RM0197
Reference manual

SH-4 generic and C specific application binary interface

Introduction

The SH-4 application binary interface (ABI) defines a system interface for application programs on SH-4 systems using the ELF executable and linking file format.

The language independent ABI defines the minimal conventions that must be used by all languages. The C specific ABI extends the language independent ABI to include the conventions required for the C language.

Adherence to this standard facilitates inter language calls, and the operation of language tools such as debuggers and operating systems.

Purpose

The purpose of this document is to describe the language independent application binary interface and the ANSI C language specific application binary interface for use on the SH-4 architecture, using the ELF executable and linking file format.

Contents

1 Language independent ABI

1.1 Scope and aims

The language independent ABI is the minimal set of conventions to be observed by all languages. A particular language is free to enhance the basic ABI for its own purposes and the particular ABI's for other languages should be consulted for further details. This chapter covers the following:

- memory organization, stack and global space,
- function frame layout,
- register usage conventions and call sequences.

1.2 Definition of terms

Function	A function is intended to be a language neutral term for that part of a program that can be invoked from other parts of the program as often as needed. It covers C functions and C++ functions.
Leaf function	A function that statically makes no further calls to other functions.
Frame	The stack space pushed for a function invocation.
Compilation unit	The individual unit of a program which is presented to a compiler at a single time. In C, a unit is loosely the collection of functions in a single file.
External reference	A reference from one compilation unit to an object defined outside the compilation unit.
Local reference	A local reference is a reference to an object within the same unit.
Top of stack	The lowest used address on the stack and usually corresponds to the most recent or current frame.
Bottom of stack	The highest used address in the stack and usually corresponds to the oldest frame on the stack.
Stack unwind	The process of decoding a function's stack frame to recreate the machine state at the point of call of the function. This process is required to support certain language features, in particular exception handling as found in C++.
Position independent code	Position independent code (PIC) is code that can be loaded and will successfully execute anywhere in a program's virtual address space, in other words, it is code that contains no absolute code or data addresses.

1.3 ABI variations

This ABI specification recognizes a number of variations. These variations are explained in *Section 1.3.1* to *Section 1.3.3*.

In general, code built for one variant is not compatible with code built for a different variant.

1.3.1 Byte ordering

Byte ordering defines how the bytes that make up an object are ordered in memory. Most significant byte (MSB) ordering or big-endian as it is often called, means that the most significant byte is located in the lowest addressed byte position in a storage unit. Least significant byte (LSB) ordering or little endian as it is often called, means that the least significant byte is located in the lowest addressed byte position in a storage unit. The SH-4 architecture supports both big-endian and little-endian byte ordering.

This ABI specification also supports both big-endian byte ordering and little-endian byte ordering.

Big-endian code and little-endian code may not be mixed in the same program.

1.3.2 Use of floating-point unit

Code may be created either to use the floating-point unit, or to perform all floating-point operations in software. Code that uses the floating-point unit is said to use the **fpu** model, code that performs all floating-point operations in software is said to use the **nofpu** model.

The choice of floating-point model affects the way that function arguments and results are passed.

In general, **fpu** model code and **nofpu** model code cannot be mixed in the same program.

1.3.3 Pervading floating-point precision

Within the SH-4 architecture, certain floating-point instructions operate in either single or double precision, depending upon the value of the floating-point precision flag, PR. When one of these floating-point instructions is executed, the PR flag must contain the correct value for the actual precision needed. This may require additional code to set the PR flag to the appropriate value.

This ABI specification has the concept of pervading floating-point precision. This is the precision (single or double) that the PR flag must be set to before calling a function, and before returning from a function. If most of an application's floating-point calculations are performed in the pervading precision, then less code is required to adjust the value of the PR flag.

Code may be created for either pervading single precision, or pervading double precision. Code with differing pervading precision may not be mixed in the same program.

1.4 Stack layout

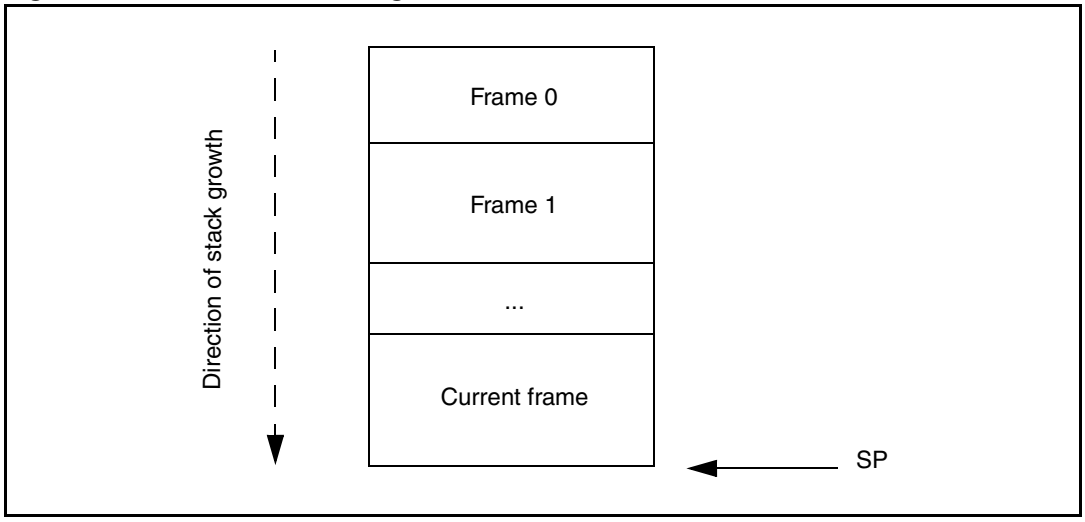
The stack of a single thread is a contiguous region of memory. Compilers allocate space on the stack to represent the local data of a function, usually referred to as a frame. Each called function creates and deletes its own frame. The stack grows as extra frames are allocated and in accordance with the architecture convention, the stack grows from high address to low address. The top of the stack (that is, the lowest address) is always referenced by a register known as the stack pointer, SP.

On SH-4, the stack pointer is always 4-byte aligned.

The stack pointer must address the top of stack at all times. The stack pointer contains the address of the last used byte on the stack. For instance, SP+0 is a valid address.

The topmost frame is the frame of the currently executing function. When a function is called, it allocates its own frame by decreasing SP; on exit, it deletes the frame by restoring SP to its original value. Each function is responsible for creating and deleting its own frame. Not all functions require a stack frame and a stack frame is allocated only if required. The stack growth is seen in *Figure 1*.

Figure 1. Overview of stack growth



As well as the stack pointer, a frame may also have a frame pointer, FP, a register used to address parts of the frame. Only a subset of frames need frame pointers.

On SH-4, if a stack frame uses a frame pointer, the frame pointer should be held in R14.

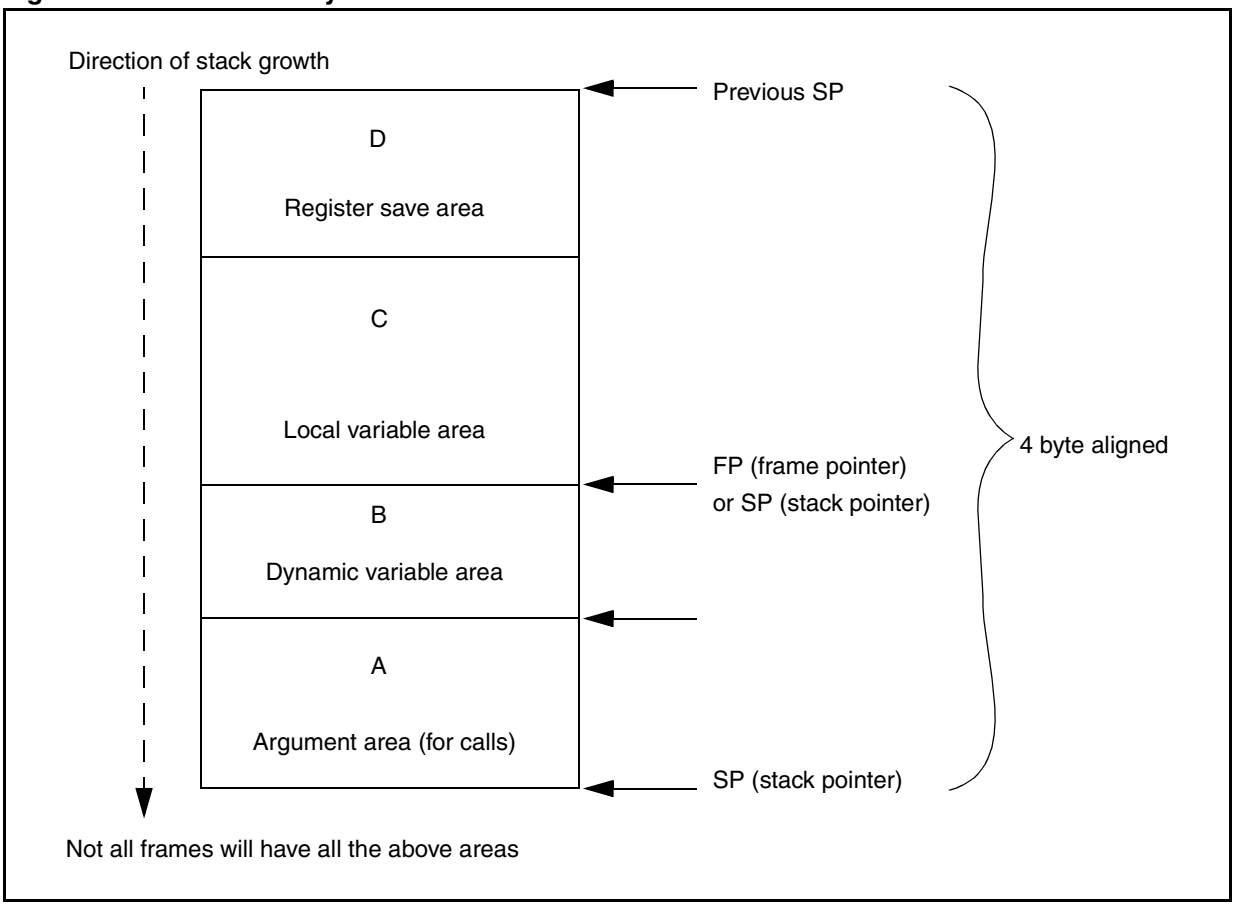
A register chosen to act as a frame pointer for a frame cannot be used for any other purpose, and must always be valid for the lifetime of that frame.

The ABI does not make any statements and does not assume anything about the state of the stack beyond the stack pointer (that is, any addresses < SP). The ABI is so written as to avoid any accesses beyond the current value of the stack pointer.

1.5 Frame layout

The SH-4 frame layout is described in *Figure 2*.

Figure 2. SH-4 frame layout



The stack frame is partitioned into five distinct areas to facilitate stack unwinding and for function calling.

- **Register save area (D)** is used to save and restore the callee save registers for this function, that is, the subset of local registers used by this function which are in the callee save set.
- **Local variable and temporary area (C)** is an area for local variables which need memory locations and for any compiler temporaries, for example, register spills. Its size is known at compile time. The objects in this area are accessed by offsets from either the stack pointer or the frame pointer.
- **Dynamic variable area (B)** is used for any objects which are allocated by extending the stack frame of the current procedure. For example, the alloca function in C. The size of this area is not known until run time but the existence of such an area is known at compile time. In practice, most frames will not have a dynamic area. The existence of a dynamic variable area means that the frame requires a frame pointer, since variables in the local variable area (C) are no longer at a fixed offset from the stack pointer. Space in the dynamic variable area is created by decreasing the stack pointer (which must be kept aligned).

- **Argument area (A)** is used to pass an argument list to functions called by the current function where the argument list is such that it cannot be accommodated in the parameter registers. The argument area contains the elements of the argument list that cannot be passed in registers because all the parameter registers of the appropriate type have been used. This area may be allocated as needed on each function call (by decreasing and increasing SP around the function call). Alternatively, if the frame has no dynamic variable area (B), then the argument area (A) can be created once on function entry by allocating the maximum area needed for all calls from this function. Allocating per function call requires code to manipulate SP at each function call, whereas allocating once on function entry means the stack frame will be larger throughout the lifetime of the function. The start (that is, the lowest address) of the argument area (A) must satisfy the alignment requirements for SP on entry to a function. If a function has both a dynamic variable area (B) and an argument area (A), the size of the dynamic variable area may not be altered while the argument area is in use. The argument area is only in use during the building of argument lists which means that dynamic memory allocations (and deallocations) must not happen during argument list building if the argument area (A) is also in use. In the local variable area (C) for example, any alloca processing may have to be hoisted out of the argument list generation.

A leaf function may not require any space on the stack if all of its local variables and intermediate expressions can be allocated to scratch registers. If a leaf function requires no space on the stack, then it does not need to create a frame.

1.6 Global data

When position independent code (PIC) is not required, global data may be accessed using absolute addressing. See *Chapter 4: Coding examples on page 28* for examples.

When position independent code is required, global data should be accessed using a global offset table. See *Chapter 3: Position-independent code on page 20* for details.

1.7 Function linkage and parameter passing

1.7.1 Function linkage

The linkage register is used in function calling to record the return address for any function call.

On SH-4, the PR register is used as the linkage register.

1.7.2 Parameter passing

The term ‘parameter passing convention’ refers to how the actual machine resources (for example, registers, memory) are used to pass the parameters from the caller to the callee.

Parameters are passed in registers or memory.

On SH-4, registers R4 to R7, FR4 to FR11 and the stack argument area (A) are used for parameter passing.

The mapping from an argument list to machine resources is dependent on the type associated with the elements of the list and is therefore language dependent. *Chapter 2: ANSI C ABI on page 12* describes the details of these two mappings for ANSI C.

1.7.3 Register usage conventions

Several terms are defined for use in the specification of the register usage conventions.

- A register is CALLER SAVE if its value is not guaranteed to be preserved across function calls. Such a register is also termed SCRATCH since the caller will have to save and restore the register around function calls.
- A register is CALLEE SAVE if its value is guaranteed to be preserved across calls. The implication is that the callee will either not modify the register or else save it to memory.
- A register is RESERVED if it has some special use required either by a software convention or by the hardware.

The register usage for SH-4 is given in *Table 1*.

Table 1. SH-4 ABI register usage

Register name	Usage
R0 to R3	Return value, caller save
R2	Large struct return address, caller save
R4 to R7	Parameter passing, caller save
R8 to R13	Callee save
R12	Global context pointer, GP, callee save
R13	Callee save
R14	Frame pointer, FP, callee save
R15	Stack pointer, SP, callee save
FR0 to FR3	Return value, caller save
FR4 to FR11	Parameter passing, caller save
FR12 to FR15	Callee save
MACH	Caller save
MACL	Caller save
PR	Linkage register, caller save
FPSCR	PR bit must be pervading precision on entry to prolog and exit from epilog FR and SZ bits must be zero on entry to prolog and exit from epilog(1)
FPUL	Caller save
SR	Status register: S, M, Q and T bits are caller save(2)
GBR	Reserved

1. The SZ and FR bits in the floating-point status register must be zero upon entry to prolog and exit from epilog of any function. The PR bit must be set to the correct value for the pervading precision upon entry to prolog and exit from epilog of any function: 0 for single-precision, 1 for double-precision. See *Section 1.3.3: Pervading floating-point precision on page 5*.
2. The M, Q, S and T bits in the status register have undefined values on entry to a function, and their values are not guaranteed to be preserved across calls.

1.8 DWARF register assignments

The DWARF information generated by GCC references SH-4 registers as numbers. *Table 2* gives the allocation of register numbers (as provided by GCC 4.2.1).

Table 2. DWARF register assignments

Register name	Number	Notes
R0 to R15	0 to 15	
PC	16	Linux only
PR	17	
GBR	18	Previously 19
VBR	19	Not generated by GCC, but used by GDB
MACH	20	
MACL	21	
SR	22	Linux only
SR.T	22	Previously 18
FPUL	23	
FPSCR	24	
FR0 to FR15	25 to 40	
SSR	41	Not generated by GCC, but used by GDB
SPC	42	Not generated by GCC, but used by GDB
DBR	59	Not generated by GCC, but used by GDB
SGR	60	Not generated by GCC, but used by GDB
XF0 to XF15	61 to 76	Not generated by GCC, but used by GDB
XD0 to XD7	87 to 94	

1.9 Function prolog and epilog

The generic ABI does not specify an exact code sequence that must be performed on entry (the prolog) or on exit (the epilog) of a function. Instead, function prologs and epilogs are characterized by a set of tasks which are carried out.

On entry to a function, several optional tasks can be performed.

- Create a stack frame. This is performed by decreasing SP. No accesses beyond SP are permitted. The decrement of SP may be performed by a single instruction, or by a number of instructions.
- Create a working register set. A function always has access to a set of scratch (caller-save) registers. If it needs further registers, it must save and use callee-save registers.
- Save the return address.
- Establish a frame pointer if needed, by copying SP to the frame pointer register.
- Establish a global context pointer, GP, if needed, as an offset from the PC.

On exit from a function, four tasks are performed.

- Restore the callee save registers that were saved in the prolog code.
- Restore the return address.
- Delete the stack frame by restoring SP. Again, the increment of SP may be performed by a number of instructions but after each of these increments, SP must be correctly aligned.
- Perform a return to the caller using the return address.

2 ANSI C ABI

This section covers the run-time model for the implementation of ANSI C based on the language independent ABI.

2.1 Type mapping

Signed integers use a two's complement representation. Single-precision and double-precision floating-point numbers use the IEEE754 single-precision and double-precision formats.

2.1.1 SH-4 ABI fundamental type mapping

The type mapping for the SH-4 ABI is given in *Table 3*.

Table 3. SH-4 ABI mapping of ANSI C data types

ANSI C type	SH-4 ABI representation		
	Type	Size (bytes)	Alignment (bytes)

`char`
`signed char`
`unsigned char`
	signed integer
	signed integer
	unsigned integer
		1	1

`short int (signed)` `unsigned short int`	signed integer unsigned integer	2	2

`int (signed)`
`unsigned int`
`enum`
	signed integer
	unsigned integer
	signed integer
		4	4

`long int (signed)` `unsigned long int`	signed integer unsigned integer	4	4
`long long int (signed)` `unsigned long long int`	signed integer unsigned integer	8	4
`float`	single-precision floating-point	4	4
`double` `long double`	double-precision floating-point	8	4
`_Complex float`	single-precision floating-point pair, real part first	8	4
`_Complex double`	double-precision floating-point pair, real part first	16	4
`pointer`	unsigned integer	4	4

2.1.2 Null pointer

A null pointer (for all types) has the value zero.

2.1.3 Function pointers

On SH-4, a function pointer contains the address to call to enter the function.

2.1.4 Aggregate types

Arrays of types inherit the alignment of the elements of the array. Each element of an array is correctly aligned, that is an array will have the alignment of its elements. The size of an array is always a multiple of the element alignment and an array does not cause any extra (internal or tail) padding to be added.

Structures and unions have the alignment of their most strictly aligned member. Each member is assigned to the lowest available offset with the appropriate alignment. This may require internal padding. The contents of any padding is undefined. The size of a structure is always a multiple of its alignment and this may require tail padding.

This padding allows the following familiar C idiom to be used to allocate arrays of structures:

```
struct T {...} *ptr;
ptr = (struct T *)malloc (n * sizeof(struct T));
```

The address of a structure or union is its lowest (smallest) address. Structure fields are allocated in declarative order from lowest address to highest address. Fields of the structure or union are addressed with positive offsets from the base of the structure. The qualifier `volatile` applied to an aggregate type has no effect on its layout. The `volatile` qualifier applied to a structure or union field will also not affect the layout of the record.

Bit-fields

Bit-fields are associated with an underlying integral type (char, short, int, long or long long). The associated type is the type used in the bit-field definition.

Bit-fields may be of any integral type (char, short, int, long, long long, enum) and can be of any size from 0 to the maximum width of the underlying type. For example, a char bit-field can be up to 8 bits wide whereas a long long bit-field has a maximum of 64 bits.

On SH-4, bit-fields obey the same size and alignment rules as other structure members, with several additions.

- A bit-field is allocated within a storage unit whose alignment and size are the same as the alignment and size of the underlying type of the bit-field. The bit-field must reside entirely within this storage unit: a bit-field never straddles the natural boundary of the underlying type.
- In the little-endian ABI, bit-fields are allocated from right to left (least to most significant) within a storage unit. In the big-endian ABI, bit-fields are allocated from left to right (most to least significant) within a storage unit.
- A bit-field shares a storage unit with the previous structure member if there is sufficient space within the storage unit.
- Plain bit-fields are treated as signed.

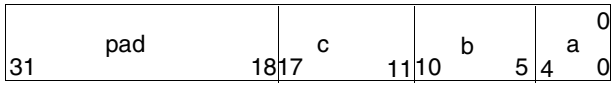
- The effect of a zero-length bit-field is to:
  - pad up to the alignment of the underlying type of the zero-length bit-field, and
  - force the next (non-zero-length) field to be allocated in a new storage unit.
- Unnamed bit-fields (including zero-length unnamed bit-fields) do not affect the overall alignment of a struct.

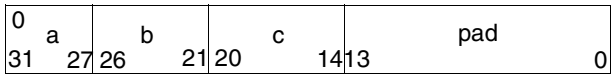
Table 4 shows some examples of bit-field layout. Big-endian byte numbers are shown in the upper left corners, little-endian byte numbers in the upper right corners, and bit numbers in the lower corners.

Table 4. SH-4 bit-field examples

<table>
<tr>
<td>

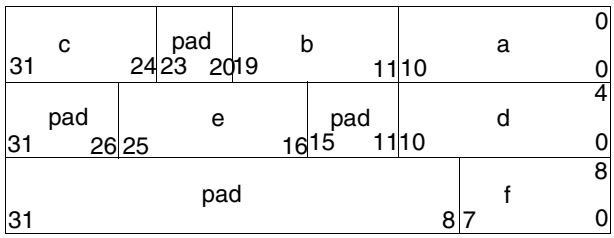
```
struct {
    int a:5;
    int b:6;
    int c:7;
};
```

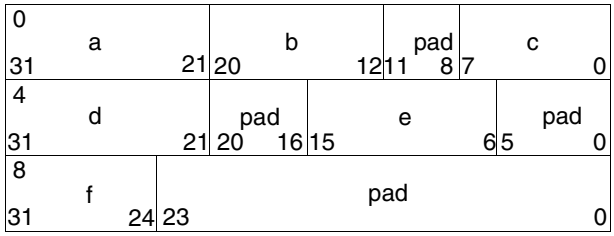
</td>
<td>


SH-4 little-endian layout, 4-byte aligned, sizeof is 4


SH-4 big-endian layout, 4-byte aligned, sizeof is 4
</td>
</tr>
<tr>
<td>

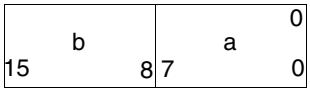
```
struct {
    short a:11;
    int b:9;
    char c;
    short d:11;
    short e:10;
    char f;
};
```

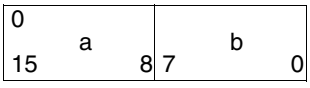
</td>
<td>


SH-4 little-endian layout, 4-byte aligned, sizeof is 12


SH-4 big-endian layout, 4-byte aligned, sizeof is 12
</td>
</tr>
<tr>
<td>

```
struct {
    char a;
    short b:8;
};
```

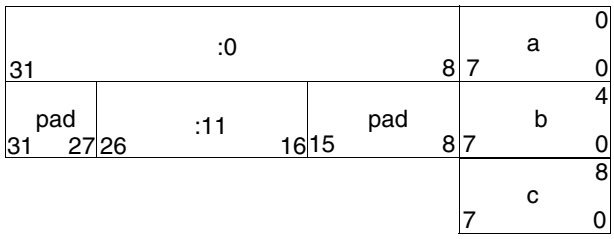
</td>
<td>


SH-4 little-endian layout, 2-byte aligned, sizeof is 2


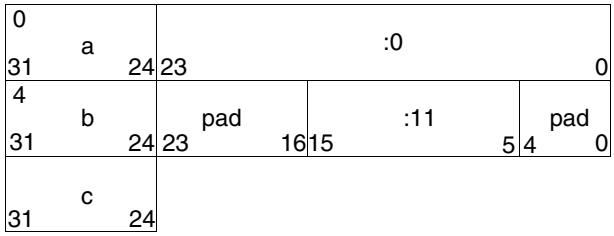
SH-4 big-endian layout, 2-byte aligned, sizeof is 2
</td>
</tr>
</table>

Table 4. SH-4 bit-field examples (continued)

```
struct {
    char a;
    int :0;
    char b;
    short :11;
    char c;
    char :0;
};
```



SH-4 little-endian layout, 1-byte aligned, sizeof is 9



SH-4 big-endian layout, 1-byte aligned, sizeof is 9

2.2 Function results and argument passing

2.2.1 SH-4 function results

Table 5 lists how function values are returned on SH-4 for the C fundamental types.

Table 5. SH-4 ABI return location for fundamental types

Result type	Returned in
`unsigned char`	R0. Zero extended (that is, bits 8 to 31 are zero).
`char` `signed char`	R0. Sign extended (that is, bit 7 is duplicated through bits 8 to 31).
`unsigned short int`	R0. Zero extended (that is, bits 16 to 31 are zero).
`short int (signed)`	R0. Sign extended (that is, bit 15 is duplicated through bits 16 to 31).

`unsigned int`
`int (signed)`
`unsigned long`
`long (signed)`
	R0.

`enum`	R0.
`unsigned long long` `signed long long`	Little-endian model: R0 (least significant), R1 (most significant). Big-endian model: R0 (most significant), R1 (least significant).
`float`	Fpu model: FR0. Nofpu model: R0.

Table 5. SH-4 ABI return location for fundamental types (continued)

Result type	Returned in
`double` `long double`	Fpu model: DR0. Nofpu, little-endian model: R0 (least significant), R1 (most significant). Nofpu, big-endian model: R0 (most significant), R1 (least significant).
`_Complex float`	Fpu model: FR0 (real part), FR1 (imaginary part). Nofpu model: R0 (real part), R1 (imaginary part).
`_Complex double`	Fpu model: DR0 (real part), DR2 (imaginary part). Nofpu, little-endian model: R0 (real part, least significant), R1 (real part, most significant), R2 (imaginary part, least significant), R3 (imaginary part, most significant). Nofpu, big-endian model: R0 (real part, most significant), R1 (real part, least significant), R2 (imaginary part, most significant), R3 (imaginary part, least significant).
`pointer`	R0.
`void`	Nowhere.

Aggregate types not bigger than 8 bytes that have the same size and alignment as one of the integer scalar types are returned in the same registers as the integer type they match. For example, a 2-byte aligned structure with a size of 2 bytes has the same size and alignment as a `short int`, and is returned in R0. A 4-byte aligned structure with size 8 bytes has the same size and alignment as a `long long int`, and is returned in R0 and R1. When an aggregate type is returned in R0 and R1, R0 contains the first four bytes of the aggregate, and R1 contains the remainder. If the size of the aggregate type is not a multiple of 4 bytes, the aggregate is tail-padded up to a multiple of 4 bytes. The value of the padding is undefined. For little-endian targets the padding appears at the most significant end of the last register used. For big-endian targets the padding appears at the least significant end of the last register used.

Note: Aggregates that are 8 bytes or smaller are only returned in registers if they have the same size and alignment as an integer scalar type. This is sometimes referred to as the GCC struct return rule.

For example, given:

```
struct s { char c[3]; } str;
struct s foo(void) { return str; }
```

the return value from `foo()` will be in memory, not in R0, because there is no 3-byte integer type.

All other aggregate types are returned by address. The caller function passes the address of an area large enough to hold the aggregate value in R2. The called function stores the result in this location.

2.2.2 SH-4 argument passing

Actual parameters are processed in lexical order and are mapped to registers or memory as described in this section. The lexical order is either the order the parameters appear in the function prototype or the order of the actual arguments in the absence of a prototype.

The type conversions listed below are performed as dictated by the ISO C standard.

- When a prototype is specified, the actual arguments are converted to the corresponding formal parameter type. In the ellipsis part of a function that takes a variable number of arguments, `char`, `short`, `unsigned char`, `unsigned short` are converted to type `int` and `float` is converted to type `double`.
- When a prototype is not specified, `char`, `short`, `unsigned char`, `unsigned short` are converted to type `int` and `float` is converted to type `double`.

Each scalar actual parameter is mapped to one or more registers or stack longwords, as described in *Table 6*. Parameters passed on the stack are passed in the argument area (A) at the next available longword (32-bit), starting from the lowest address in the argument area.

Table 6. SH-4 parameter passing location for fundamental types

Actual parameter type	Passed in

`unsigned char`
`char`
`signed char`
`unsigned short`
`short (signed)`
`unsigned int`
`int`
`enum`
`unsigned long`
`long`
`pointer`
	Next available register from R4, R5, R6, R7 or, if none of these registers are available, next longword on the stack.
	For parameters narrower than 4 bytes, the upper bits are undefined.

`unsigned long long` `signed long long`	Next two available registers from R4, R5, R6, R7 or, if there are not two registers available, next two longwords on the stack. The parameter is passed either entirely in registers or entirely on the stack. In the little-endian model, the least significant half of the value is passed in the lower numbered register or lower addressed stack slot. In the big-endian model, the most significant half of the value is passed in the lower numbered register or lower addressed stack slot.
`float`	In the fpu, little-endian model: the next available register from the list FR5, FR4, FR7, FR6, FR9, FR8, FR11, FR10. (This allocation ordering is necessary to get the correct behavior for varargs functions.) In the fpu, big-endian model: the next available register from the list FR4, FR5, FR6, FR7, FR8, FR9, FR10, FR11. In the nofpu model: the next available register from the list R4, R5, R6, R7. Otherwise for all models, if no register is available from the list, the parameter is passed in the next longword on the stack.

Table 6. SH-4 parameter passing location for fundamental types (continued)

Actual parameter type	Passed in
`double`	In the fpu model: the next available register from the list DR4, DR6, DR8, DR10. If this requires skipping a single-precision floating-point register, then that single-precision register is marked as unavailable. It cannot be allocated to a later parameter. For example, to assign a double-precision floating-point element where FR4 has already been allocated a single-precision element, but FR5-FR11 are still available. The double-precision element cannot be assigned to DR4 because FR4 is allocated, so it is assigned to DR6. Now FR5 becomes unavailable. If the next floating-point element to be assigned is single-precision, it will be assigned to FR8, leaving FR5 unused. Otherwise, if no double-precision floating-point parameter register is available, the parameter is passed in next two longwords on the stack. The parameter is always passed entirely in registers or entirely on the stack. In the little-endian model, the least significant half of the value is passed in the lower addressed stack slot. In the big-endian model, the most significant half of the value is passed in the lower addressed stack slot.
	In the nofpu model: the next two available registers from R4, R5, R6, R7 or, if there are not two registers available, next two longwords on the stack. The parameter is always passed entirely in registers or entirely on the stack. In the little-endian model, the least significant half of the value is passed in the lower numbered register or lower addressed stack slot. In the big-endian model, the most significant half of the value is passed in the lower numbered register or lower addressed stack slot.
`_Complex float`	In the fpu model: the next two available registers from the list FR4, FR5, FR6, FR7, FR8, FR9, FR10, FR11. In the nofpu model: the next two available registers from the list R4, R5, R6, R7. For both models, if there are not two registers available from the list, the parameter is passed in the next two longwords on the stack. The parameter is always passed entirely in registers or entirely on the stack. The real part of the value is passed in the lower numbered register or lower addressed stack slot. The imaginary part of the value is passed in the higher numbered register or higher addressed stack slot.
`_Complex double`	In the fpu model: the next two available registers from the list DR4, DR6, DR8, DR10. If this requires skipping a single-precision floating-point register, then that register is marked as unavailable: it cannot be allocated to a later parameter. In the nofpu model: in the four registers R4, R5, R6, R7 if available. For both models, if the registers are not available, the value is passed in the next four longwords on the stack. The parameter is always passed entirely in registers or entirely on the stack. The real part of the value is passed first. This means that the real part will appear in lower numbered registers or lower addressed stack slots than the imaginary part. In the little-endian model, the least significant half of the real and imaginary parts is passed in the lower numbered register or lower addressed stack slot. In the big-endian model, the most significant half of the real and imaginary parts is passed in the lower numbered register or lower addressed stack slot.

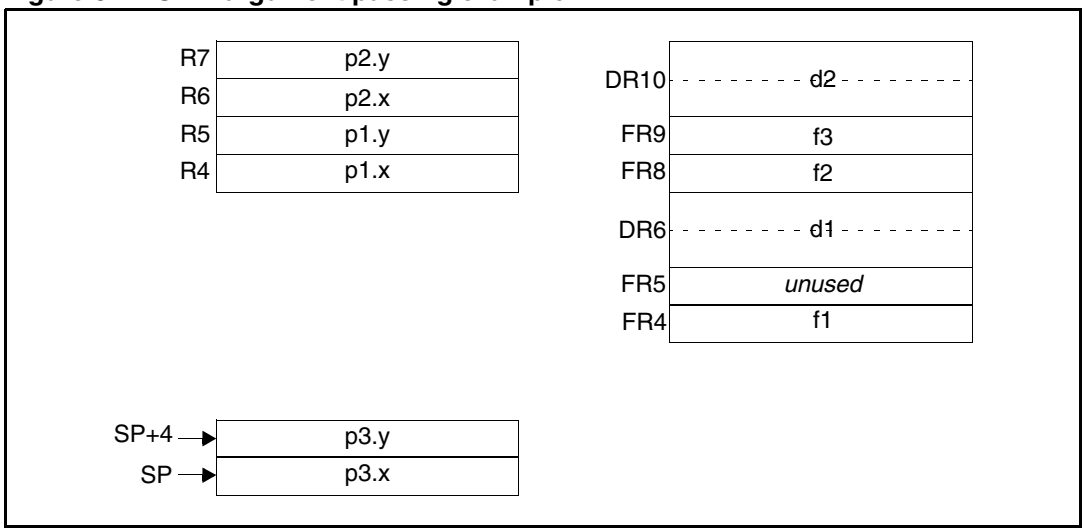
The mapping of an aggregate parameter is determined by the memory layout of that parameter. The parameter is broken into a series of 4-byte elements. If there are enough registers available from the list R4, R5, R6, R7 to contain all of the 4-byte elements of the parameter, the 4-byte elements are assigned to the available registers in order. Otherwise, if there are not enough registers available, all of the 4-byte elements are assigned to successive longwords on the stack.

When the size of an aggregate parameter is not a multiple of 4 bytes, it is tail padded up to a multiple of 4 bytes. The value of this padding is undefined. For little-endian targets the padding appears at the most significant end of the last element, for big-endian targets the padding appears at the least significant end of the last element.

Example

```
typedef struct s_point {
   float x, y;
} point;
int foo(point p1, float f1, double d1, float f2, point p2, point p3,
float f3, double d2);
foo(p1,f1,d1,f2,p2,p3,f3,d2);
```

Figure 3. SH-4 argument passing example



2.3 Symbol names

It is a property of the target environment whether C identifiers are prepended with an underscore when they are used as global symbol names in the resulting object file.

On a “bare machine” target environment (an environment where there is no operating system), C identifiers used as global symbol names in the resulting object file are prepended with an underscore. For example, the entry point of the function `foo()` is represented by the symbol `_foo` in the object file.

In the Linux environment, C identifiers are traditionally not prepended with an underscore. For example, the entry point of the function `foo()` will be represented by the symbol `foo` in the object file.

3 Position-independent code

This section describes how position-independent code is implemented in Unix-like environments, in particular Linux. However, the method of supporting position-independent code may be different in other target environments.

Position-independent code is useful in order to support dynamically loaded objects, as it allows the objects to be loaded at different virtual addresses without having to change the program text image. Multiple processes can share a single position-independent text segment, even though the segment resides at a different virtual address in each process.

3.1 The global offset table

For code to be position-independent, it must contain no absolute addresses. Instead, all absolute addresses are collected into a table, called the **global offset table**. When the code requires an absolute address, it loads the appropriate entry from the global offset table. Code that references the global offset table uses PC-relative addressing, and is therefore position-independent. *Chapter 4: Coding examples on page 28* gives example code sequences for position-independent addressing of functions and data using the global offset table.

Each shared object has a separate global offset table. When control is transferred from a function in one shared object to a function in another shared object, the global offset table used will also change.

The global offset tables reside in the data segment of a process. Processes do not share data segments, therefore each process has its own global offset tables. A process has one global offset table for each shared object it references.

When the memory image of a process is created, the contents of its global offset tables are relocated by the dynamic linker, to reflect the absolute addresses assigned to that process.

For efficiency, if a function needs to access the global offset table, then it calculates a pointer to the global offset table in the prolog, and saves this in the GP register (R12). This value may be the base address of the global offset table, or it may be biased from the base of the global offset table. As R12 is a callee-save register, it must be saved before this calculation, and the original value restored in the function epilog.

3.2 The procedure linkage table

The procedure linkage table (PLT) addresses two problems.

1. Shared objects can often contain calls to a very large number of global functions, many of which are never executed. The global offset table contains an entry for each of these functions. At the time a process is created, the dynamic linker needs to resolve all of these entries in the global offset table, even though many of them are never needed. This can be time-consuming. An alternative is to have the entry in the global offset table point to a piece of stub code that resolves the symbol reference at run-time, if and only if the function is called. This is called **lazy** symbol resolution.
2. In order to call a global function, it is necessary to load the address of the function from the global offset table, and perform an indirect call. This can require several instructions. An alternative is to perform a PC-relative call to a piece of stub code which loads the entry from the global offset table and performs the indirect call. This can improve code density on many architectures, though it requires more instructions to be executed.

The procedure linkage table collects together these stubs. It contains an entry for each global function that may be called. Each entry contains two pieces of code: the first piece loads the function address from the global offset table and calls it (that is, the stub required for problem *2*), the second piece calls a runtime fix up function that resolves the true address of the function (that is, the stub required for problem *1*).

Figure 4 illustrates the code in an SH-4 PLT entry. This is just an example of how a PLT could be implemented, there is no requirement that this exact code sequence is used.

Figure 4. SH-4 position-independent PLT entry

```
PLTn:   MOV.L   .L1,R0
        MOV.L   @(R0,R12),R0
        JMP     @R0
        NOP
.L0:    MOV.L   @(8,R12),R0
        MOV.L   .L2,R1
        JMP     @R0
        MOV.L   @(4,R12),R0
        NOP
        NOP
.L1:    .long   n@GOT
.L2:    .long   reloc. offset
```

The code beginning at label `PLTn` loads the value of the function's entry in the global offset table and calls it.

The code beginning at label `.L0` loads R0 with the value of the second global offset table entry `@(4,R12)`, and R1 with the relocation offset of the function, and then calls the address in the third global offset table entry `@(8,R12)`. The second and third entries in the global offset table have special values: the second entry is a pointer to a `link_map` data structure, that provides link data specific to the shared object, and the third entry is the address of a run-time fix up function, `_dl_runtime_resolve`. The relocation offset is the offset into the dynamic relocation table of the relocation that will fix-up the global offset table entry for the function. So the overall effect of this code is to perform the function call

```
_dl_runtime_resolve(&link_map, reloc_offset);
```

but note that the function arguments are not passed in the normal argument passing registers.

The sequence of events involved in lazy run-time symbol resolution are described below.

1. When the memory image of the process is created, entry 2 of the global offset table is initialized to be the address of the `link_map` data structure, and entry 3 of the global offset table is initialized to be the address of `_dl_runtime_resolve`, a function that performs run-time symbol resolution.

   Additionally, for every function whose address is to be resolved lazily, the global offset table entry for that function is initialized to be the address of the second part of the procedure linkage table entry for that function, that is `.L0` in the above example PLT entry.
2. The program requires to call the function *n*. This is performed by loading the arguments to *n* and making a PC-relative call to label `PLT`*n*. The calling function must ensure that R12 has been initialized to the address of the global offset table before making this call.
3. The code at `PLT`*n* loads the global offset table entry for *n* and calls it. As this global offset table entry has been initialized to label `.L0`, control is passed to this label.
4. The code at `.L0` calls `_dl_runtime_resolve`, passing `link_map` and `reloc_offset` as arguments. Note these arguments are passed in scratch registers rather than the normal argument passing registers, because the arguments to *n* are in the normal argument passing registers, and must not be overwritten.
5. `_dl_runtime_resolve` applies the relocation specified by `reloc_offset`. That is, it finds the value of symbol *n*, and stores the true address of *n* in the global offset table entry for *n*. It then transfers control to *n*.
6. Subsequent calls to *n* will avoid the runtime resolution overhead. As the global offset table entry for *n* has been overwritten with the true address of *n*, step *3* will transfer control directly to *n*.

Some variations on this sequence are possible. For example, the program may call *n* by loading the global offset table entry for *n* and calling it, rather than by calling `PLT`*n* at step *2*. The first time *n* is called, this results in step *3* being skipped and control is passed directly to `.L0`. On subsequent calls, this passes control directly to *n*.

3.2.1 Absolute procedure linkage table

An executable is often compiled as absolute code even though it uses position-independent shared libraries. The executable contains absolute calls to functions in the shared libraries, but the absolute addresses of these functions cannot be determined at the time of static linking. Instead, the absolute calls are relocated so that they call an “absolute” procedure linkage table entry. Absolute procedure linkage tables are part of the executable and have a known absolute address. They perform the same actions as position-independent procedure linkage tables, but they do not have a global context pointer available and so must use PC-relative or absolute addressing.

Figure 5 illustrates the code in an SH-4 absolute PLT entry. It is an example of how a PLT could be implemented, there is no requirement that this exact code sequence is used.

Figure 5. SH-4 absolute PLT entry

```
PLT0:  MOV.L  .L02,R0
       MOV.L  @R0,R0
       MOV.L  R0,@-R15
       MOV.L  .L01,R0
       MOV.L  @R0,R0
       JMP    @R0
       MOV.L  @R15+,R0
       NOP
       NOP
       NOP
.L01:  .long  address of GOT entry 2
.L02:  .long  address of GOT entry 1
...
PLTn:  MOV.L  .L2,R0
       MOV.L  @R0,R0
       MOV.L  .L1,R1
       JMP    @R0
       MOV    R1,R0
.L0:   MOV.L  .L3,R1
       JMP    @R0
       NOP
.L1:   .long  PLT0
.L2:   .long  address of GOT entry for n
.L3:   .long  reloc. offset
```

PLT0 is a special entry that does not correspond to any external function. It contains common code used by all the other PLT entries. PLT*n* is the PLT entry used to call the external function *n*.

The code beginning at label PLT*n* loads R1 with the address of PLT entry 0, then jumps to the address contained in the function’s entry in the global offset table. R1 is copied into R0 in the delay slot of the jump.

The code beginning at label .L0 loads R1 with the relocation offset of the function and then branches to the address in R0. This code is reached only reached after executing the code at PLT*n*, which has placed the address PLT0 in R0. So the branch always transfers control to PLT0.

The code beginning at PLT0 loads R0 with the value of the second global offset table entry, and then branches to the address in the third global offset table entry. R1 still contains the relocation offset of the function. Just as for the position-independent code, the second and third entries in the global offset table contain pointers to link_map and _dl_runtime_resolve. So the overall effect of this code is to call

```
_dl_runtime_resolve(&link_map, reloc_offset);
```

but again the function arguments are passed in R0 and R1, rather than the normal argument passing registers.

The sequence of events involved in lazy run-time symbol resolution when an absolute PLT entry is used is very similar to that for a position-independent PLT entry.

1. When the memory image of the process is created, entry 2 of the global offset table is initialized to be the address of the link_map data structure, and entry 3 of the global offset table is initialized to be the address of _dl_runtime_resolve, a function that performs run-time symbol resolution.

   Additionally, for every function whose address is to be resolved lazily, the global offset table entry for that function is initialized to be the address of the second part of the procedure linkage table entry for that function, that is .L0 in the above example PLT entry.
2. The program requires to call the function *n*. This is performed by loading the arguments to *n* and making an absolute call to label PLT*n*. R12 does not contain a global context pointer (executables containing absolute code do not maintain a global context pointer in R12).
3. The code at PLT*n* loads the global offset table entry for *n* and branches to it. As this global offset table entry has been initialized to label .L0, control is passed to this label.
4. The code at .L0 loads reloc_offset and then branches to the code at PLT0. The code at PLT0 calls _dl_runtime_resolve, passing link_map and reloc_offset as arguments. Note these arguments are passed in scratch registers rather than the normal argument passing registers, because the arguments to *n* are in the normal argument passing registers, and must not be overwritten.
5. _dl_runtime_resolve applies the relocation specified by reloc_offset. That is, it finds the value of symbol *n*, and stores the true address of *n* in the global offset table entry for *n*. It then transfers control to *n*.
6. Subsequent calls to *n* will avoid the runtime resolution overhead. As the global offset table entry for *n* has been overwritten with the true address of *n*, step *3* will transfer control directly to *n*.

3.3 Relocations involved in dynamic linking

Some relocation types have special semantics to support dynamic linking. *Table 7* lists the relocations and the value they calculate.

The calculation uses the following values:

A The relocation's addend.

B The base address at which a shared object has been loaded into memory. Normally, a shared object is build with a 0 base address, but the object's load address will be non-zero.

G The offset from the global context pointer to the global offset table entry for the symbol. The global context pointer may be the address of the first entry in the global offset table, or it may be biased from this address.

GOT The value of the global context pointer. The global context pointer may be the address of the first entry in the global offset table, or it may be biased from this address.

L The address of the procedure linkage table entry for the symbol.

P The address of the storage unit being relocated.

S The base address at which the shared object has been loaded into memory, plus the value of the relocation's symbol.

The values used for field are:

word32 A 32-bit longword.

word64 A 64-bit quadword.

T_*wwsaa*for*ll* An *ll*-bit wide field starting at the *aa*'th bit in a storage unit of width *ww*. *s* indicates the signedness: s means signed, u means unsigned. The value stored in the field is truncated to *ll* bits before storing. For example, T_32s10for16 means a signed field in bits 10 through 25 of a 32-bit longword: this matches the constant operand in an SHmedia **movi** instruction.

V_*wwsaa*for*ll* As for T_*wwsaa*for*ll* except that the value is verified to fit in the field before storage. If the value cannot fit an error should be reported.

Table 7. Relocations for position independent code

Name	Value	Field	Calculation
R_SH_GOT32	160	word32	G + A
R_SH_GOT_LOW16	169	T_32s10for16	(G + A) & 65535
R_SH_GOT_MEDLOW16	170	T_32u10for16	((G + A) >> 16) & 65535
R_SH_GOT_MEDHI16	171	T_32u10for16	((G + A) >> 32) & 65535
R_SH_GOT_HI16	172	T_32u10for16	((G + A) >> 48) & 65535
R_SH_GOT10BY4	189	V_32s10for10	(G + A) / 4
R_SH_GOT10BY8	191	V_32s10for10	(G + A) / 8
R_SH_PLT32	161	word32	L + A - P

Table 7. Relocations for position independent code (continued)

Name	Value	Field	Calculation
R_SH_PLT_LOW16	177	T_32s10for16	(L + A - P) & 65535
R_SH_PLT_MEWLOW16	178	T_32u10for16	((L + A - P) >> 16) & 65535
R_SH_PLT_MEDHI16	179	T_32u10for16	((L + A - P) >> 32) & 65535
R_SH_PLT_HI16	180	T_32u10for16	((L + A - P) >> 48) & 65535
R_SH_GOTPLT32	168	word32	G + A
R_SH_GOTPLT_LOW16	169	T_32s10for16	(G + A) & 65535
R_SH_GOTPLT_MEDLOW16	170	T_32u10for16	((G + A) >> 16) & 65535
R_SH_GOTPLT_MEDHI16	171	T_32u10for16	((G + A) >> 32) & 65535
R_SH_GOTPLT_HI16	172	T_32u10for16	((G + A) >> 48) & 65535
R_SH_GOTPLT10BY4	189	V_32s10for10	(G + A) / 4
R_SH_GOTPLT10BY8	191	V_32s10for10	(G + A) / 8
R_SH_GOTOFF	166	word32	S + A - GOT
R_SH_GOTOFF_LOW16	181	T_32s10for16	(S + A - GOT) & 65535
R_SH_GOTOFF_MEWLOW16	182	T_32u10for16	((S + A - GOT) >> 16) & 65535
R_SH_GOTOFF_MEDHI16	183	T_32u10for16	((S + A - GOT) >> 32) & 65535
R_SH_GOTOFF_HI16	184	T_32u10for16	((S + A - GOT) >> 48) & 65535
R_SH_GOTPC	167	word32	GOT + A - P
R_SH_GOTPC_LOW16	185	T_32s10for16	(GOT + A - P) & 65535
R_SH_GOTPC_MEDLOW16	186	T_32u10for16	((GOT + A - P) >> 16) & 65535
R_SH_GOTPC_MEDHI16	187	T_32u10for16	((GOT + A - P) >> 32) & 65535
R_SH_GOTPC_HI16	188	T_32u10for16	((GOT + A - P) >> 48) & 65535

Relocations that use GOT in their calculation cause the link editor to create a global offset table. Relocations that use G in their calculation cause the link editor to create a global offset table and add the referenced entry to it (if not already present). Relocations that use L in their calculation cause the link editor to create a procedure linkage table and a global offset table, and add the referenced entry to both, if not already present.

The link editor creates dynamic relocations to initialize each entry in the global offset table. If the entry is referenced by at least one R_SH_GOT*xxx* type relocation, then it will be initialized to the absolute address of the symbol. If the entry is only referenced by R_SH_PLT*xxx* or R_SH_GOTPLT*xxx* type relocations, then it will be initialized to the second part of the PLT entry for the symbol.

Although they calculate the same value, R_SH_GOT*xxx* and R_SH_GOTPLT*xxx* cause a different initial value to be placed in the global offset table entry. R_SH_GOTPLT*xxx* allows the global offset table entry to initially point to the PLT entry, allowing lazy run-time relocation of the symbol. Whereas R_SH_GOT*xxx* requires the global offset table entry to be initialized to the absolute address of the symbol, preventing lazy run-time relocation.

Table 8 lists the dynamic relocations created by the link editor and resolved by the dynamic linker, and the values calculated.

Table 8. Dynamic relocations for dynamic linking

Name	Value	Field	Calculation
R_SH_COPY	162	none	none
R_SH_COPY64	193	none	none
R_SH_GLOB_DAT	163	word32	S
R_SH_GLOB_DAT64	194	word64	S
R_SH_JMP_SLOT	164	word32	S
R_SH_JMP_SLOT64	195	word64	S
R_SH_RELATIVE	165	word32	B + A
R_SH_RELATIVE64	196	word64	B + A

In addition, the relocations listed in *Table 9* can appear as ordinary relocations in the input to the static linker, and be copied to the output as dynamic relocations that require resolution by the dynamic linker. R_SH_DIR32 and R_SH_64 should never appear as relocations on position-independent code, but they can appear as dynamic relocations on data.

Table 9. Additional dynamic relocations for dynamic linking

Name	Value	Field	Calculation
R_SH_DIR32	1	word32	S + A
R_SH_REL32	2	word32	S + A - P
R_SH_64	254	word64	S + A
R_SH_64_PCREL	255	word64	S + A - P

4 Coding examples

This section provides example coding sequences for various operations such as accessing static data, and calling functions. This section gives descriptions of how these operations could be performed, it does not require that they are performed in this way.

This section discusses both absolute and position-independent coding models.

In the **absolute** model, absolute addresses of code and data objects can be embedded in the program code, either as instruction operands or in tables in the text section. The program must be loaded at a specific address, to ensure that the absolute addresses contained in the program code are correct.

In the **position-independent** model, the program code may use relative addresses only. This means the address at which the program code and data may be loaded need not be specified until load time.

In general, it is possible to combine position-independent code with absolute code. The resulting combination is absolute code.

4.1 Position-independent code model

All absolute (virtual) addresses are held in a table called the global offset table. This table is contained in the data segment of a process. Each process has a private data segment, therefore each process has its own unique global offset table. The global context pointer, GP (R12), is used to point to the global offset table.

Note: R12 is a callee save register, and may be used for other purposes when access to the global offset table is not required.

To improve the efficiency of function calling, a procedure linkage table is also used. This table is contained in the text segment of a process. If an executable or shared object has position-independent code, then its procedure linkage table is also position independent, and can be shared by multiple processes. The procedure linkage table is normally accessed relative to the program counter (PC).

The following notation is used in position independent code examples.

`name@GOT`

This evaluates to the offset from the global context pointer to the entry for `name` in the global offset table.

This causes a global offset table to be created.

An entry for `name` in the global offset table is created if one does not already exist. At program load time, the entry for `name` in the global offset table is initialized to the absolute address of `name`.

`name@GOTPLT`

This evaluates to the offset from the global context pointer to the entry for `name` in the global offset table.

This causes a global offset table, and a procedure linkage table to be created.

	An entry for `name` in the global offset table is created if one does not already exist. At program load time, the entry for `name` in the global offset table will not necessarily contain the absolute address of `name`, it may instead contain the address of a fix up function that resolves the absolute address of `name`, updates the global offset table entry, and then passes control to `name`. An entry for `name` in the procedure linkage table is created if one does not already exist.
`name@GOTOFF`	This evaluates to the offset from the global context pointer to `name`. (Note that this is the offset to `name`, not to the global offset table entry for `name`.) This causes a global offset table to be created.
`name@PLT`	This evaluates to the offset from the current PC to the entry for `name` in the procedure linkage table. This causes a procedure linkage table and a global offset table to be created. An entry for `name` in the global offset table is created if one does not already exist. At program load time, the entry for `name` in the global offset table will not necessarily contain the absolute address of `name`, it may instead contain the address of a fix up function that resolves the absolute address of `name`, updates the global offset table entry, and then passes control to `name`. An entry for `name` in the procedure linkage table is created if one does not already exist.
`_GLOBAL_OFFSET_TABLE_`	This evaluates to the offset from the current PC to the global context pointer. The global context pointer may be the beginning of the global offset table, or it may be biased from the beginning of the global offset table. This causes a global offset table to be created.

4.2 Position-independent function prolog

If a position-independent function needs access to the global offset table, it must initialize the GP register (R12) to be the global context pointer. R12 is a callee-save register, so it must be saved to the stack before being initialized, and restored in the function epilog. As it is preserved across function calls, once R12 is initialized in the function prolog, it retains that value throughout the function body.

Figure 6 shows an SH-4 position-independent function prolog that saves R12 on the stack, sets up the global context pointer, and allocates 16 bytes for local variables in the stack frame.

Figure 6. SH-4 position-independent function prolog

```
fn: MOV.L  R12,@-R15
    MOVA   .L1,R0
    MOV.L  .L1,R12
    ADD    R0,R12
    ADD    #-16,R15
    ...
    .align 2
.L1:.long _GLOBAL_OFFSET_TABLE_
```

4.3 Data access

In the SH-4 architecture, memory is accessed using load and store instructions. These instructions cannot directly hold an absolute address, so a program normally computes an address into a register.

4.3.1 Absolute data access

Table 10 shows SH-4 code sequences for absolute load and store.

Table 10. SH-4 absolute load and store

C code	SH-4 assembly code

`extern int src;`
`extern int dst;`
`extern int *ptr;`	

`ptr = &dst;`	`MOV.L  .L2,R2`
	`MOV.L  .L3,R1`
	`MOV.L  R2,@R1`

Table 10. SH-4 absolute load and store (continued)

C code	SH-4 assembly code

`*ptr = src;`	`MOV.L .L4,R1`
	`MOV.L @R1,R1`
	`MOV.L .L3,R2`
	`MOV.L @R2,R2`
	`MOV.L R1,@R2`

	`...`
	`.align 2`
	`.L2:.long dst`
	`.L3:.long ptr`
	`.L4:.long src`

4.3.2 Position-independent data access

Position-independent code cannot contain absolute data addresses, instead the absolute data addresses are contained in the global offset table. A pointer to the global offset table is held in R12. Position-independent code accesses the global offset table entries using offsets from this global context pointer.

Table 11 shows SH-4 code sequences for position-independent load and store.

Table 11. SH-4 position-independent load and store

C code	Assembly code

`extern int src;`
`extern int dst;`
`extern int *ptr;`	

`ptr = &dst;`	`MOV.L .L4,R0`
	`MOV.L @(R0,R12),R2`
	`MOV.L .L5,R0`
	`MOV.L @(R0,R12),R1`
	`MOV.L R2,@R1`

`*ptr = src;`	`MOV.L .L6,R0`
	`MOV.L @(R0,R12),R1`
	`MOV.L @R1,R1`
	`MOV.L .L5,R0`
	`MOV.L @(R0,R12),R2`
	`MOV.L @R2,R2`
	`MOV.L R1,@R2`

	`...`
	`.align 2`
	`.L4:.long dst@GOT`
	`.L5:.long ptr@GOT`
	`.L6:.long src@GOT`

4.3.3 Position-independent static data access

Accesses to externally visible data must use the global offset table entry, because dynamic linking may bind the entry to a definition outside of the compilation unit containing the access.

However, it is possible to optimize position-independent accesses to static data that is not visible outside of the compilation unit that defines it. This data is at a known offset from the global context pointer, so may be accessed relative to R12.

Table 12 shows SH-4 code sequences for position-independent access to static data visible only in one compilation unit.

Table 12. SH-4 position-independent static data access

C code	Assembly code

`static int src;`
`static int dst;`
`static int *ptr;`	

`ptr = &dst;`	`MOV.L .L4,R2`
	`ADD R12,R2`
	`MOV.L .L5,R0`
	`MOV.L R2,@(R0,R12)`

`*ptr = src;`	`MOV.L .L6,R0`
	`MOV.L @(R0,R12),R1`
	`MOV.L .L5,R0`
	`MOV.L @(R0,R12),R2`
	`MOV.L R1,@R2`

	`...`
	`.align 2`
	`.L4:.long dst@GOTOFF`
	`.L5:.long ptr@GOTOFF`
	`.L6:.long src@GOTOFF`

4.4 Function calls

4.4.1 Absolute direct function call

On SH-4, absolute direct function calls may be made using BSR (if the target is within 4 Kbytes of the call) or JSR. *Table 13* shows SH-4 code sequences for absolute direct function calls.

Table 13. SH-4 absolute direct function call

C code	Assembly code
`extern void foo(void);`	

`foo();`	`foo` known to be within 4 Kbytes of call site
	`BSR foo`
	`NOP`

`foo();`	`foo` may be more than 4 Kbytes from call site
	`MOV.L .L2,R1`
	`JSR @R1`
	`NOP`
	`....`
	`.align 2`
	`.L2:.long foo`

4.4.2 Position-independent direct function call

Position-independent calls to static functions may be performed using a PC-relative call.

Position-independent calls to external functions are performed using the global offset table or the procedure linkage table. The address of the function is resolved by the dynamic linker.

Position-independent calls to externally visible functions must also be performed using the global offset table or the procedure linkage table, because dynamic linking may bind the name to a function outside of the compilation unit performing the call.

Calling using the procedure linkage table entry has the advantage that resolution of the function address can be delayed until the first time the function is actually called, rather than at the time the code is loaded. This reduces program load time. Calling the procedure linkage table also sometimes requires fewer instructions at the call site, leading to better code density. However, every call must also execute the instructions in the procedure linkage table entry, which means that more instructions are executed per call.

An alternative is to load the function address from the global offset table and then call it using an indirect call. This requires fewer instructions to be executed per call. Normally, this sequence would require the function address held in the global offset table to be resolved at load time, increasing program load time even when the function is never called. However, the notation `name@GOTPLT` can be used for direct calls, to indicate that lazy resolution can be used for `name`. This allows the global offset table entry for `name` to initially contain the address of the run-time address resolution function. The first time a call to `name` occurs, the run-time address resolution function is called instead. The run-time address resolution function resolves the address of `name`, updates the global offset table entry for `name`, and then calls `name`.

However, the compiler should not use the address loaded with `name@GOTPLT` for multiple calls to `name`, unless it can guarantee that `name` has been called previously. This is because

if `name` has not been called previously, the address loaded is the address of the run-time address resolution function, rather than the true address of `name`. On each call of that address, the run-time address resolution function is called to resolve the address of `name`. Although this operates correctly, it will be very slow.

Table 14 shows SH-4 code sequences for position-independent direct calls.

Table 14. SH-4 position-independent direct function call

C code	Assembly code
`static void sfoo(void);` `extern void efoo(void);`	

`sfoo();`	`MOV.L .L4,R1`
	`BSRF R1`
	`NOP`
	`.L0:`
	`...`
	`.align 2`
	`.L4:.long sfoo - .L0`

`efoo();`	Using procedure linkage table
	`MOV.L .L4,R1`
	`BSRF R1`
	`NOP`
	`.L0:`
	`...`
	`.align 2`
	`.L4:.long efoo@PLT+(.-.L0)`

`efoo();`	Using global offset table directly
	`MOV.L .L4,R0`
	`MOV.L @(R0,R12),R1`
	`JSR @R1`
	`NOP`
	`...`
	`.align 2`
	`.L4:.long efoo@GOTPLT`

4.4.3 Indirect function call

On SH-4, indirect function calls may be made using JSR. *Table 15* shows SH-4 code sequences for indirect function calls.

Table 15. SH-4 absolute indirect function call

C code	Assembly code
`extern void (*ptr)(void);`	

`(*ptr)();`	Absolute addressing:
	`MOV.L .L2,R1`
	`MOV.L @R1,R1`
	`JSR @R1`
	`NOP`
	`...`
	`.align 2`
	`.L2:.long ptr`

`(*ptr)();`	Position-independent addressing:
	`MOV.L .L4,R0`
	`MOV.L @(R0,R12),R1`
	`MOV.L @R1,R1`
	`JSR @R1`
	`NOP`
	`...`
	`.align 2`
	`.L4:.long ptr@GOT`

4.5 Branching

On SH-4, the conditional branch instructions, BF and BT, can be used to branch to a location 256 bytes in either direction. These instructions are PC-relative, and can be used for both absolute and position-independent code. For destinations that are more than 256 bytes away, it is possible to use a conditional branch over an unconditional branch.

The unconditional branch instruction, BRA, can be used to branch to a location 4 Kbytes in either direction. This instruction is PC-relative, and can be used for both absolute and position-independent code. For destinations that are more than 4 Kbytes away, JMP or BRAF may be used. JMP may only be used in absolute code. BRAF is PC-relative, and so may be used in position-independent code.

Table 16 shows some code sequences for branching on SH-4.

Table 16. SH-4 branching

C code	Assembly code
`label:` `...`	

`goto label;`	`label` within 4 Kbytes:
	`BRA label`
	`NOP`

`goto label;`	`label` further than 4 Kbytes away, absolute addressing
	`MOV.L .L4,R0`
	`JMP @R0`
	`NOP`
	`.align 2`
	`.L4:.long label`

`goto label;`	`label` further than 4 Kbytes away, position-independent addressing
	`MOV.L .L4,R0`
	`BRAF @R0`
	`NOP`
	`.L0:`
	`.align 2`
	`.L4:.long label-.L0`

5 Revision history

Table 17. Document revision history

Date	Revision	Changes
24-Apr-2005	A	Initial release.
18-Oct-2011	2	Minor changes throughout Added *Section 1.8: DWARF register assignments on page 10*. Corrected the value of “S” in *Section 3.3: Relocations involved in dynamic linking on page 25*.