering

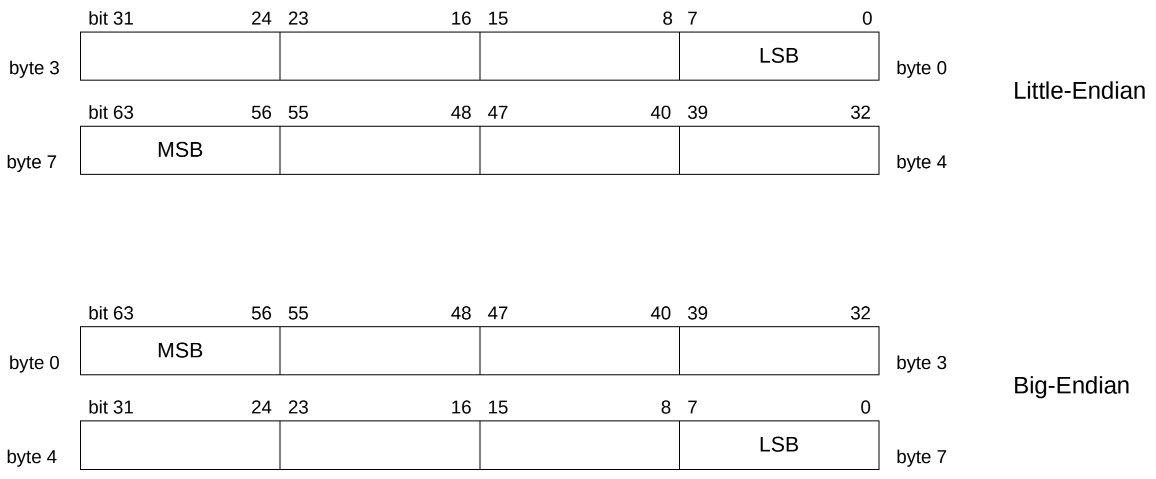


Table 2-1 shows the mapping between these fundamental data types and the C language data types. Note that fundamental data is always naturally aligned; that is, a double-long word is 8-byte aligned, a long word is 4-byte aligned, and a word is 2-byte aligned.

Table 2-1. Mapping of C Data Types to SC100

Type	C Type	Size (bits)	Align (bits)	Limits	SC100
	`_Bool`[1]	8	8	0 .. 1	signed byte

Character	`char`
	`signed char`
		8	8	-2^{7} .. $2^{7}-1$	signed byte

	`unsigned char`	8	8	0 .. $2^{8}-1$	unsigned byte

	`short`
	`signed short`
		16	16	-2^{15} .. $2^{15}-1$	signed word

	`unsigned short`	16	16	0 .. $2^{16}-1$	unsigned word

Integral	`int`
	`signed int`
	`enum`
	`long`
	`signed long`
		32	32	-2^{31} .. $2^{31}-1$	signed long word

	`unsigned int`
	`unsigned long`
		32	32	0 .. $2^{32}-1$	unsigned long word

	`long long`[1] `signed long long`[1]	64	64	-2^{63} .. $2^{63}-1$	signed double-long word
	`unsigned long long`[1]	64	64	0 .. $2^{64}-1$	unsigned double-long word

Pointer	`pointer to data`
	`pointer to function`
		32	32	0 .. $2^{32}-1$	unsigned long word

Floating[2] Point	`float`	32	32	$-3.402e^{38}$ .. $-1.175e^{-38}$ $1.175e^{-38}$ .. $3.402e^{38}$	unsigned long word

	`double`
	`long double`
		64	64	$-1.797e^{308}$ .. $-2.225e^{-308}$
				$2.225e^{-308}$ .. $1.797e^{308}$
					unsigned double-long word

Notes:

1. This data type is specified in the latest ISO C definition (ISO/IEC 9899:1999). Support of this data type is optional. If used, this data type must be implemented with the size and alignment shown.
2. Floating point types conform to the IEEE 754 format.

Fractional types are supported in C using intrinsic functions; Table 2-2 shows the fractional types that are supported.

Table 2-2. Mapping of C Fractional Types to SC100

C Type	C Type Definition	Size (bits)	Align (bits)	Limits
fractional	`short`	16	16	$-1 \ .. \ \frac{(2^{15}-1)}{2^{15}}$
long fractional	`long` or `int`	32	32	$-1 \ .. \ \frac{(2^{31}-1)}{2^{31}}$

long fractional with extension bits	Little-Endian:
	`typedef struct {`
	`   unsigned int body;`
	`   signed char ext;`
	`} word40;`
	
	Big-Endian:
	`typedef struct {`
	`   char pad[3];`
	`   signed char ext;`
	`  unsigned int body;`
	`} word40;`
		64	32	$-256 \ .. \ \frac{(2^{39}-1)}{2^{31}}$

double precision fractional	`typedef struct {`
	`   int  lsb;`
	`   int  msb;`
	`} word64;`
		64	32	$-1 \ .. \ \frac{(2^{63}-1)}{2^{63}}$

2.4 Aggregates and Unions

Aggregates (structures and arrays) and unions assume the alignment of their most strictly aligned member (that is, the member with the largest alignment). 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.

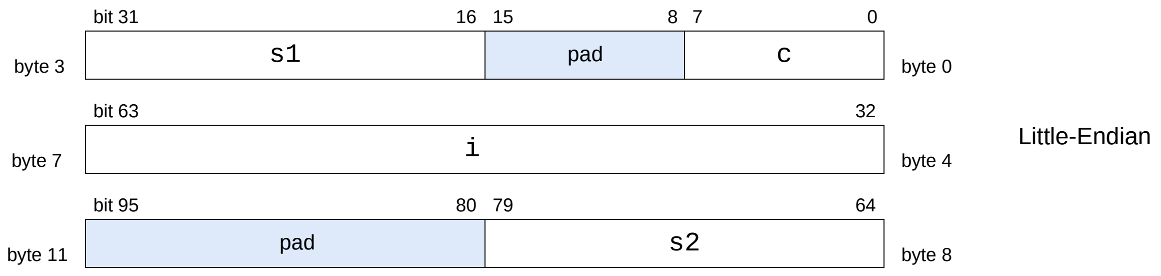
The size of any structure, array, or union must be an integral multiple of its alignment. Structure and unions may require padding to meet size and alignment constraints:

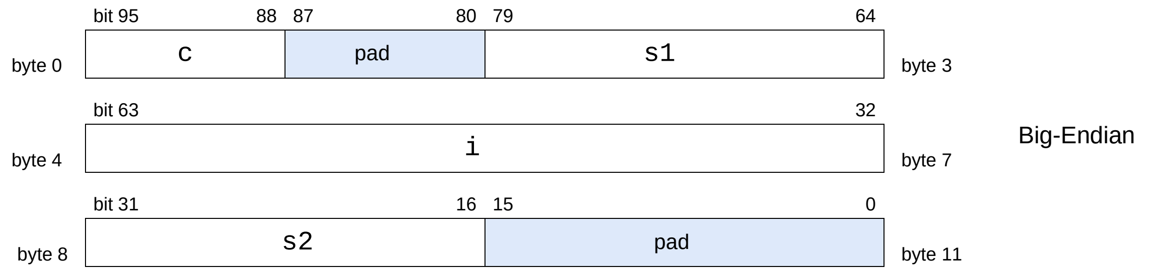
- An entire structure or union is aligned on the same boundary as its most strictly aligned member.
- Each member is allocated starting at the next byte that satisfies the alignment requirement for that member. This may require internal padding.
- If necessary, a structure's size is increased to make it a multiple of the structure's alignment. This may require tail padding, depending on the last member.

In both endian modes, members are allocated starting with the low order (lowest addressed) byte of the structure or union, as shown in the following examples. In Example 2-4, there is internal padding so that 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.

Example 2-4. Structure With Internal and Tail Padding

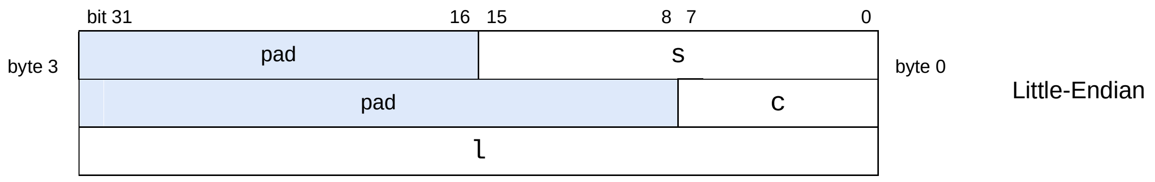
```
struct {                /* 12 bytes, 4-byte aligned */
       char   c;
       short  s1;
       int    i;
       short  s2;
};
```

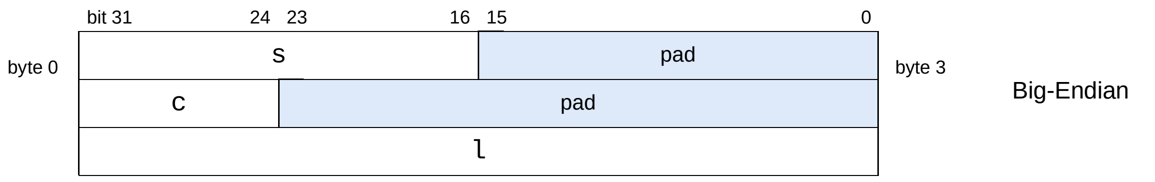




Example 2-5. `union` **Allocation**

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





2.5 Bit Fields

Structure and union definitions may have bit fields as listed in Table 2-3.

Table 2-3. C Bit Field Types

C Type	Maximum Width (bits)
`_Bool`[1] `char`[2] `signed char`[2] `unsigned char`[2]	1 to 8
`short`[2] `signed short`[2] `unsigned short`[2]	1 to 16

`int`
`signed int`
`enum`[2]
`long`[2]
`signed long`[2]
`unsigned int`
`unsigned long`[2]
	1 to 32

Notes:

1. Support of `_Bool` is optional. If implemented, it must be implemented with the width and range shown.
2. This bit field type is not required for ISO C conformance, but is required for ABI conformance.

Support of `_Bool` is optional, but all other types shown in Table 2-3 must be supported. 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 the bit field's width in bits. Signed bit-field values range from -2^{w-1} to $2^{w-1}-1$.

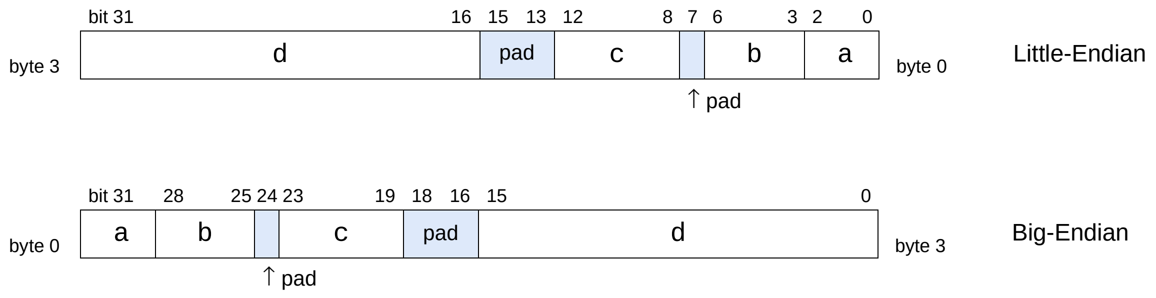
A "plain" bit field (one that is not explicitly declared signed or unsigned) is signed. Although they may have type `char`, `short`, `int`, or `long` (which can have negative values), bit fields of these types have the same range as bit fields of the same size with the corresponding signed type. The 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:

- In little-endian implementations, bit fields are allocated right to left. The first bit field occupies the least significant bits while subsequent bit fields occupy more significant bits.
- In big-endian implementations, bit fields are allocated left to right. The first bit field occupies the most significant bits 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 eight bits in width, and it 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.

Note in the following examples that alignments are driven not by the widths of the bit fields but by the underlying types. Example 2-6 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.

Example 2-6. Bit Field Alignment and Padding

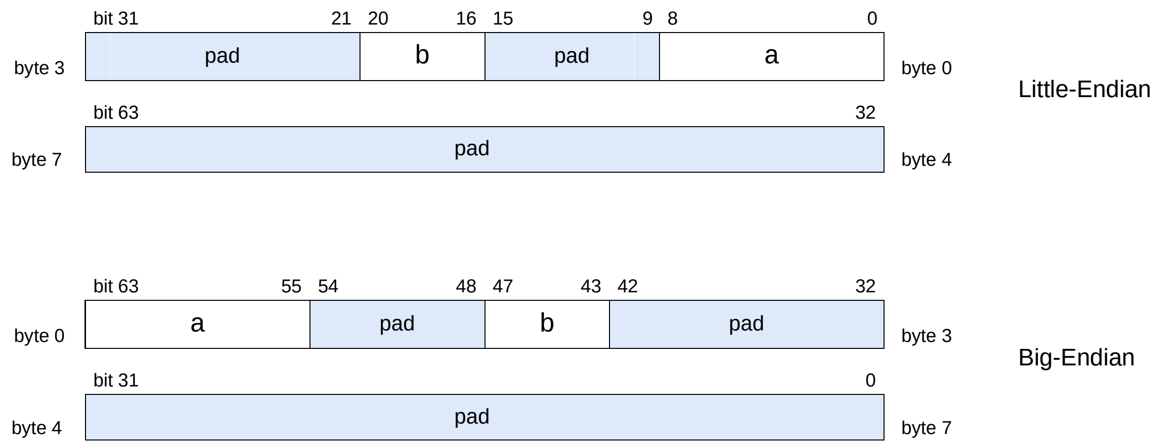
```
struct {                    /* 4 bytes, 4-byte aligned */
       int    a  :  3;
       int    b  :  4;
       char   c  :  5;
       short  d;
};
```



In Example 2-7, the structure is 2-byte aligned because the unnamed `long` bit field does not affect structure alignment. The zero-width `short` bit field pads to the next word boundary.

Example 2-7. Unnamed and Zero-Width Bit Fields

```
struct {                    /* 8 bytes, 2-byte aligned */
       short  a  :  9;
       short     :  0;
       char   b  :  5;
       long      :  15;
};
```



2.6 Function Calling Sequence

Compilers must support the conventions described in this section.

2.6.1 Argument Passing and Return Values

The following calling conventions must be supported.

- If the first function argument is 4 or fewer bytes and is an integral type, floating type, structure, or union, the argument is passed in D0. If it is a pointer, it is passed in R0.
- If the second argument is 4 or fewer bytes and is an integral type, floating type, structure, or union, the argument is passed in D1. If it is a pointer, it is passed in R1.
- When an argument is passed in D0 or D1, only the lower order register bytes that constitute the argument are defined. For example, a first argument of type short is passed in D0[15:0], and the contents of D0[31:16] and D0.e are undefined.
- If the first argument is a `long long` (where implemented), `double`, or `long double`, it is passed in D0 and D1, as if it were first stored in an 8-byte aligned memory area and then the low-addressed word were loaded into D0 and the high-addressed word into D1. This means that D0 contains the most significant long word in big-endian and the least significant long word in little-endian.
- Functions with a variable number of arguments pass the last fixed argument and all subsequent variable arguments on the stack. The rules above apply to arguments before the last fixed argument.
- All other arguments are passed on the stack. Note that the first argument may be passed on the stack, followed by the second argument being passed in D1 or R1.
- Arguments are passed on the stack, in order, from higher addresses to lower addresses. Each argument on the stack is passed in the byte order appropriate for the endian mode.
- An argument that is 8-byte aligned according to Section 2.3, “Fundamental Data Types,” Section 2.4, “Aggregates and Unions,” and Section 2.5, “Bit Fields,” is passed 8-byte aligned on the stack. All other arguments are passed 4-byte aligned on the stack.
- The constituent bytes of an integral argument of fewer than 4 bytes are located on the stack as if the argument had been promoted to 32 bits, although the caller might not sign or zero extend the argument. Thus, in little-endian, those arguments are placed in the lower addressed bytes within their 4-byte memory blocks, and in big-endian they are placed in the higher addressed bytes.
- An integral return value, other than a `long long`, is sign or zero extended to 40 bits and returned in D0. A float value is returned in D0. A `long long`, `double`, or `long double` return value is returned in D0 and D1, as if it were first stored in an 8-byte aligned memory area and then the low-addressed word were loaded into D0 and the high-addressed word into D1.
- A pointer return value is returned in R0.
- A function returning a structure or union receives in R2 the address of the returned structure or union. The caller allocates space for the returned object.
- Registers will be saved as shown in Table 2-4.
- Compilers will make the following assumptions about operating control bits:
  - Rounding mode default is 1 (SR[3]=1), which means two’s complement rounding.
  - Scaling mode bits default is 0 (SR[4,5]=[00]), which means no scaling.

  Setting these mode bits is the application’s responsibility.

Example 2-8 shows two function calls and the arguments that are allocated for each call.

Example 2-8. Function Calls and Allocation of Arguments

Function Call 1:

```
foo(int a1, struct fourbytes a2, struct eightbytes a3, short a4)
```

Arguments:

```
a1 - in register d0
a2 - in register d1
a3 - on the stack at SP (stack pointer address)
a4 - on the stack at SP - 8 (little-endian) or
     SP - 10 (big-endian)
```

Function Call 2:

```
bar(long *b1, int b2, char b3, int b4[])
```

Arguments:

```
b1 - in r0
b2 - in d1
b3 - on stack at SP (little-endian) or
     SP - 3 (big-endian)
b4 – on stack at SP - 4
```

Table 2-4 summarizes register usage in the calling convention.

Table 2-4. Register Usage in the Calling Convention

Register	Caller Saved	Callee Saved	Used As
D0	+		First numeric argument Return numeric value
D1	+		Second numeric argument
D2–D5	+		
D6–D7		+	
D8–D15	+		
D0.e–D5.e	+		
D6.e–D7.e		+	
D8.e–D15.e	+		
R0	+		First pointer argument Return pointer value
R1	+		Second pointer argument
R2	+		Structure or union return address

Table 2-4. Register Usage in the Calling Convention (Continued)

Register	Caller Saved	Callee Saved	Used As
R3–R5	+		
R6		+	Global offset pointer, used for PIC and PID
R7		+	Optional frame pointer
R8–R15, B0–B7	+		
N0–N3, M0–M3	+		
MCTL	+		
SP (NSP, ESP)		+	
SA0–SA3	+		
LC0–LC3	+		

2.6.2 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. On the SC100 and other architectures, C programs intended to be portable use the header files `<stdarg.h>` or `<varargs.h>` to deal with variable argument lists.

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 mark (...). However, compiler vendors are expected to provide options for non-ANSI programs to allow them to declare variable argument functions in the command line or to treat all non-prototyped functions as (potentially) having variable argument lists.

2.6.3 Stack

The SP register serves as the stack pointer. SP will point to the first available location, with the stack direction being towards higher addresses (i.e., a push will be 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.

2.6.4 Stack Frame Layout

The stack pointer points to the top (high address) of the stack frame. Space at higher addresses than the stack pointer is considered invalid and may actually be unaddressable. The stack pointer value must always be a multiple of eight.

Figure 2-1 shows typical stack frames 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 with the local variables. This space is typically allocated only in functions that make calls returning structures.

A "return address" value of 0xffffffff (-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.

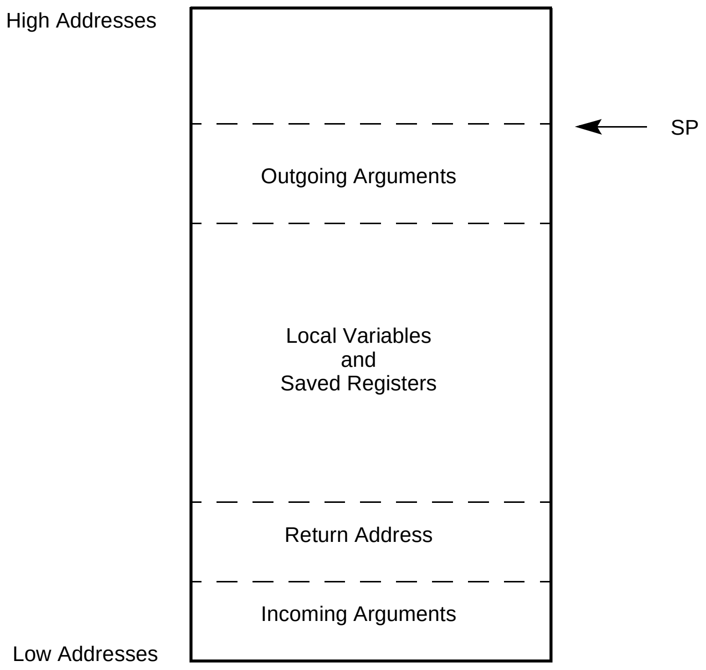


Figure 2-1. Stack Frame Layout

2.6.5 Stack Unwinding

The compiler will create special symbols when a module is compiled without debug enabled (e.g., the `-g` compiler option is not used). These symbols will appear as local symbols in the `.symtab` ELF section and will 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 outlined in Section 5.3.1, “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 shown below.

```
Value   Size Binding Type    Section  Name
-----   ---- ------- ----    -------  ----
0x10120    0 LOCAL   NOTYPE  .text    TextStart_hello
0x0        0 LOCAL   NOTYPE  ABS      StackOffset__main
0x2        0 LOCAL   NOTYPE  ABS      StackOffset_DW_2
0x0        0 LOCAL   NOTYPE  ABS      StackOffset_DW_5
0x1012a    0 LOCAL   NOTYPE  .text    DW_2
0x10136    0 LOCAL   NOTYPE  .text    DW_5
0x10138    0 LOCAL   NOTYPE  .text    FuncEnd_main
0x10138    0 LOCAL   NOTYPE  .text    TextEnd_hello
```

In this example, 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`.

Example 2-9 illustrates how these symbols might be defined in an assembly language program.

Example 2-9. Generating Stack Unwinding Symbols in Assembly Code

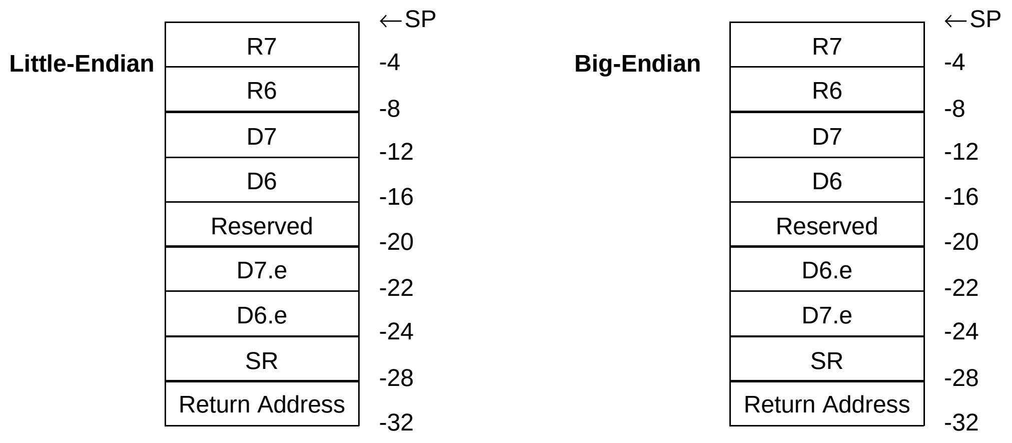
```
    section .text local

TextStart_hello

;*************************************************************
; Example function _main
;*************************************************************

    global _main
_main type func
 [
    push r6
    push r7
 ]
DW_2
    . . .
 [
    pop r6
    pop r7
 ]
DW_5
    rts

FuncEnd__main

StackOffset__main        equ   0     ; at _main sp = 0 words
StackOffset_DW_2         equ   2     ; at DW_2 sp = 2 words
StackOffset_DW_5         equ   0     ; at DW_5 sp = 0 words

TextEnd_hello
    endsec
```

2.6.6 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 and the SR. These functions are provided to save and restore these registers with a minimal increase in static code size. The functions use nonstandard calling conventions which 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 `___Qabi_callee_save`, the stack frame values relative to the address in the stack pointer (SP) will be:



The restoring function `___Qabi_callee_restore` assumes the stack frame layout above. It restores the callee-saved registers and returns through the caller return address stored at SP-32. There is no need for an RTS after calling the restoring function, since it returns automatically for the caller.

Example 2-10 shows an example use of the saving and restoring functions. The functions do not modify any caller-saved registers.

Example 2-10. 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
```

2.6.7 setjmp and longjmp Layout

The layout for the `jmp_buf` used by setjmp and longjmp follows. This layout preserves the callee-saved registers, which is needed to restore the state when longjmp is called.

```
typedef int jmp_buf[7];
```

Little-Endian

Offset	Saved Register
+ 0	D6
+ 4	D7
+ 8	R6
+ 12	R7
+ 16	D6.e
+ 18	D7.e
+ 20	SP
+ 24	Return Address

Big-Endian

Offset	Saved Register
+ 0	D6
+ 4	D7
+ 8	R6
+ 12	R7
+ 16	D7.e
+ 18	D6.e
+ 20	SP
+ 24	Return Address

2.6.8 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 or a global pointer is necessary, a compiler may allocate R7 as a frame pointer and R6 as a global pointer. When these registers are allocated for this purpose, they should be saved and restored as part of the function prologue/epilog code.

2.6.9 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.

2.6.10 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 Function Call Modes

Compilers must support the following pragma directives to control how external functions are called. The directives affect all functions declared after the pragma. If the compiler encounters inconsistent pragma directives for a given function, it will generate a warning and use the information from the original directive.

```
#pragma starcore callmode=near
#pragma starcore callmode=far
#pragma starcore callmode=default
```

If the callmode is far, the compiler will generate a 32-bit absolute call. If the callmode is near, the compiler will generate a 20-bit PC-relative call. If a function is out of range at link time, the linker will generate an error. The default callmode is determined by compiler options.

2.8 Address Modifier Modes

Compilers will make the following assumptions about address modifier modes:

- 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.

2.9 Saturation Mode

Compilers shall be able to set arithmetic saturation mode on or off using a compiler command line option, and they shall document their default saturation mode settings. Compilers need not emit the same code when saturation mode is off as they emit when the mode is on.

Compilers must support the saturation mode intrinsics as described in Table 3-8.

2.10 Data Addressing Models

A Zero Data Area (ZDA) has special data sections located near zero, allowing the compiler to more effectively use the 16-bit absolute addressing mode. The sections, `.zdata` and `.zbss`, need to be located in the low 16-bits of address space. The compiler supports directives to place data in the zero data area, and knows to use the more efficient addressing modes to access it. If more data is placed in ZDA than can fit, the linker will generate errors.

By default, data is placed in the standard data areas. Compilers will support an option that allows a coarse level of control, in which the user has the option of allocating all data to ZDA or allocating only those data items of a specified size.

The following pragma directives allow a finer level of control:

```
#pragma starcore startzda
#pragma starcore endzda
#pragma starcore startdata
#pragma starcore enddata
```

Any data declared between the `startzda` and `endzda` directives will be placed in ZDA. The corresponding `startdata` and `enddata` directives force data into the standard data section even if the zero data compiler option is specified.

Compilers must support both unsigned 16-bit, signed 16-bit, and signed 32-bit addresses. If the application is small enough to allow all static data to fit into the lower 64K or 32K of the address space, then more efficient code can be generated. The big memory model does not restrict the amount of space allocated to addresses; this model is the default. The small memory model assumes that all addresses are within the address range of an unsigned 16-bit immediate. The tiny memory model assumes that all addresses are within the range of a signed 16-bit immediate (effectively an unsigned 15-bit range).

These three compilation models are provided to allow the compiler to generate references to global and static data without global knowledge as to the variables' final allocation address in memory. For each model, the compiler will assume that references to global and static data fit within the corresponding size implied by the model. The expectation is that the linker will generate errors whenever a symbolic reference is resolved to not fit within the range defined by the memory model.

When the compiler uses the big memory model to access a data object, whether static or global, it must use a longer instruction that includes a 32-bit address. This operation requires an additional word, and as a result it produces code that is larger, and in some cases, slower, than a similar operation using the small or tiny memory models.

Example 2-11 illustrates the code sequence to generate the address of a global symbol in memory and the sequence to reference the memory contents of a global symbol for each memory model.

Example 2-11. Memory Models

```
;;Big Memory Model
    move.l address,d0    (3 16-bit words)
    moveu.l #address,d0  (3 16-bit words)

;;Small Memory Model
    move.l <address,d0   (2 16-bit words)
    moveu.l #address,d0  (3 16-bit words)

;;Tiny Memory Model
    move.l <address,d0   (2 16-bit words)
    move.w #address,d0   (2 16-bit words)
```

Certain instructions can be used only in small and tiny memory models. If < is omitted in conjunction with these instructions, an error results. Example 2-12 shows the instruction BMSET.W, which sets bit 0 in the specified address, and is valid only in small and tiny memory models.

Example 2-12. Small and Tiny Memory Mode Instruction

```
    bmset.w #0001,<address
```

Chapter 3
High-Level Languages Issues

3.1 C Preprocessor Predefines

All C/C++ language compilers must have the predefined macros as in Table 3-1, in addition to the predefined macros required by the C and C++ language standards.

As future cores become available, their predefined macros will be noted in the document, *SC100 Application Binary Interface Supplement*. This supplement will be available through the StarCore web site at http://www.starcore-dsp.com.

Table 3-1. Predefined Macros

Macro	Description
__SC100__	Defined for use with all compilers based on the SC100 architecture
__SC110__	The architecture variant which specifies that one MAC unit is to be used by the compiler
__SC140__	The architecture variant which specifies that four MAC units are to be used by the compiler
__LITTLE_ENDIAN__	Defined for use in little-endian mode
__BIG_ENDIAN__	Defined for use in big-endian 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 following:

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

generates assembly code similar to the following fragment:

```
_testfunc:
      rts
```

3.3 C System Calls

There are several typedefs specified in POSIX.1 which are required for system call wrappers. These types are defined as follows for the SC100 architecture:

```
typedef unsigned int mode_t;
typedef long int off_t;
typedef unsigned int size_t;
typedef int ssize_t;
typedef long int clock_t;
typedef long int time_t;
```

The following system calls must also be supported:

```
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 *);
```

3.4 Fractional Arithmetic Support

Fractional arithmetic is supported through the intrinsic functions listed in Table 3-2. Compilers must recognize the function names as shown with the double underscore (__) prefix. A header file may be provided that maps the unprefixed function names to the prefixed names.

The file `abi_intrinsics.c` contains a reference implementation in C of the intrinsics listed in Table 3-2. This reference implementation is in accordance with the ITU/ETSI definition of these functions. The `abi_intrinsics.c` file will be available through StarCore's documentation web site at http://www.starcore-dsp.com.

Table 3-2. Required Intrinsics for Fractional Types

Intrinsic Function	Description
Fractional Arithmetic:	
`short __add(short,short)`	Short add
`short __sub(short,short)`	Short sub
`short __mult(short,short)`	Short multiplication
`short __div_s(short,short)`	Short div
`short __mult_r(short,short)`	Multiply with round

Table 3-2. Required Intrinsics for Fractional Types (Continued)

Intrinsic Function	Description
Fractional Arithmetic (continued):	
`long __L_mac(long,short,short)`	Multiply accumulate
`long __L_macNs(long,short,short)`	Multiply accumulate with no saturation
`short __mac_r(long,short,short)`	Multiply accumulate with round
`long __L_msu(long,short,short)`	Multiply subtract
`long __L_msuNs(long,short,short)`	Multiply subtract with no saturation
`short __msu_r(long,short,short)`	Multiply subtract with round
`short __abs_s(short)`	Short abs
`short __negate(short)`	Short negate
`short __round(long)`	Round
`short __shl(short,short)`	Short shift left
`short __shr(short,short)`	Short shift right
`short __shr_r(short,short)`	Short shift right with round
`short __norm_s(short)`	Normalize any fractional value
Long Fractional Arithmetic:	
`long __L_add(long,long)`	Long add
`long __L_sub(long,long)`	Long subtract
`long __L_mult(short,short)`	Long multiplication
`short __extract_h(long)`	Extract high
`short __extract_l(long)`	Extract low
`long __L_deposit_h(short)`	Deposit short in MSB
`long __L_deposit_l(short)`	Deposit short in LSB
`long __L_abs(long)`	Long abs
`long __L_negate(long)`	Long negate
`short __norm_l(long)`	Normalize any long fractional value
`long __L_shl(long,short)`	Long shift left
`long __L_shr(long,short)`	Long shift right
`long __L_shr_r(long,short)`	Long shift right with round
`long __L_sat(long)`	Long saturation

3.5 Libraries

The following sections provide details on support libraries.

3.5.1 Compiler Assist Libraries

The SC100 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.

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

- Floating-point math routines
- Integer divide routines
- Integer modulo 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 Table 3-3, Table 3-4, and Table 3-5.

Routines in support libraries must satisfy the following constraints:

- The only external state information used is floating-point operation mode (rounding mode, flush to zero, etc.).
- No other global state can be modified.
- Identical results must be returned when a routine is reinvoked 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, previously dereferenced pointers need not be referenced again.

3.5.2 Floating-Point Routines

Conformant library support must include the floating point routines listed in Table 3-3 (the routine interfaces are shown as C function prototypes). These floating point routines must comply with the calling conventions described in Section 2.6, “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 3-3. Floating-Point Routines

`_fp_round`						
`_d_add`	`_d_fgt`	`_f_add`	`_f_ftod`	`_q_add`	`_q_fne`	`_q_utoq`
`_d_cmp`	`_d_fle`	`_f_cmp`	`_f_ftoi`	`_q_cmp`	`_q_itoq`	
`_d_cmpe`	`_d_flt`	`_f_cmpe`	`_f_ftoq`	`_q_cmpe`	`_q_mul`	
`_d_div`	`_d_fne`	`_f_div`	`_f_ftou`	`_q_div`	`_q_neg`	
`_d_dtof`	`_d_itod`	`_f_feq`	`_f_itof`	`_q_dtoq`	`_q_qtod`	
`_d_dtoi`	`_d_mul`	`_f_fge`	`_f_mul`	`_q_feq`	`_q_qtoi`	
`_d_dtoq`	`_d_neg`	`_f_fgt`	`_f_neg`	`_q_fge`	`_q_qtos`	
`_d_dtou`	`_d_qtod`	`_f_fle`	`_f_qtof`	`_q_fgt`	`_q_qtou`	
`_d_feq`	`_d_sub`	`_f_flt`	`_f_sub`	`_q_fle`	`_q_stoq`	
`_d_fge`	`_d_utod`	`_f_fne`	`_f_utof`	`_q_flt`	`_q_sub`	

`int _fp_round(int rounding_mode)`

Sets the rounding mode for floating point library routines. If rounding mode is:

- -1, then return the current rounding mode without setting it (this is required for conformance)
- 0, then request round to nearest (this is required for conformance)
- 1, then request round toward 0 (optional)
- 2, then request round toward positive infinity (optional)
- 3, then request round toward negative infinity (optional)

This function returns the resulting rounding mode (0–3), which will be `rounding_mode` if that rounding mode is supported by the floating point routines.

`double _d_add(double a, double b)`

Returns `a+b`, computed to double precision.

`int _d_cmp(double a, double b)`

Performs an unordered comparison of the double precision values of `a` and `b`, and returns an integer value, as follows, that indicates their relative ordering:

Relation	Value
`a` equal to `b`	0
`a` less than `b`	1
`a` greater than `b`	2
`a` unordered with respect to `b`	3

`int _d_cmpe(double a, double b)`

Performs an ordered comparison of the double precision values of `a` and `b`, and returns an integer value, as follows, that indicates their relative ordering:

Relation	Value
`a` equal to `b`	0
`a` less than `b`	1
`a` greater than `b`	2

`double _d_div(double a, double b)`
: Returns `a/b`, computed to double precision.

`float _d_dtof(double a)`
: Converts the double precision value of `a` to single precision, and returns the single precision value.

`int _d_dtoi(double a)`
: Converts the double precision value of `a` to a signed integer by truncating any fractional part, and returns the signed integer value.

`long double _d_dtoq(double a)`
: Converts the double precision value of `a` to extended precision, and returns the extended precision value.

`unsigned int _d_dtou(double a)`
: Converts the double precision value of `a` to an unsigned integer by truncating any fractional part, and returns the unsigned integer value.

`int _d_feq(double a, double b)`
: Performs an unordered comparison of the double precision values of `a` and `b`. Returns a 1 if they are equal, and a 0 otherwise.

`int _d_fge(double a, double b)`
: Performs an ordered comparison of the double precision values of `a` and `b`. Returns a 1 if `a` is greater than or equal to `b`, and a 0 otherwise.

`int _d_fgt(double a, double b)`
: Performs an ordered comparison of the double precision values of `a` and `b`. Returns a 1 if `a` is greater than `b`, and a 0 otherwise.

`int _d_fle(double a, double b)`
: Performs an ordered comparison of the double precision values of `a` and `b`. Returns a 1 if `a` is less than or equal to `b`, and a 0 otherwise.

`int _d_flt(double a, double b)`
: Performs an ordered comparison of the double precision values of `a` and `b`. Returns a 1 if `a` is less than `b`, and a 0 otherwise.

`int _d_fne(double a, double b)`
: Performs an unordered comparison of the double precision values of `a` and `b`. Returns a 1 if they are unordered or not equal; returns a 0 otherwise.

`double _d_itod(int a)`
: Converts the signed integer value of `a` to double precision, and returns the double precision value.

`double _d_mul(double a, double b)`
: Returns `a*b`, computed to double precision.

`double _d_neg(double a)`
: Returns `-a`.

`double _d_qtod(const long double *a)`
: Converts the extended precision value of `a` to double precision, and returns the double precision value.

`double _d_sub(double a, double b)`
: Returns `a-b`, computed to double precision.

`double _d_utod(unsigned int a)`

Converts the unsigned integer value of a to double precision, and returns the double precision value.

`float _f_add(float a, float b)`

Returns a+b, computed to single precision.

`int _f_cmp(float a, float b)`

Performs an unordered comparison of the single precision values of a and b, and returns an integer value, as follows, that indicates their relative ordering:

Relation	Value
a equal to b	0
a less than b	1
a greater than b	2
a unordered with respect to b	3

`int _f_cmpe(float a, float b)`

Performs an ordered comparison of the single precision values of a and b, and returns an integer value, as follows, that indicates their relative ordering:

Relation	Value
a equal to b	0
a less than b	1
a greater than b	2

`float _f_div(float a, float b)`

Returns a/b, computed to single precision.

`int _f_feq(float a, float b)`

Performs an unordered comparison of the single precision values of a and b. Returns a 1 if they are equal, and a 0 otherwise.

`int _f_fge(float a, float b)`

Performs an ordered comparison of the single precision values of a and b. Returns a 1 if a is greater than or equal to b, and a 0 otherwise.

`int _f_fgt(float a, float b)`

Performs an ordered comparison of the single precision values of a and b. Returns a 1 if a is greater than b, and a 0 otherwise.

`int _f_fle(float a, float b)`

Performs an ordered comparison of the single precision values of a and b. Returns a 1 if a is less than or equal to b, and a 0 otherwise.

`int _f_flt(float a, float b)`

Performs an ordered comparison of the single precision values of a and b. Returns a 1 if a is less than b, and a 0 otherwise.

`int _f_fne(float a, float b)`

Performs an unordered comparison of the single precision values of a and b. Returns a 1 if they are unordered or not equal; returns a 0 otherwise.

`double _f_ftod(float a)`
Converts the single precision value of a to double precision, and returns the double precision value.

`int _f_ftoi(float a)`
Converts the single precision value of a to a signed integer by truncating any fractional part, and returns the signed integer value.

`long double _f_ftoq(float a)`
Converts the single precision value of a to extended precision, and returns the extended precision value.

`unsigned int _f_ftou(float a)`
Converts the single precision value of a to an unsigned integer by truncating any fractional part, and returns the unsigned integer value.

`float _f_itof(int a)`
Converts the signed integer value of a to single precision, and returns the single precision value.

`float _f_mul(float a, float b)`
Returns `a*b`, computed to single precision.

`float _f_neg(float a)`
Returns `-a`.

`float _f_sub(float a, float b)`
Returns `a-b`, computed to single precision.

`float _f_utof(unsigned int a)`
Converts the unsigned integer value of a to single precision, and returns the single precision value.

`long double _q_add (const long double *a, const long double *b)`
Returns a+b, computed to extended precision.

`int _q_cmp(const long double *a, const long double *b)`
Performs an unordered comparison of the extended precision values of a and b, and returns an integer value, as follows, that indicates their relative ordering:

Relation	Value
a equal to b	0
a less than b	1
a greater than b	2
a unordered with respect to b	3

`int _q_cmpe(const long double *a, const long double *b)`
Performs an ordered comparison of the extended precision values of a and b, and returns an integer value, as follows, that indicates their relative ordering:

Relation	Value
a equal to b	0
a less than b	1
a greater than b	2

`long double _q_div(const long double *a, const long double *b)`
: Returns `a/b`, computed to extended precision.

`long double _q_dtoq(double a)`
: Converts the double precision value of `a` to quadruple precision, and returns the extended precision value.

`int _q_feq(const long double *a, const long double *b)`
: Performs an unordered comparison of the extended precision values of `a` and `b`. Returns a nonzero value if they are equal, and a 0 otherwise.

`int _q_fge(const long double *a, const long double *b)`
: Performs an ordered comparison of the extended precision values of `a` and `b`. Returns a nonzero value if `a` is greater than or equal to `b`, and a 0 otherwise.

`int _q_fgt(const long double *a, const long double *b)`
: Performs an ordered comparison of the extended precision values of `a` and `b`. Returns a nonzero value if `a` is greater than `b`, and a 0 otherwise.

`int _q_fle(const long double *a, const long double *b)`
: Performs an ordered comparison of the extended precision values of `a` and `b`. Returns a nonzero value if `a` is less than or equal to `b`, and a 0 otherwise.

`int _q_flt(const long double *a, const long double *b)`
: Performs an ordered comparison of the extended precision values of `a` and `b`. Returns a nonzero value if `a` is less than `b`, and a 0 otherwise.

`int _q_fne(const long double *a, const long double *b)`
: Performs an unordered comparison of the extended precision values of `a` and `b`. Returns a nonzero value if they are unordered or not equal; returns a 0 otherwise.

`long double _q_itoq(int a)`
: Converts the integer value of `a` to extended precision, and returns the extended precision value.

`long double _q_mul(const long double *a, const long double *b)`
: Returns `a*b`, computed to extended precision.

`long double _q_neg(const long double *a)`
: Returns `-a` without raising any exceptions.

`double _q_qtod(const long double *a)`
: Converts the extended precision value of `a` to double precision, and returns the double precision value.

`int _q_qtoi(const long double *a)`
: Converts the extended precision value of `a` to a signed integer by truncating any fractional part, and returns the signed integer value.

`float _q_qtos(const long double *a)`
: Converts the extended precision value of `a` to single precision, and returns the single precision value.

`unsigned int _q_qtou(const long double *a)`
: Converts the extended precision value of `a` to an unsigned integer by truncating any fractional part, and returns the unsigned integer value.

`long double _q_stoq(float a)`
: Converts the single precision value of `a` to extended precision, and returns the extended precision value.

`long double _q_sub(const long double *a, const long double *b)`
: Returns `a-b`, computed to extended precision.

`long double _q_utoq(unsigned int a)`
: Converts the unsigned integer value of `a` to extended precision, and returns the extended precision value.

3.5.3 Integer Routines

Conformant library support must include the integer routines listed in Table 3-4 (the routine interfaces are shown as C function prototypes). These integer routines must comply with the calling conventions described in Section 2.6, “Function Calling Sequence.” These routines have no side effects.

Table 3-4. Integer Routines

__div16	__div32	__rem16	__rem32
__udiv16	__udiv32	__urem16	__urem32

`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.

3.5.4 Optional Integer Routines

If the optional C `long long` data type is supported, then library support must also include the following `long long` integer routines. These routines must comply with the calling conventions described in Section 2.6, "Function Calling Sequence."

Table 3-5. Optional Integer Routines

`__div64`	`_d_dtoll`	`_f_ftoll`	`_q_lltoq`
`__udiv64`	`_d_dtoull`	`_f_ftoull`	`_q_qtoll`
`__rem64`	`_d_lltod`	`_f_lltof`	`_q_qtoull`
`__urem64`	`_d_ulltod`	`_f_ulltof`	`_q_ulltoq`

`long long __div64(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 __rem64(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.

`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 _f_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 _f_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 _f_lltof(long long a)`
: Converts the signed `long long` value of `a` to a single precision value, and returns the single precision value.

`float _f_ulltof(unsigned long long a)`
: Converts the unsigned `long long` value of `a` to a single precision value, and returns the single precision value.

`long double _q_lltoq(long long a)`
: Converts the `long long` value of `a` to an extended precision value, and returns the extended precision value.

`long long _q_qtoll(const long double *a)`
: Converts the extended precision value of `a` to a signed `long long` by truncating any fractional part, and returns the signed `long long` value.

`unsigned long long _q_qtoull(const long double *a)`
: Converts the extended precision value of `a` to an unsigned `long long` by truncating any fractional part, and returns the unsigned `long long` value.

`long double _q_ulltoq(unsigned long long a)`
: Converts the unsigned `long long` value of `a` to an extended precision value, and returns the extended precision value.

3.6 Function Argument and Return Type Checking in C

ABI-conforming implementations support the following mechanism for checking that arguments and return types of function calls match the called functions' signatures.

3.6.1 Signature Symbols

For every direct call to a non-static function in a source file (that is, a call using the function name as opposed to a call through a function pointer), the compiler system produces in the ELF object file a symbol of the following convention:

`__caller.`*`name`*`.`*`return_type`*`.`*`argument_types`*

For every non-static function definition, the compiler system produces a symbol of the following convention:

`__callee.`*`name`*`.`*`return_type`*`.`*`parameter_types`*

Table 3-6 explains the construction of the italicized fields in the symbol names:

Table 3-6. Italicized Fields in the Symbol Names

Field	Value	Meaning
`name`	ASCII string	The name of the called function
`return_type`	`basetype`	
`argument_types`	`basetype[basetype[...]]`	
`parameter_types`	`basetype[basetype[...]]`	

Table 3-7 explains the possible values for `basetype`.

Table 3-7. Basetype Values

Code	Meaning
`i`	Scalar type (e.g., `char`, `short`, `int`) of size <= 32 bits, passed in register
`l`	Scalar type of size = 64 bits, passed in register
`p`	Pointer, passed in a register
`f`	Float, passed in a register
`d`	Double float, passed in a register
`snum`	Struct, passed in a data register
`anum`	Struct, passed in an address register
`n`	An argument or parameter passed on the stack
`v`	Void
`x`	Start of a variable argument list (...)

Example:

Definition:

```
int foo(struct { int a,b; } parm1, double parm2);
```

Call:

```
struct { int a,b; } tmp;
foo(tmp, 1.0);
```

Special Symbols:

```
__callee.foo.i.s2f
__caller.foo.v.s2f
```

3.6.2 Return Value

In generating a signature symbol for a call to a function defined as returning a (non-void) value, if the return value is ignored by the caller, then the compiler may specify `i` as the return value type for the function.

3.6.3 Using Signature Symbols

The caller/callee match verification using signature symbols is implementation-dependent. The implementation must accept object modules that do not contain signature symbols.

3.7 Access to Architectural Features

The following set of intrinsics must be supported. These intrinsics allow access to hardware resources from a C application without using assembly inserts.

Table 3-8. Intrinsics for Access to Architectural Features

Feature	Intrinsic Function	Description
Saturation Mode	`__setnosat()`	Sets saturation mode off.
	`__setsat()`	Sets saturation mode on.
Rounding Mode	`__set2crm()`	Sets rounding mode to two's complement rounding mode.
	`__setcnvrm()`	Sets rounding mode to convergent rounding mode.
Scaling Mode	`void __setnoscale(void)`	Sets scaling mode off.
	`void __setdownscale(void)`	Scales down.
	`void __setupscale(void)`	Scales up.
Trapping	`void __trap_r(void *)`	Executes TRAP exception. Argument is passed in R0. Calling convention register usage is assumed.
	`int __trap_d(int)`	Executes TRAP exception. Argument is passed in D0. Calling convention register usage is assumed.
Registers	*Reading*:	
	`unsigned int __read_SR();`	These functions set the values of their related registers.
	`unsigned int __read_PCTL0();`	
	`unsigned int __read_PCTL1();`	
	`unsigned int __read_MCTL();`	
	`short * __read_VBA();`	
	`unsigned int __read_SP();`	
	`unsigned int __read_OSP();`	
	`int __read_EMR();`	
	Writing:	
	`void __write_SR(int);`	These functions set the values of their related registers.
	`void __write_PCTL0(int);`	Care is required when writing the EMR register. Bits cannot be set by the user, and clearing a given bit is done by writing a 1 to it using the BMCLR instruction.
	`void __write_PCTL1(int);`	
	`void __write_MCTL(int);`	
	`void __write_VBA(short *);`	
	`void __bit_clr_EMR(int);`	

Table 3-8. Intrinsics for Access to Architectural Features (Continued)

Feature	Intrinsic Function		Description
Registers *(continued)*	*SR Masks:*		
	`__SR_I2`	`(0x1 << 23)`	These masks provide access to individual SR bits or fields.
	`__SR_I1`	`(0x1 << 22)`	
	`__SR_I0`	`(0x1 << 21)`	
	`__SR_I_MASK`	`(0x7 << 21)`	
	`__SR_OVE`	`(0x1 << 20)`	
	`__SR_DI`	`(0x1 << 19)`	
	`__SR_EXP`	`(0x1 << 18)`	
	`__SR_S`	`(0x1 << 6)`	
	`__SR_S1`	`(0x1 << 5)`	
	`__SR_S0`	`(0x1 << 4)`	
	`__SR_S_MASK`	`(0x3 << 4)`	
	`__SR_RM`	`(0x1 << 3)`	
	`__SR_SM`	`(0x1 << 2)`	
	`__SR_T`	`(0x1 << 1)`	
	`__SR_C`	`(0x1 << 0)`	
	PCTL1 Masks:		
	`__PCTL1_COE`	`(0x1 << 16)`	These masks provide access to individual PCTL1 bits or fields.
	`__PCTL1_PODF2`	`(0x1 << 2)`	
	`__PCTL1_PODF1`	`(0x1 << 1)`	
	`__PCTL1_PODF0`	`(0x1 << 0)`	
	`__PCTL1_PODF_MASK`	`(0x7 << 0)`	
	MCTL Masks:		
	`__R7_AM3`	`(0x1 << 31)`	These masks provide access to individual MCTL bits or fields.
	`__R7_AM2`	`(0x1 << 30)`	
	`__R7_AM1`	`(0x1 << 29)`	
	`__R7_AM0`	`(0x1 << 28)`	
	`__R7_AM_MASK`	`(0xF << 28)`	
	`__R6_AM3`	`(0x1 << 27)`	
	`__R6_AM2`	`(0x1 << 26)`	
	`__R6_AM1`	`(0x1 << 25)`	
	`__R6_AM0`	`(0x1 << 24)`	
	`__R6_AM_MASK`	`(0xF << 24)`	
	`__R5_AM3`	`(0x1 << 23)`	
	`__R5_AM2`	`(0x1 << 22)`	
	`__R5_AM1`	`(0x1 << 21)`	
	`__R5_AM0`	`(0x1 << 20)`	
	`__R5_AM_MASK`	`(0xF << 20)`	

Table 3-8. Intrinsics for Access to Architectural Features (Continued)

Feature	Intrinsic Function	Description
Registers *(continued)*	*MCTL Masks (continued):*	
	`__R4_AM3          (0x1 << 19)`	
	`__R4_AM2          (0x1 << 18)`	
	`__R4_AM1          (0x1 << 17)`	
	`__R4_AM0          (0x1 << 16)`	
	`__R4_AM_MASK      (0xF << 16)`	
	`__R3_AM3          (0x1 << 15)`	
	`__R3_AM2          (0x1 << 14)`	
	`__R3_AM1          (0x1 << 13)`	
	`__R3_AM0          (0x1 << 12)`	
	`__R3_AM_MASK      (0xF << 12)`	
	`__R2_AM3          (0x1 << 11)`	
	`__R2_AM2          (0x1 << 10)`	
	`__R2_AM1          (0x1 << 9)`	
	`__R2_AM0          (0x1 << 8)`	
	`__R2_AM_MASK      (0xF << 8)`	
	`__R1_AM3          (0x1 << 7)`	
	`__R1_AM2          (0x1 << 6)`	
	`__R1_AM1          (0x1 << 5)`	
	`__R1_AM0          (0x1 << 4)`	
	`__R1_AM_MASK      (0xF << 4)`	
	`__R0_AM3          (0x1 << 3)`	
	`__R0_AM2          (0x1 << 2)`	
	`__R0_AM1          (0x1 << 1)`	
	`__R0_AM0          (0x1 << 0)`	
	`__R0_AM_MASK      (0xF << 0)`	
	EMR Masks:	
	`__EMR_GP6         (0x1 << 23)`	These masks provide access to individual EMR bits or fields. Note that some of these bits are read only.
	`__EMR_GP5         (0x1 << 22)`	
	`__EMR_GP4         (0x1 << 21)`	
	`__EMR_GP3         (0x1 << 20)`	
	`__EMR_GP2         (0x1 << 19)`	
	`__EMR_GP1         (0x1 << 18)`	
	`__EMR_GP0         (0x1 << 17)`	
	`__EMR_GP_MASK     (0x7F << 17)`	
	`__EMR_BEM         (0x1 << 16)`	
	`__EMR_NMID        (0x1 << 3)`	
	`__EMR_DOVF        (0x1 << 2)`	
	`__EMR_ILST        (0x1 << 1)`	
	`__EMR_ILIN        (0x1 << 0)`	

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) Executable and Linking Format (ELF) Specification, Version 1.1*. This chapter highlights differences between the ELF version 1.1 definition and the SC100 implementation.

This chapter focuses on 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.

4.1 Interface Descriptions

ELF presents two views of binary data, as shown in Figure 4-1:

- 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.

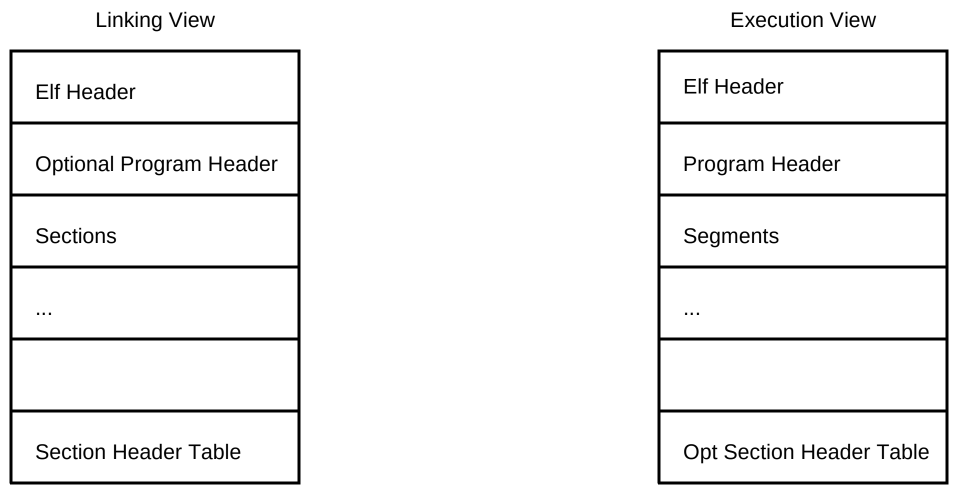


Figure 4-1. Object File Format

4.2 The ELF Header

The ELF header structure is shown in Example 4-1. This structure and its fields are defined by the ELF version 1.1 specification. SC100-specific code is shown in Example 4-2.

Example 4-1. ELF Header Structure

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

Example 4-2. SC100 Specifics

```
e_ident[EI_CLASS] = ELFCLASS32
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, different core revisions, and different ABI versions. The `e_flags` field is split into three parts:

- **Bits 0-5:** The core type. The defined core types are:

  ```
  #define EF_STARCORE_CORE_SC140          0
  #define EF_STARCORE_CORE_SC110          1
  ```

  Mixing object files with `EF_STARCORE_CORE_SC110` and `EF_STARCORE_CORE_SC140` will result in an object file with `EF_STARCORE_CORE_SC140`.

- **Bits 6-11:** The revision of the used core type. The defined core revisions are:

  ```
  #define EF_STARCORE_CORE_REV_UNKNOWN    0
  #define EF_STARCORE_CORE_REV_SC140_V1   1
  #define EF_STARCORE_CORE_REV_SC140_V2   2
  #define EF_STARCORE_CORE_REV_SC140_E    3
  ```

- **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
  ```

  ABI versions newer than 2.0 will be backward compatible. The result of linking object files with mixed ABI versions of 2.0 or higher will result in an object file with the lowest ABI version number. Linking with Pre-ABI or with non-conforming object files may result in linker errors or undetermined output.

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

As future cores become available, their respective core types and core revisions will be noted in the document, *SC100 Application Binary Interface Supplement*. This supplement will be available through the StarCore web site at http://www.starcore-dsp.com.

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

```
#define ELF32_EF_STARCORE_CORE(e_flags)  ((e_flags) & 0x3f)
#define ELF32_EF_STARCORE_REV(e_flags)   (((e_flags) >> 6) & 0x3f))
#define ELF32_EF_STARCORE_ABI(e_flags)   (((e_flags) >> 12) & 0x3f))
#define ELF32_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 Example 4-4. It is identical to the ELF version 1.1 definition.

Example 4-4. Section Header Structure

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

Sections used in SC100 ELF binaries are listed in Table 4-1. The section names listed in this table are case sensitive and are reserved for the system.

Table 4-1. SC100 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
`.bss`	`SHT_NOBITS`	`SHF_ALLOC, SHF_WRITE`	Uninitialized data [1]
`.zbss`	`SHT_NOBITS`	`SHF_ALLOC, SHF_WRITE`	Zero Data Area uninitialized data [1]
`.rela`*`section`*	`SHT_RELA`	None	Relocation info for *`section`* [2]
`.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 [3]
`.debug_abbrev`	`SHT_PROGBITS`	None	Abbreviation tables [4]
`.debug_aranges`	`SHT_PROGBITS`	None	Address range tables [4]
`.debug_frame`	`SHT_PROGBITS`	None	Call frame information [4]
`.debug_info`	`SHT_PROGBITS`	None	Debugging information entries [4]
`.debug_line`	`SHT_PROGBITS`	None	Line number information [4]
`.debug_loc`	`SHT_PROGBITS`	None	Location lists [4]
`.debug_macinfo`	`SHT_PROGBITS`	None	Macro information [4]
`.debug_pubnames`	`SHT_PROGBITS`	None	Global name tables [4]
`.SC100.delay_slots`	`SHT_PROGBITS`	None	Static delay slot information [5]

Notes:

1. Contents of `.bss` and `.zbss` sections are zeroed when loaded.
2. See Section 4.5, “Relocation.”
3. See Section 4.6, “NOTE Section.”
4. This information in DWARF2 format.
5. See Section 4.4, “Special Sections.”

4.4 Special Sections

A debug section called `.SC100.delay_slots` is used to hold all static delay slot information for each SC100 executable file. Assemblers must identify and generate sufficient relocatable file information (sections and relocation entries) to support this feature; linkers should need no special knowledge of this feature when creating executable files. Assemblers must also populate this section when creating executable files in absolute mode.

The `.SC100.delay_slots` section uses DWARF2 definitions like those used in the `.debug_line` section, and consists of an unpadded sequence of opcodes with zero or more operands. No special headers, padding, alignment, or sequence terminators are required.

Opcodes are represented by a single unsigned byte (8 bit) value. To accommodate future expansion without breaking existing readers, 4 bits are used for a unique ID (provides 16 opcodes) and 4 bits are used to indicate the size in bytes for the operands (provides up to 15 bytes of operands).

Three opcode IDs are initially required; additional IDs may be added later to support such features as overlays and position independent code. The required opcode IDs are:

- `SDS_EXPLICIT_OP` (explicit “delayed” instructions, for example, JSRD).

  Accepts two unsigned word (32 bits) operands. The first is the address of the explicit Variable Length Execution Set (VLES), and the second is the address of the delay slot VLES.

- `SDS_LONGLOOP_OP` (last two VLESes of a long loop).

  Accepts three unsigned word (32 bits) operands. The first is the address of the lpmark VLES, the second is the address of the next VLES, and the third is the address of the last VLES in the loop.

- `SDS_SHORTLOOP_OP` (last VLES of a short loop).

  Accepts two unsigned word (32 bits) operands. The first is the address of the first VLES, and the second is the address of the last (second) VLES in the loop.

Each opcode with operands is intended to completely describe all information potentially needed to implement features or checks that any debugger may reasonably expect to perform. This includes the static delay slot type, the addresses of the VLES immediately before the delay slot, and the address of each VLES in a static delay slot.

Debuggers need simply walk the byte stream (opcode then operands) of the `.SC100.delay_slots` section until all data is exhausted.

Example 4-5 and Example 4-6 define the opcode IDs and the macros for accessing opcode parts.

Example 4-5. Definition of Opcode IDs

```
#define     SDS_EXPLICIT_OP        0
#define     SDS_LONGLOOP_OP        1
#define     SDS_SHORTLOOP_OP       2
```

Example 4-6. Definition of Macros for Accessing Opcode Parts

```
#define     SDS_ID(opcode)      (((opcode) >> 4) & 0xf)
#define     SDS_SIZE(opcode)    ((opcode) & 0xf)
#define     SDS_OPCODE(id,size) (((size) & 0xf) | (((id) & 0xf) << 4))
```

4.5 Relocation

Each section which contains relocatable data has a corresponding relocation section of type SHT_RELA. The sh_info field of the relocation section defines the section header index of the section (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, the name of the relocation section is .rela prepended to the name of the data section.

A relocation entry is defined by the Elf32_Rela structure and associated macros as shown in Example 4-7. The r_offset field defines an offset into the data section to which the individual relocation applies. The r_info field specifies both the type of the relocation and the symbol used in computation of the relocation data.

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

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 Table 4-2, 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 Section 4.5.2, "Relocation Stack," particularly R_STARCORE_OPER and R_STARCORE_POP, for which a symbol value is rarely useful.

Example 4-7. Relocation Entry Defined with Elf32_Rela

```
typedef struct
    {
      Elf32_Addr   r_offset;
      Elf32_Word   r_info;
      Elf32_Sword  r_addend;
    } Elf32_Rela;

    #define ELF32_R_SYM(i)     ((i)>>8)
    #define ELF32_R_TYPE(i)    ((i)&0xff)
    #define ELF32_R_INFO(s,t)  (((s)<<8)|((t)&0xff))
```

4.5.1 Relocation Types

Device-specific relocations describe how a memory location should be patched by the linker. An ordinary relocation encodes exactly one instruction operand (or, in the case of data relocations, exactly one data value). It is the responsibility of the linker to ensure that the operand meets the range and alignment requirement specified by the relocation.

For each relocation type in Table 4-2, the **Type** field indicates the value extracted using `ELF32_R_TYPE`, both as a number and as a standard C preprocessor symbol. A brief abstract of the relocation follows in parentheses.

The **Size** field indicates the number of bits used to represent the relocation value. If the operand range is a subset of the values which can be represented in these bits, that restriction is indicated in parentheses.

The **Signedness** field indicates whether the relocation value is treated as signed, unsigned, or either.

The **Alignment** field indicates the alignment requirement in bits of the relocation value. This is the number of least significant bits in the relocation value which must be zero.

The **Shift** field indicates 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.

The **Special** field indicates any other special processing performed during relocation. For instructions which 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 (see Section 4.5.3, “Instruction Address vs. VLES Address”).

The **Encoding** field indicates 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 are used to indicate relocation value bits which are more significant than bits indicated by lower case letters. `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.

The **Applies To** field indicates which instructions or directives generate this relocation.

Example:

```
----S---sss--SSS ---sssssssssssS
    1   111  111    1198765432101
    9   432  876    10          5
```

Left to right, the “`S`”s represent bits 19-15 and the “`s`”s represent bits 14-0 of the relocation value. For a big-endian target, this corresponds to a byte representation of:

```
byte 0    byte 1    byte 2    byte 3
----S---  sss--SSS  ---sssss  sssssssS
    1     111  111     11987  65432101
    9     432  876     10            5
```

For a little-endian target, this corresponds to a byte representation of:

```
byte 0    byte 1    byte 2    byte 3
sss--SSS  ----S---  ssssssssS  ---sssss
111  111      1     65432101     11987
432  876      9            5     10
```

Table 4-2. Relocation Type Definitions

Type:	`1, R_STARCORE_DIRECT_8` (8-bit direct)
Size:	8
Signedness:	either
Alignment:	0
Shift:	0

Encoding:	`xxxxxxxx`
	`76543210`

Applies To:	DCB

Type:	`2, R_STARCORE_DIRECT_16` (16-bit direct)
Size:	16
Signedness:	either
Alignment:	0
Shift:	0

Encoding:	`xxxxxxxxxxxxxxxx`
	`1111119876543210`
	`543210`

Applies To:	DCW

Type:	`3, R_STARCORE_DIRECT_32` (32-bit direct)
Size:	32
Signedness:	either
Alignment:	0
Shift:	0

Encoding:	`xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx`
	`33222222222211111111119876543210`
	`1098765432109876543210`

Applies To:	DCL

Type:	`4, R_STARCORE_R9_1_1` (9-bit PC-relative)
Size:	9
Signedness:	signed
Alignment:	1
Shift:	1

Encoding:	`-------ssssssss-`
	`       87654321`

Special:	The PC is subtracted from the relocation value before any range checking or alignment checking is performed.
Applies To:	BF, BFD, BSR, BSRD, BT, BTD

Table 4-2. Relocation Type Definitions (Continued)

Type:	`5, R_STARCORE_R11_1_1` (11-bit PC-relative)
Size:	11
Signedness:	signed
Alignment:	1
Shift:	1

Encoding:	`-----ssssssssss-`
	`     1987654321`
	`     0`

Special:	The PC is subtracted from the relocation value before any range checking or alignment checking is performed.
Applies To:	BRA, BRAD
Type:	`6, R_STARCORE_R17_1_1` (17-bit PC-relative)
Size:	17
Signedness:	signed
Alignment:	1
Shift:	1

Encoding:	`--------sss----- ---ssssssssssssS`
	`        111         1119876543211`
	`        543         210           6`

Special:	The PC is subtracted from the relocation value before any range checking or alignment checking is performed.
Applies To:	BREAK, CONT, CONTD, DOSETUP0, DOSETUP1, DOSETUP2, DOSETUP3, SKIPLS
Type:	`7, R_STARCORE_R21_1_1` (21-bit PC-relative)
Size:	21
Signedness:	signed
Alignment:	1
Shift:	1

Encoding:	`----S---sss--SSS ---ssssssssssssS`
	`    2   111  111    1119876543211`
	`    0   543  987    210           6`

Special:	The PC is subtracted from the relocation value before any range checking or alignment checking is performed.
Applies To:	BF, BFD, BRA, BRAD, BSD, BSRD, BT, BTD

Table 4-2. Relocation Type Definitions (Continued)

Type:	`8, R_STARCORE_S7_0_0` (7-bit signed)
Size:	7
Signedness:	signed
Alignment:	0
Shift:	0
Encoding:	` ---------sssssss`
	`          6543210`
Applies To:	MOVE.W
Type:	`9, R_STARCORE_S15_0_0` (15-bit signed)
Size:	15
Signedness:	signed
Alignment:	0
Shift:	0
Encoding:	`---------ss----- ---sssssssssssss`
	`         11         1119876543210`
	`         43         210`
Applies To:	MOVE.B, MOVE.F, MOVE.L, MOVE.W, MOVES.F, MOVEU.B, MOVEU.W
Type:	`10, R_STARCORE_S15_1_0` (15-bit signed)
Size:	15
Signedness:	signed
Alignment:	1
Shift:	0
Encoding:	`---------ss----- ---sssssssssssss`
	`         11         1119876543210`
	`         43         210`
Applies To:	MOVE.F, MOVE.W, MOVES.F, MOVEU.W
Type:	`11, R_STARCORE_S15_2_0` (15-bit signed)
Size:	15
Signedness:	signed
Alignment:	2
Shift:	0
Encoding:	`---------ss----- ---sssssssssssss`
	`         11         1119876543210`
	`         43         210`
Applies To:	MOVE.L

Table 4-2. Relocation Type Definitions (Continued)

Type:	`12, R_STARCORE_S16_0_0` (16-bit signed)
Size:	16
Signedness:	signed
Alignment:	0
Shift:	0

Encoding:	`--------sss----- ---sssssssssssss`
	`        111         1119876543210`
	`        543         210`

Applies To:	ADDA, ADDNC.W, CMPEQ.W, CMPGT.W, IMPY.W, MAC, MOVE.F, MOVE.W, SUBNC.W
Type:	`13, R_STARCORE_S16_1_0` (16-bit signed)
Size:	16
Signedness:	signed
Alignment:	1
Shift:	0

Encoding:	`--------sss----- ---sssssssssssss`
	`        111         1119876543210`
	`        543         210`

Applies To:	AND.W, BMCHG.W, BMCLR.W, BMSET.W, BMTSET.W, BMTSTC.W, BMTSTS.W, EOR.W, MOVE.W, NOT.W, OR.W
Type:	`14, R_STARCORE_T16_0_0` (16-bit signed)
Size:	16
Signedness:	signed
Alignment:	0
Shift:	0

Encoding:	`-----------ss--- ---------------- --ssssssssssssss`
	`           11                       11119876543210`
	`           54                       3210`

Applies To:	MOVE.W
Type:	`15, R_STARCORE_S32_0_0` (32-bit signed)
Size:	32
Signedness:	signed
Alignment:	0
Shift:	0

Encoding:	`--------sssSS--- ---sssssssssssss --SSSSSSSSSSSSSS`
	`        11133       1119876543210   22222222221111`
	`        54310       210             98765432109876`

Applies To:	MOVE.L

Table 4-2. Relocation Type Definitions (Continued)

Type:	`16, R_STARCORE_U4_1_1` (4-bit unsigned)
Size:	4
Signedness:	unsigned
Alignment:	1
Shift:	1
Encoding:	`-------------uuu`
	`321`
Applies To:	MOVE.W
Type:	`17, R_STARCORE_U5_2_2` (5-bit unsigned)
Size:	5
Signedness:	unsigned
Alignment:	2
Shift:	2
Encoding:	`-------------uuu`
	`432`
Applies To:	MOVE.L
Type:	`18, R_STARCORE_U5_0_0` (5-bit unsigned)
Size:	5
Signedness:	unsigned
Alignment:	0
Shift:	0
Encoding:	`-----------uuuuu`
	`43210`
Applies To:	ADD, ADDA, ASLL, ASRR, CMPEQ.W, CMPGT.W, DECA, INCA, LSRR, SUBA
Type:	`19, R_STARCORE_U6_1_1` (6-bit unsigned)
Size:	6
Signedness:	unsigned
Alignment:	1
Shift:	1
Encoding:	`-----------uuuuu`
	`54321`
Applies To:	AND.W, BMCHG.W, BMCLR.W, BMSET.W, BMTSET.W, BMTSTC.W, BMTSTS.W, EOR.W, MOVE.W, NOT.W, OR.W

Table 4-2. Relocation Type Definitions (Continued)

Type:	`20, R_STARCORE_U6_0_0` (6-bit unsigned)
Size:	6
Signedness:	unsigned
Alignment:	0
Shift:	0

Encoding:	`----------uuuuuu`
	`          543210`

Applies To:	DOEN0, DOEN1, DOEN2, DOEN3, DOENSH0, DOENSH1, DOENSH2, DOENSH3
Type:	`21, R_STARCORE_U7_1_1` (7-bit unsigned)
Size:	7
Signedness:	unsigned
Alignment:	1
Shift:	1

Encoding:	`----------uuuuuu`
	`          654321`

Applies To:	MOVE.W
Type:	`22, R_STARCORE_U8_2_2` (8-bit unsigned)
Size:	8
Signedness:	unsigned
Alignment:	2
Shift:	2

Encoding:	`----------uuuuuu`
	`          765432`

Applies To:	MOVE.L
Type:	`23, R_STARCORE_V6_0_0` (6-bit unsigned)
Size:	6 (range 0..39)
Signedness:	unsigned
Alignment:	0
Shift:	0

Encoding:	`---------------- ----uuuuuu------`
	`                     543210`

Applies To:	EXTRACT, EXTRACTU, INSERT

Table 4-2. Relocation Type Definitions (Continued)

Type:	`24, R_STARCORE_W6_0_0` (6-bit unsigned)
Size:	6 (range 0..39)
Signedness:	unsigned
Alignment:	0
Shift:	0

Encoding:	`---------------- ----------uuuuuu`
	`                           543210`

Applies To:	EXTRACT, EXTRACTU, INSERT

Type:	`25, R_STARCORE_U16_0_0` (16-bit unsigned)
Size:	16
Signedness:	unsigned
Alignment:	0
Shift:	0

Encoding:	`--------uuu----- ---uuuuuuuuuuuuu`
	`        111         1119876543210`
	`        543         210`

Applies To:	AND, BMCHG, BMCHG.W, BMCLR, BMCLR.W, BMSET, BMSET.W, BMTSET, BMTEST.W, BMTSTC, BMTSTC.W, BMTSTS, BMTSTS.W, DOEN0, DOEN1, DOEN2, DOEN3, DOENSH0, DOENSH1, DOENSH2, DOENSH3, EOR, EOR.W, MOVE.B, MOVEU.B, MOVEU.W, OR, OR.W

Type:	`26, R_STARCORE_U16_1_0` (16-bit unsigned)
Size:	16
Signedness:	unsigned
Alignment:	1
Shift:	0

Encoding:	`--------uuu----- ---uuuuuuuuuuuuu`
	`        111         1119876543210`
	`        543         210`

Applies To:	AND.W, BMCHG.W, BMCLR.W, BMSET.W, BMTSET.W, BMTSTC.W, BMTSTS.W, EOR.W, MOVE.F, MOVE.W, MOVES.F, MOVEU.W, NOT.W, OR.W

Type:	`27, R_STARCORE_U16_2_0` (16-bit unsigned)
Size:	16
Signedness:	unsigned
Alignment:	2
Shift:	0

Encoding:	`--------uuu----- ---uuuuuuuuuuuuu`
	`        111         1119876543210`
	`        543         210`

Applies To:	MOVE.L

Table 4-2. Relocation Type Definitions (Continued)

Type:	`28, R_STARCORE_V16_0_0` (16-bit unsigned)
Size:	16
Signedness:	unsigned
Alignment:	0
Shift:	0
Encoding:	`-----------uu--- ---------------- --uuuuuuuuuuuuuu`
	`           11                       11119876543210`
	`           54                       3210`
Applies To:	BMCHG.W, BMCLR.W, BMSET.W, BMTSET.W, BMTSTC.W, BMTSTS.W, EOR.W, OR.W

Type:	`29, R_STARCORE_N16_0_0` (16-bit unsigned)
Size:	16
Signedness:	unsigned
Alignment:	0
Shift:	0
Encoding:	`--------uuu----- ---uuuuuuuuuuuuu`
	`        111         1119876543210`
	`        543         210`
Special:	The relocation value is exclusive-ORed with 0xFFFF before any range checking or alignment checking is performed.
Applies To:	AND, AND.W

Type:	`30, R_STARCORE_O16_0_0` (16-bit unsigned)
Size:	16
Signedness:	unsigned
Alignment:	0
Shift:	0
Encoding:	`-----------uu--- ---------------- --uuuuuuuuuuuuuu`
Special:	The relocation value is exclusive-ORed with 0xFFFF before any range checking or alignment checking is performed.
Applies To:	AND.W

Type:	`31, R_STARCORE_U32_0_0` (32-bit unsigned)
Size:	32
Signedness:	unsigned
Alignment:	0
Shift:	0
Encoding:	`--------uuuUU--- ---uuuuuuuuuuuuu --UUUUUUUUUUUUUU`
Applies To:	MOVE.B, MOVE.L, MOVEU.B, MOVEU.L

Table 4-2. Relocation Type Definitions (Continued)

Type:	`32, R_STARCORE_U32_1_0` (32-bit unsigned)
Size:	32
Signedness:	unsigned
Alignment:	1
Shift:	0

Encoding:	`--------uuuUU--- ---uuuuuuuuuuuuu --UUUUUUUUUUUUUU`
	`        11133       1119876543210   22222222221111`
	`        54310       210             98765432109876`

Applies To:	JF, JFD, JMP, JMPD, JSR, JSRD, JT, JTD, MOVE.F, MOVE.W, MOVES.F, MOVEU.W
Type:	`33, R_STARCORE_U32_2_0` (32-bit unsigned)
Size:	32
Signedness:	unsigned
Alignment:	2
Shift:	0

Encoding:	`--------uuuUU--- ---uuuuuuuuuuuuu --UUUUUUUUUUUUUU`
	`        11133       1119876543210   22222222221111`
	`        54310       210             98765432109876`

Applies To:	MOVE.L
Type:	`34, R_STARCORE_U32_16_16` (32-bit unsigned)
Size:	32
Signedness:	unsigned
Alignment:	16
Shift:	16

Encoding:	`--------uuu----- ---uuuuuuuuuuuuu`
	`        332         2222222221111`
	`        109         8765432109876`

Applies To:	AND

4.5.2 Relocation Stack

For those situations in which the relocation value cannot be expressed as a simple symbol value plus an addend, there are three special relocation types used 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 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).

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 Table 4-3. 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
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 in on the right.
Logical shift right:	Zeroes are shifted in on the left.
Arithmetic shift left:	Zeroes are shifted in on the right, and the most significant bit is always unaffected.
Arithmetic shift right:	Copies of the most significant bit are shifted in on 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 Table 4-2. 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.

4.5.3 Instruction Address vs. 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. To compensate for this, a PC-relative instruction must have the instruction offset subtracted from the PC-relative operand as follows:

1. In an ordinary relocation, the offset should be subtracted from the value in the `r_addend` field. Example:

VLES Offset	Instruction		
0 (1w prefix)	[		
2		tsteq	d2
4		doen3	d4
6		dosetup3	lptab+32
	]		

The “dosetup3” instruction would generate an ordinary relocation with the following field values:

```
r_info:   ELF32_R_INFO(<symbol number of lptab>, R_STARCORE_R17_1_1)
r_addend: 26 (32 - 6)
```

2. In an extended relocation, the subtraction of the offset should be inserted at the end of relocation expression, just before the `R_STARCORE_POP` operation. Example:

VLES Offset	Instruction		
0 (1w prefix)	[		
2		tsteq	d2
4		doen3	d4
6		dosetup3	lptab+4*ndx
	]		

The “dosetup3” instruction would generate a sequence of extended relocations with the following field values:

	r_info (type shown first, then symbol)		r_addend
	R_STARCORE_PUSH	<symbol number of lptab>	0
	R_STARCORE_PUSH	0	4
	R_STARCORE_PUSH	<symbol number of ndx>	0
	R_STARCORE_OPER	0	4 (*)
	R_STARCORE_OPER	0	7 (+)
*	R_STARCORE_PUSH	0	6
*	R_STARCORE_OPER	0	8 (-)
	R_STARCORE_POP	0	R_STARCORE_R17_1_1

The relocations marked with asterisks implement the offset subtraction.

4.6 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 Figure 4-2. 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, 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.

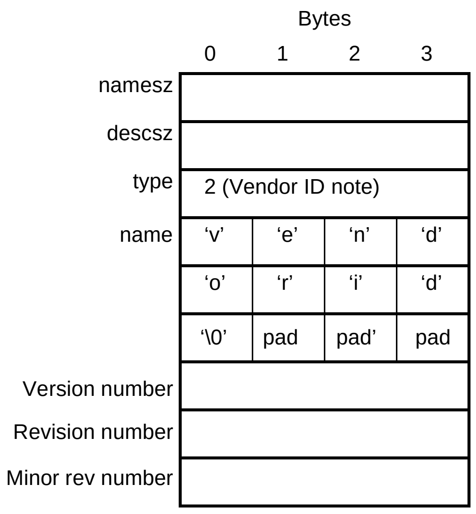


Figure 4-2. Vendor Identification Note Format

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 Figure 4-3.

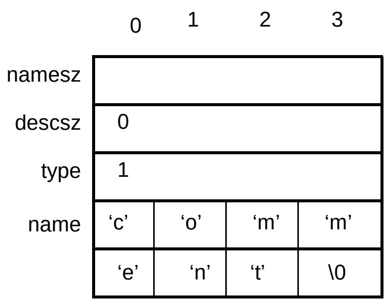


Figure 4-3. User (Application-Specific) Note Format

4.7 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. See Example 4-8 for a sample program header.

Example 4-8. Program Header

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

The program header members are described below.

`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`	Gives the number of bytes in segment’s file image. (May be zero.)
`p_memsz`	Gives the number of bytes in segment’s memory image. (May be zero.)
`p_flags`	Gives flags relevant to the segment. Defined flags are `PF_R`, `PF_W`, and `PF_X`.
`p_align`	Segment alignment requirements in file and memory.

4.8 Debugging Information

SC100 tools must use the Debug With Arbitrary Record Format (DWARF) debugging format, as defined in the *Tool Interface Standard (TIS) DWARF Debugging Information Format Specification*, Version 2.0.

4.8.1 DWARF Register Number Mapping

Table 4-4 outlines the register number mapping for the SC100 generation of DSP cores.

Table 4-4. SC100 Register Number Mapping

Register Name	Number	Abbreviation
Stack Pointer	0	SP
General Data Registers	1–16	D0–D15
Address Registers	17–32	R0–R15
Data Registers—extension portion	33–48	D0_e–D15_e
Data Registers—high portion	49–64	D0_h–D15_h
Data Registers—low portion	65–80	D0_l–D15_l
Loop Counter Registers	81–84	LC0–LC3
Modulo Registers	85–88	M0–M3
Offset Registers	89–92	N0–N3
Program Counter	93	PC
Clock Control Registers	94–97	PCTL0–PCTL3
Start Address Registers	98–101	SA0–SA3
Vector Base Address Register	102	VBA
Exception and Mode Register	103	EMR
Modifier Control Register	104	MCTL

Chapter 5
Assembler Syntax and Directives

Assemblers must support the directives, special characters, and syntax identified in this section. Details on these topics can be found in the *SC100 Assembly Language Tools User's Manual, Rev 2.0*.

5.1 Assembler Significant Characters

Several one- and two-character sequences are significant to the assembler and must be supported. Some have multiple meanings depending on the context in which they are used. These characters are listed in Table 5-1.

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
@	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

The assembler directives listed in Table 5-2 must be supported.

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
Symbol Definition	ENDSEC	End section
	EQU	Equate symbol to a value
	GLOBAL	Global section symbol declaration
	GSET	Set global symbol to a value
	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 / 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

Table 5-2. Assembler Directives

Type	Directive	Description
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 following sections provide details on assembler syntax.

5.3.1 Symbol Names

Symbol names follow these conventions:

- Symbol names can be from one to 4000 characters long.
- Symbol names 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 SC100 core registers are reserved by the assembler and cannot be used.

Examples of symbol names are shown below.

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. In order 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 may also be enclosed in double quotes (") in which case 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'

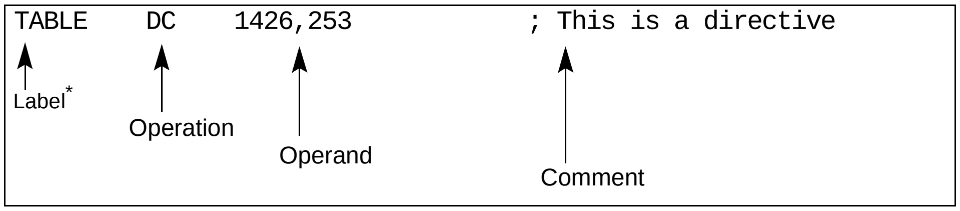
The assembler has a substring extraction capability using brackets ([]). Refer to the following example:

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

Substrings may 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

As shown in Figure 5-1, an assembly language source statement may include four fields in its most basic form: label, operation, operand, and comment.



*Begins in column 1

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, since 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 Section 5.3.3.1, "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.

5.3.3.1 Instruction Groups

The SC100 architecture supports instruction groups, which allow multiple instructions to be executed in parallel. Rules governing how instructions may 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 Example 5-1.

Example 5-1. Single-Line Instruction Group

```
    move.f (r2)+,d0    move.f (r2)+,d8     clr d5     ; Instruction group with 3 instructions
```

When delimited with brackets ([]), an instruction group may span multiple lines, as shown in Example 5-2.

Example 5-2. Multiple-Line Instruction Group (SC140)

```
    [
     mac d0,d1,d2            mac d3,d4,d5              ; multiply operands
     add d0,d1,d3            add d3,d4,d6              ; add operands
     move.f (r0)+,d0         move.w (r1)+,d1           ; load new operands
    ]
```

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. In this case, all characters preceding the label on the line must be whitespace characters—spaces or tabs.
- A label may 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 will be 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 may be one of three types:

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

5.3.3.4 Operand Field

The interpretation of the operand field is dependent on the contents of the operation field. The operand field, if present, must follow the operation field, and must be preceded by at least one space or tab. The operand field may 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

Comments are ignored by the assembler, but 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 that must be adhered to, in order to ensure correct code execution. Each core's reference manual defines the instructions that an assembly programmer or compiler will use. It is the role of the assembler and simulator to ensure that the instructions are legally used.

To ensure ABI conformance, it is required that third party assemblers and simulators follow the requirements defined in the design specification, "*Support in the Assembler and Simulator Required for Correct Reporting of SC100 Restrictions*." This specification defines which rules must be validated statically by the assembler. In some cases, it is only possible to validate a rule dynamically through the simulator. These cases are also documented.

For each rule violation, the specification defines an error or warning message. Each message contains an identifier (for example, A1, GG4, or LC7). Third party assemblers must include this identifier as part of the error or warning message. Beyond this requirement, the message may be formatted or worded as desired.

A set of programming rule test cases will be made available to third parties to validate conformance to the specification. The test suite and the design specification will be provided to third parties under a non-disclosure agreement with the StarCore Technology Center.

How to reach us:

USA/Europe/Locations Not Listed:
Motorola Literature Distribution
P.O. Box 5405
Denver, Colorado 80217
1-303-675-2140 or 1-800-441-2447

Japan
Motorola Japan Ltd.
SPS, Technical Information Center
3-20-1, Minami-Azabu. Minato-ku, Tokyo 106-8573 Japan
81-3-3440-3569

Asia/Pacific
Motorola Semiconductors H.K. Ltd.
Silicon Harbour Centre, 2 Dai King Street, Tai Po Industrial Estate, Tai Po, N.T., Hong Kong. 852-26668334

Technical Information Center
1 (800) 521-6274

Home Page
http://www.motorola.com/sps/dsp

Agere Systems Internet
http://www.agere.com

Email
docmaster@agere.com

N. America
Agere Systems Inc.
1-800-372-2447, FAX 610-712-4106
In CANADA: 1-800-553-2448, FAX 610-712-4106

Asia/Pacific
Agere Systems Singapore Pte. Ltd., Singapore
Tel. (65) 778 8833, FAX (65) 777 7495

China
Agere Systems (Shanghai) Co., Ltd., Shanghai
Tel. (86) 21 5047 1212, Fax (86) 21 5047 2266

Japan
Agere Systems Japan Ltd., Shinagawa-ku, Japan
Tel. (81) 3 5421 1600, FAX (81) 3 5421 1700

Europe Dataline
Tel. (44) 7000 582 368, FAX (44) 1189 328 148