Language Independent Application Binary Interface

The Language Independent ABI is intended to define the minimal conventions that must be used by all languages on the SH-5 architecture.

The SH-5 ISA comprises instructions in 2 modes:

- SHmedia - 32-bit instructions
- SHcompact - 16-bit instructions

defined in the *SH-5 Core Architecture* manuals.

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

Reference

Please refer to the *SH-5 CPU Core Architecture* manuals (05-CC-1000n) for further information.

Table of Contents

1 Introduction

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-5 architecture.

2 Language independent ABI

2.1 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.

An ABI is a set of trade-offs between many different possibilities, time or space, data size or code size, size limitations or no size limitations etc. This particular ABI is designed with the following guidelines in mind:

- The ABI should efficiently exploit the two modes, SHmedia and SHcompact, of the SH-5 architecture.
- Functions which are compiled in SHcompact should be able to call functions which are compiled in SHmedia and vice versa without the usage of wrappers.
- The code and data conventions in this ABI should not prevent the support of position independent code.
- The common cases should be treated most efficiently even if this causes inefficiency in less common cases.
- The ABI should not over specify the conventions.
- There should be no size limitations imposed other than those of the underlying hardware.
- The languages taken into account are C, C++, C9x, Java and SH-5 Assembler.

The ABI defines two addressing models known as the 32-bit ABI and the 64-bit ABI.

- In the 32-bit ABI (supporting ILP32), all addresses and offsets in addressing calculations are 32-bit width unsigned calculations.
- In the 64-bit ABI (supporting LP64), all addressing calculations are unsigned 64-bit operations. The 64-bit ABI is only supported in SHmedia since SHcompact does not support 64-bit addresses.

These two ABI's cause different code sequences and table sizes to be used for code addresses. The 32-bit ABI is designed to provide efficient support for programs designed to work in a 32-bit address space and to target limited address space implementations of SH-5. Compilers using these ABIs should be aware of which ABI is being used.

2.2 Definition of terms

In this section, we define the following terms used in this document.

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 is meant to cover C functions, C++ functions.

A **leaf function** is a function that statically makes no further calls to other functions.

A **frame** is the stack space pushed for a function invocation.

A **Compilation Unit** is 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.

An **external reference** is a reference from one compilation unit to an object defined outside the compilation unit.

Extended ASCII denotes the 8 bit ASCII character set which includes non English characters. The set is also known as Latin_8.

A **local reference** is a reference to an object within the same unit.

The **top of stack** is the lowest used address on the stack and usually corresponds to the most recent or current frame.

The **bottom of stack** is the highest used address in the stack and usually corresponds to the oldest frame on the stack.

Stack Unwind is 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(PIC) is code that can be loaded and will successfully execute anywhere in a program's virtual address space, i.e. the code contains no absolute code or data addresses.

2.3 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. SH-5 architecture supports both big-endian and little-endian byte ordering. The SH-5 ABI specification also supports both byte orderings.

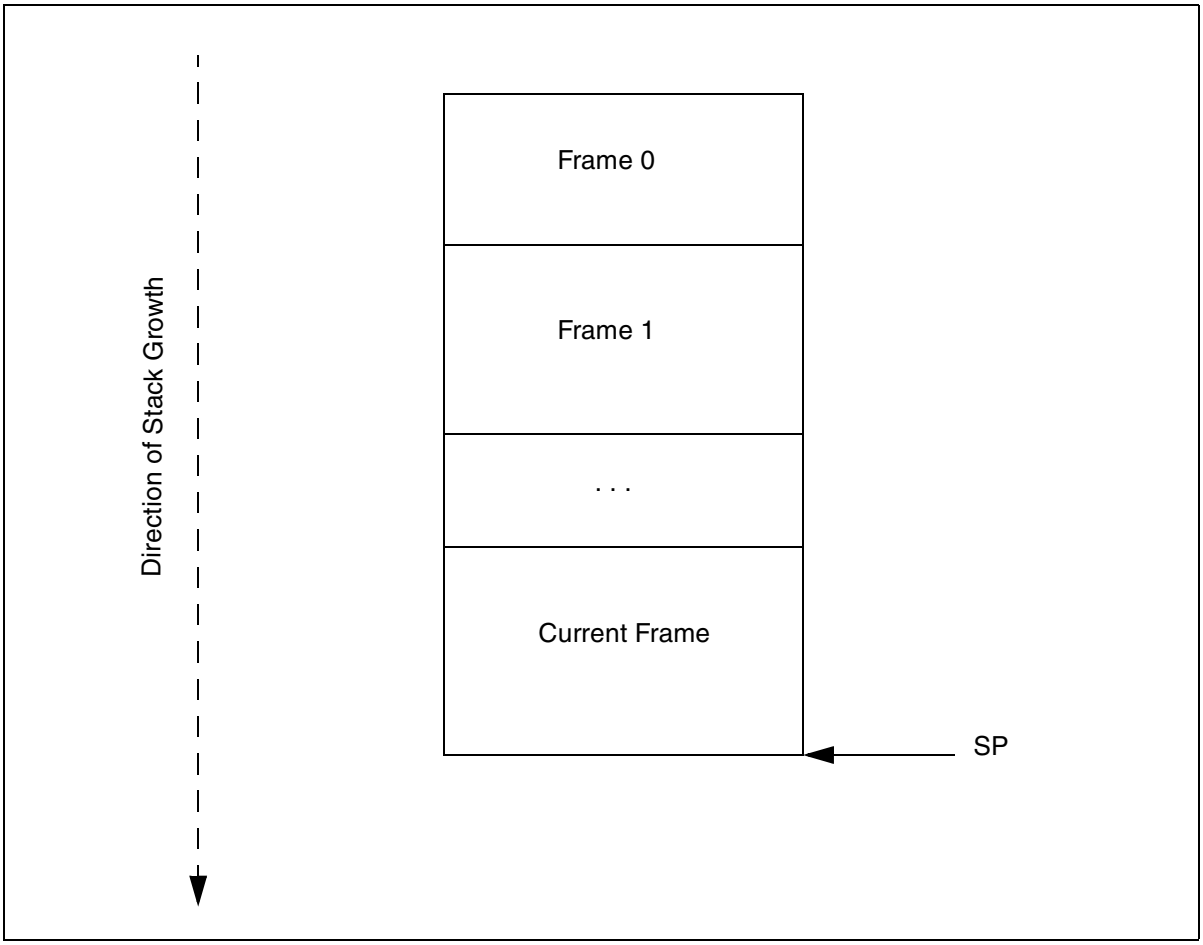

Figure 1: Overview of stack

2.4 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 SH-5 convention, the stack grows from high address to low address. The top of the stack (i.e. the smallest address) is always referenced by a register known as SP, the Stack Pointer. It is a requirement that this stack pointer be aligned on a 8 byte boundary on entry to a function due to the alignment requirement of the 64-bit register loads and stores. Within a function, the stack pointer is not necessarily 8 byte aligned at all times: for instance, during an SHcompact entry sequence, 4 byte register values may be pushed onto the stack one at a time. This means that an event handler cannot necessarily assume that SP is 8 byte aligned on entry to the handler.

A compiler implementation must not use the stack pointer register for any other purposes and at all times it must address the top of stack. 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 the value upon entry. Each function is responsible for creating and deleting its own frame. Not all functions will require a stack frame and a stack frame is allocated only if required. The stack growth is seen in *Figure 1: Overview of stack* on page 6.

As well as the stack pointer (SP), 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. Unlike the SP, FP is not a dedicated register.

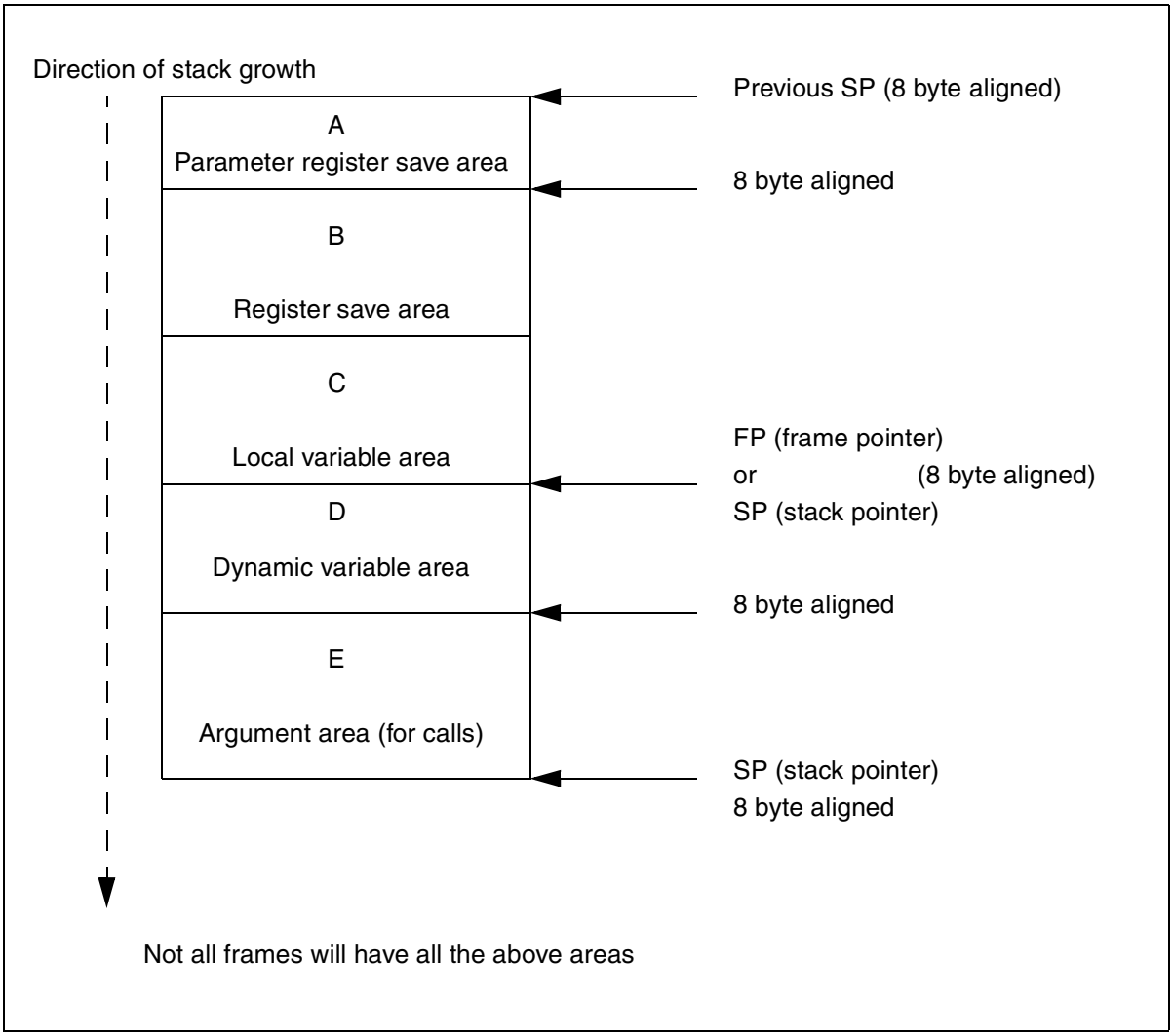

Figure 2: Frame layout

If a stack frame uses a frame pointer, the implementation may choose any suitable register for use as a frame pointer. 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 the frame.

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

2.5 Frame layout

The frame layout on SH-5 is described in *Figure 2: Frame layout* on page 7 .

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

We define the uses of the partitions of a frame as follows:

- **Parameter register save area (A)** is an area needed only when the called routine needs a memory copy of its parameters which are otherwise passed in registers. This spill area is located at the very start of the stack frame to make it contiguous with the remaining parameters (if any) which will reside in the caller's frame (in area E). This arrangement supports languages with variable length arguments such as C. Use of the dedicated parameter register save area is only necessary when an ordering relationship is required between memory copies of parameters such as is required in C.
- **Register save area (B)** is an area used to save and restore the callee save registers for this function. i.e, 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 SP or FP.
- **Dynamic variable area (D)** is an area 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. Most frames encountered in practice will not have any dynamic area. The objects in this area are addressed via pointers which reside in the local variable area. The existence of such an area implies the use of a Frame Pointer. Space in this area is created by decreasing SP which must be kept 8 byte aligned.
- **Argument area (E)** is an area needed to pass an argument list to functions called by this current function where the argument list is such that it cannot be accommodated in the parameter registers. The argument area will contain the remaining elements of the argument list after all the parameter registers have been used. Rather than allocating the exact area needed on each function call (by decreasing and increasing SP), the maximum area needed for calls from this function may be allocated on function entry. This reduces function call overhead by avoiding any further manipulations of SP, at the expense of allocating a larger stack frame throughout the lifetime of a function. The size of this area must be a multiple of 8 bytes in order to maintain the alignment of both the start and end of the area. If a function has both a dynamic variable area (D) and an argument area (E), 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 hence this means that dynamic memory allocations (and deallocations) must not happen during argument list building if the argument area (E) is also in use. In C for example, any alloca calls may have to be hoisted out of the argument list generation.

An additional frame pointer register (FP) may be allocated for a frame which addresses the local variable area of the stack frame. An FP is only required when a frame has a dynamic variable area which is dynamically allocated since SP can no longer be used to address the local variable area.

Leaf functions which do not make calls to other functions do not need to create a frame if no local data needs to be saved on the stack. Such leaf functions need not allocate any space on the stack if all their local variables and intermediate expressions can be allocated to scratch registers. In practice this means that all local variables can be allocated to the available scratch registers.

2.6 Global data

2.6.1 Variable data

The global data model is to have a single global data area in which all the global data from the compilation units is located. This global data area is addressed as follows:

- In SHmedia mode, the general purpose register R26 is used to point to the global data area. Where exactly in the global data area R26 will point is implementation dependent.

  All implementations should define a symbol named ___DATA that labels the address pointed to by R26. (Note that there are exactly three leading underscore characters in the name ___DATA.)

Global data located near the address in R26 may be accessed efficiently by using R26 as a base register in an indirect memory access. However, an implementation is not required to access global data relative to R26; for instance, absolute addressing schemes could be used.

- In SHcompact mode, global variable data access is implementation dependent. For instance, absolute addressing schemes could be used.

2.6.2 Constant data

Constant data is located in a single read-only section addressed as follows:

- In SHmedia mode, the general purpose register R27 is used to point to the constant data area. Where exactly in the constant data area R27 will point is implementation dependent.

  All implementations should define a symbol named ___RODATA that labels the address pointed to by R27. (There are exactly three leading underscore characters in the name ___RODATA.)

  Constant data located near the address in R27 may be accessed efficiently by using R27 as a base register in an indirect memory access. However, an implementation is not required to access all constant data relative to R27; for instance, absolute addressing schemes could be used.

- In SHcompact mode, global constant access is implementation dependent. For instance, absolute addressing schemes could be used.

2.7 Function linkage and parameter passing

2.7.1 Function linkage

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

- In SHmedia mode the general purpose register R18 is used.
- In SHcompact mode the PR register is used.

If the environment fabricates the value in the linkage register, for example to mark the outermost function of a thread, then it must ensure that the fabricated value does not generate an IADDERR exception when used as the target address in a PTABS instruction. This allows a compiler to move a function exit PTABS instruction forward in the instruction stream to a position where a corresponding branch instruction (the function exit) may not necessarily be executed.

2.7.2 Parameter passing

The term ‘parameter passing convention’ refers to how the actual machine resources (registers, memory for example) are used to pass the parameters from the caller to the callee. The abstraction of an argument list makes it easier to specify the parameter passing convention. The parameter passing is described in terms of two mappings, the first from the function call to an argument list and the second from the argument list to the machine resources.

The notion of an argument list is independent of the language semantics. An argument list is an ordered list of argument elements, each element is a 64-bit quantity and is 64-bit aligned. The actual mapping from arguments in the program to the argument list is dependent on the language semantics. Argument elements may be scalar values, floating point values, pointers, aggregate types etc. and a given language may map a given type to one or more argument list elements.

The mapping from an argument list to machine resources is dependent on the type associated with the elements of the list and hence is language dependent. *Section 3: ANSI C ABI* on page 14 describes the details of these two mappings for ANSI C.

2.7.3 Register usage conventions

We define the following terms used 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 SH-5 architecture provides

- 64 general purpose registers (GPR), R0-R63, each 64-bit wide in SHmedia mode. Only registers R0-R15 are visible in SHcompact mode.
- 64 floating point registers, FR0-FR63, each 32-bit wide in SHmedia mode. Only registers FR0-FR31 are visible in SHcompact mode and these are presented as two banks of 16 registers each.
- 8 target registers, TR0-TR7, used for branching only visible in SHmedia mode.

See the architecture manual for details of the register set. The SH-5 register classification is designed to exploit the hardware context switching optimization. The register usage for SHmedia is given in *Table 1 on page 11*. A subset of the callee-save registers, R10-R14, is visible in SHcompact as well as SHmedia. However, in SHcompact only the lower 32 bits are visible and therefore only the lower 32 bits of the registers R10-R14 are guaranteed to be saved and restored by the callee.

The SHcompact registers correspond to a subset of SHmedia registers and the register convention for SHcompact is given by SHcompact to SHmedia mapping shown in *Table 2 on page 12* and SHmedia register convention shown in *Table 1 on page 11*.

In SHcompact the SZ, FR, and PR bits in the status register in the default case should be initialized to zero and must be zero upon entry to prolog and exit from epilog of any function. However command line options could determine the initial values of the above fields and the initial values should be maintained upon entry to prolog and exit from epilog of any function.

In SHcompact, the S, M and Q bits in the status register have undefined values on entry to a function, and their values are not guaranteed to be preserved across calls.

R16 (GBR in SHcompact) is specified as a reserved register: it is neither caller nor callee save. This ABI does not place any further rules upon the use of this register. This is intended to allow the various usage models for GBR on SH-1 through SH-4 to continue to work for SHcompact code.

R25 is reserved for use by the linker to fix up relocations. It should not be used by user programs except when this is absolutely necessary, for example when saving and restoring context in event handlers. The compiler and assembler should ensure that R25 is not live across a relocation field, so that the linker is able to use it as a temporary register when fixing up relocation (unless the compiler/assembler can deduce that R25 will not be needed to fix up the relocation).

See *Section A.4: Usage of R25* on page 44 for an example illustrating the usage of R25.

Register name	Usage
R0-R1	Caller save
R2	Return value, caller save
R2-R9	Parameter passing, caller save
R10-R14	Callee save, the lower 32 bits of the registers R10-R14 are guaranteed to be saved and restored by the callee. The upper 32 bits are guaranteed to be preserved if and only if they are a correct sign extension of bit 31.
R15	Stack pointer, SP, callee save
R16	Reserved
R17	Caller save
R18	Linkage register, caller save
R19-R23	Caller save
R24	Reserved for use by the operating system
R25	Reserved for assembler/linker
R26	Global variable data pointer, reserved
R27	Global constant data pointer, reserved
R28-R35	Callee save
R36-R43	Caller save
R44-R59	Callee save
R60-R62	Caller save
R63	Value 0 always
FR0-FR1	Return value, caller save
FR0-FR11	Parameter passing, caller save
FR12-FR15	Callee save
FR16-FR35	Caller save
FR36-FR63	Callee save
SR	Status register - SZ, FR, and PR bits must be zero upon entry to prolog and exit from epilog
TR0-TR4	Caller save
TR5-TR7	Callee save

Table 1: SHmedia registers

SHcompact register	SHmedia register
R0 - R15	R0 - R15
GBR	R16
MACL	R17 (lower 32 bits)
MACH	R17 (upper 32 bits)
PR	R18
T-bit	Bit 0 of R19
FR0-FR15	if FPSCR.FR=0 then FR0-FR15 else FR16-FR31
XF0-XF15	if FPSCR.FR = 0 then FR16-FR31 else FR0-FR15
FPUL	FR32

Table 2: Mapping of SHcompact registers to SHmedia registers

In SHcompact mode the T-bit is mapped to R19, but a read of the T-bit is only defined if R19 contains the value 0 or 1. On entry to an SHcompact function, there is no guarantee that R19 contains either 0 or 1, so care must be taken not to read the T-bit until there has been a write to the T-bit.

The following SHcompact instructions read the T-bit: an SHcompact function should not execute any of these instructions until the T-bit has been initialized:

```
ADDC Rm,Rn
BF label
BF/S label
BT label
BT/S label
DIV1 Rm,Rn
MOVT Rn
NEGC Rm,Rn
ROTCL Rn
ROTCR Rn
SUBC Rm,Rn
```

2.8 Function prolog and epilog

The entry-point to a function must be 4 byte aligned (this holds even for an entry-point in SHcompact code, as the SHmedia PTB instruction can only reach 4 byte aligned instructions).

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. Such a specification would be unnecessary and would be difficult given the instruction level scheduling that a language processor may apply. Instead, function prologs and epilogs are characterized by a set of tasks which are carried out.

On entry to a function, the following tasks are performed:

- (Optional) Create a stack frame. This is performed by decreasing SP. No accesses beyond SP are permitted.
- (Optional) Create a working register set. A function always has access to a set of scratch registers. If it needs further registers, it must save and use registers in the general purpose register set.
- (Optional) Save the return address.

On exit from a function, the following tasks are performed:

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

3 ANSI C ABI

This section covers the run-time model for the implementation of ANSI C on SH-5 based on the language independent ABI.

3.1 Built-in type mapping

The ABI specifies 2 predefined type mappings for C.

- sizeof(int) = sizeof(long) = sizeof(char *) = 4,
  sizeof(long long) = 8
- sizeof(int) = 4,
  sizeof(long) = sizeof(char*) = 8

The first mapping is known as the 32-bit ABI and models a 32-bit environment on the 64-bit SH-5. Pointers are 32-bit unsigned quantities employing modulo 32-bit unsigned arithmetic. A major purpose of the 32-bit ABI is to provide an easy portability path for software developed on 32-bit processors.

The second mapping is known as the 64-bit ABI and is suitable for a SH-5 with a full 64-bit address range. This is also the model used by some other 64-bit processors. 64-bit ABI is not applicable to SHcompact since SHcompact does not support 64-bit addresses.

The 32-bit ABI and the 64-bit ABI are incompatible. Code built using either the 32-bit or the 64-bit ABI cannot be mixed.

The definition of the 32-bit ABI is the following.

ANSI C types	Byte alignment	32-bit SH-5 types
char signed char unsigned char	1	1-byte signed integer 1-byte signed integer 1-byte unsigned integer
short int (signed) unsigned short int	2	2-byte signed integer 2-byte unsigned integer
int (signed) unsigned int enum	4	4-byte signed integer 4-byte unsigned integer 4-byte signed integer
long int (signed) unsigned long int	4	4-byte signed integer 4-byte unsigned integer
long long int (signed) unsigned long long int	8	8-byte signed integer 8-byte unsigned integer
float	4	4-byte single-precision floating point
double long double	8	8-byte double-precision floating point 8-byte double-precision floating point
pointer	4	4-byte unsigned integer

Table 3: Mapping of ANSI C data types to 32-bit ABI

The definition of the 64-bit ABI is the following:

ANSI C TYPES	BYTE ALIGNMENT	64-BIT SH-5 TYPES
char signed char unsigned char	1	1-byte signed integer 1-byte signed integer 1-byte unsigned integer
short int (signed) unsigned short int	2	2-byte signed integer 2-byte unsigned integer
int (signed) unsigned int enum	4	4-byte signed integer 4-byte unsigned integer 4-byte signed integer
long int (signed) unsigned long int long long int unsigned long long int	8	8-byte signed integer 8-byte unsigned integer 8-byte signed integer 8-byte unsigned integer
float double	4 8	4-byte single-precision floating point 8-byte double-precision floating point
long double	8	8-byte double-precision floating point
pointer	8	8-byte unsigned integer

Table 4: Mapping of ANSI C data types in the 64-bit ABI

3.2 Type mapping and alignment

3.2.1 Scalar types

SH-5 has what is usually termed natural alignment where the alignment constraint is the same as the size of the type. The compiler can rearrange objects in a function frame to minimize the padding needed to preserve the alignments.

3.2.1.1 Function pointers

Bit 0 of a function pointer encodes the ISA in which the function should be entered: if bit 0 is one, then the function should be entered in SHmedia mode; if bit 0 is zero, then the function should be entered in SHcompact mode. The remaining (more significant) bits of the function contain an address to call to enter the function.

3.2.2 Aggregate types

Arrays of types inherit the alignment of the components of the array and each element will be correctly aligned, that is an array will have the alignment of the components. 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 component. The compiler inserts padding to maintain the alignment of internal components. The contents of any padding is undefined. The sizeof 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 and the 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 component will also not affect the layout of the record.

Except when they are parameters, array variables and structure variables of size greater than 4 bytes are aligned on a 8 byte boundary and this enables the compiler (and library) to generate efficient code for array copying and structure copying. Array and structure parameters are laid out using the rules in *Section 3.3.2: Argument passing* on page 18. A distinction is being made between a structure variable and a structure element of an array. Consider **`struct foo {int a,b,c;} x`** and **`struct foo y[10]`**. The variable **`x`** as well as the array **`y`** are aligned on a 8 byte boundary where as an element **`y[1]`** or **`y[3]`** is only aligned on a 4 byte boundary.

3.2.2.1 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. We follow the MS Visual C++ convention for allocation of bit-fields within a structure.

Bit-fields may be of any integral type (char, short, int, long long) 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.

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

- A bit-field never crosses a storage boundary whose alignment is same as the alignment of the underlying type of the bit field. In other words, a bit-field never straddles across the natural boundary of the underlying type.
- A bit-field shares a storage unit with the previous structure member if and only if
  - ➢ the previous member is also a bit-field,
  - ➢ the size of the type of the previous member is same as the size of the type of the current bit-field, and
  - ➢ there is sufficient space within the storage unit.
- Within a storage unit, bit-fields are allocated from right to left (least to most significant) on little-endian implementations and from left to right (most to least significant) on big-endian implementations.
- Plain bit-fields are treated as signed.
- A zero-length bit-field has effect only when it follows a bit-field of non-zero-length.
- The effect of a zero-length bit-field is to:

1 pad up to the alignment of the underlying type of the zero-length bit-field,

2 force the next (non-zero-length) field to be allocated in a new storage unit, and

3 force the overall alignment of the structure to be at least the alignment of the underlying type of the zero-length bit-field.

- Unnamed bit-fields do affect the overall alignment of a struct.

Example:

```
struct {
   int a:9;
   unsigned long b:4;
   int :0;
   int c:7;
   int :25;
   int d:9;
   char e;
   int f:5;
}
```

Figures *3* and *4* show the layout of this bit-field in big-endian and little-endian byte ordering. 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. The size of the structure is 20 bytes in either case. .

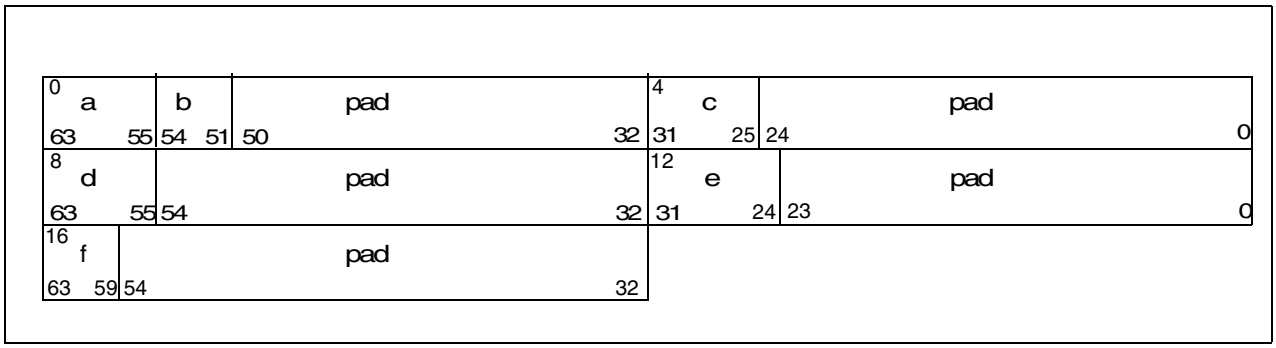

Figure 3: Big-endian layout

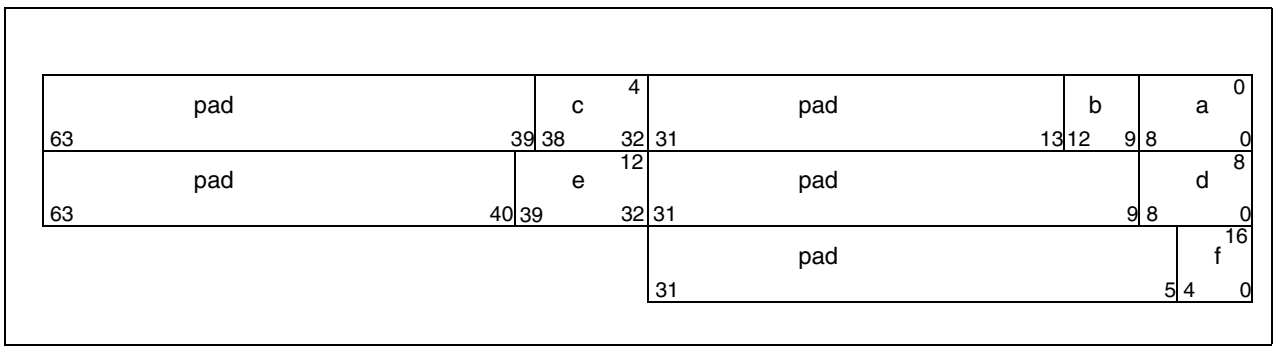

Figure 4: Little-endian layout

3.3 Function results and argument passing

3.3.1 Function results

The following rules specify how function values are returned, based on the type of the return value:

- Integer scalar types and aggregate types which are not bigger than 8 bytes are returned in general purpose register R2. Integer scalar types returned in R2 are extended to 8 bytes based on the type, according to the convention of data representation in registers described in the architecture manual. Aggregate types returned in R2 that are smaller than 8 bytes are padded at the most significant end: the value of the padding is undefined.
- Function results of structure types which are bigger than 8 bytes are returned by address. The caller function passes the address of the result destination as an implicit extra parameter in register R2. The called function stores the result in this area and returns the address of the area as its result in register R2. As the extra implicit parameter is passed in register R2, this value can act as the function result without modification.
- Single precision floating-point values are returned in FR0.
- Double precision floating-point values are returned in FR0,FR1 pair (DR0).

3.3.2 Argument passing

Following the argument-list parameter passing abstraction of the generic ABI, we define parameter passing in two steps:

- Mapping of actual parameters to the argument list
- Mapping of the argument list to the processor registers and memory.

Scalar and aggregate types only are mentioned since array type parameters are treated as pointer (scalar) types following the rule in 6.7.1:7 of the ISO C standard (9899:1990).

3.3.2.1 Actual parameter to argument list mapping

Actual parameters are processed in lexical order and are mapped to contiguous elements of the argument list. 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.

- Each scalar actual parameter is mapped to a single element of the argument list. The type associated with the element is same as the type associated with the parameter, after type conversion if necessary. The code generator shall generate the following type conversions 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`**.
- The mapping of an aggregate parameter is determined by the memory layout of that parameter. Successive 8 bytes of an aggregate parameter are mapped to successive elements of the argument list. The argument list elements are typed as non-scalar elements.
- When the size of an aggregate parameter is not a multiple of 8 bytes, the last element is padded. The value of the padding is undefined. If the size of the parameter is less than 8 bytes, that is, it is contained in a single element, then it is always padded at the most significant end. If the size of the parameter is greater than 8 bytes, then the padding depends on target endianness: for little-endian targets the element is padded at the most significant end, for big-endian targets the element is padded at the least significant end.

3.3.2.2 Argument list to processor mapping

The mapping from argument list to machine resources can be described in terms of three possible cases, namely

- callee known to be a non-variable-arguments function,
- callee known to be a variable-arguments function, and
- callee prototype not known.

If a scalar parameter with a size which is less than or equal to 4 bytes is passed in a register, then the parameter is extended to 8 bytes based on the type of the parameter (after conversion if necessary), according to the convention of data representation in registers described in the architecture manual. If the parameter is passed in a stack location then it is extended to 4 bytes only: the upper 4 bytes of the 8 bytes allocated on the stack are undefined.

Callee known to be a non-variable-arguments function

- Process argument list elements in order.
- Use R2-R9 for integer and integer-equivalent (**`char`**, pointer) and non-scalar (**`struct`**, **`union`**) elements.
- Use FR0-FR11 for single precision (**`float`**) floating-point elements.
- Use DR0-DR10 for double precision (**`double`**) floating-point elements.
- Each floating-point element will be mapped to the lowest numbered register available of the type of the element. Single precision element maps to the next available FR register. Double precision element maps to the next available DR register. For each floating-point element i which will be mapped on a FR or DR register then either the corresponding GPR(R2+i) will be unoccupied if 0<= i < 8, or the next available quad-word (64-bit) stack location will be unoccupied if i >=8. If there are no floating-point registers available then the next available GPR or stack location is used. Leaving the GPR or stack location unused makes it possible to have a simple consistent scheme that will work when a function is being called from two different contexts, in one context the function is known to be a non-variable-arguments function and in the other context there is no prototype given.
- Argument list elements not passed in registers are passed on stack in the argument area (E) at the next available quad-word (64-bit), starting from the lowest address in the argument area.

Example:

```
typedef struct s_point {
  float x, y, z;
} 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);
```

Argument	Machine Resource
p1.x, p1.y	R2
p1.z	R3
f1	FR0
d1	DR2
f2	FR1
p2.x, p2.y	R7

Table 5: Arguments to machine resources - callee non-variable-arguments function

Argument	Machine Resource
p2.z	R8
p3.x, p3.y	R9
p3.z	(0,SP)
f3	FR4
d2	DR6

Table 5: Arguments to machine resources - callee non-variable-arguments function

Note: R4,R5,R6 and FR5 are unused.

A feature of this parameter passing model is that aggregate values may be passed in registers if mapped to the first 8 argument list elements. Also, aggregate values may be passed partially in registers with the remaining fields passed on the stack.

Compilers may discover and exploit the properties of individual function calls (for example, through the use of prototypes), and thereby modify the above mapping of aggregate values to registers and stack; however the program behavior must appear as if the above mapping was applied uniformly.

Callee known to be a variable-arguments function

- The calling conventions of *Callee known to be a non-variable-arguments function on page 19* apply up until the first argument corresponding to where the ellipsis occurs in the parameter list of the callee.
- For the rest of the elements in the argument list, no element is passed in FR or DR registers. Floating-point arguments are passed in R2-R9 and the argument area in the stack, just like the other arguments. Note that arguments of type **`char`**, **`short`**, **`unsigned char`** and **`unsigned short`** are converted to **`int`**, and arguments of type **`float`** are converted to **`double`**.

Example:

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

Argument	Machine Resource
p1.x, p1.y	R2
p1.z	R3
f1	FR0
d1	R5
f2	R6
p2.x, p2.y	R7
p2.z	R8
p3.x, p3.y	R9
p3.z	(0,SP)
f3	(8,SP)

Table 6: Arguments to machine resources - callee variable-arguments function

Argument	Machine Resource
d2	(16,SP)

Table 6: Arguments to machine resources - callee variable-arguments function

Callee prototype not known

- Process argument list elements in order.
- Use R2-R9 for integer and integer-equivalent (**`char`**, pointer) and non-scalar (**`struct`**, **`union`**) elements.
- There will be no single-precision (**`float`**) floating-point elements, because when the prototype is not known, all arguments of type **`float`** are converted to type **`double`**.
- Use DR0-DR10 for double precision (**`double`**) floating-point elements.
- Each floating-point element will be mapped to the lowest numbered DR (double-precision) register available. For each floating-point element i which will be mapped on a DR register then either the floating-point element i is passed also in the corresponding GPR(R2+i) if 0<= i < 8, or the floating-point element i is passed also in the next available quad-word (64-bit) stack location if i >=8.
  - ➢ If the callee is a non-variable-arguments function, it will use the floating-point register copy.
  - ➢ If the callee is a variable-arguments function, it will use the GPR copy or from stack as the case may be.

  These rules will work only if the callee does not expect **`float`** type parameters, or if the callee is a K&R function. Even if the callee is a function that takes variable arguments, a **`float`** parameter preceding the ellipsis will be received as a **`float`** by the callee, and hence the argument converted to **`double`** by the caller would not be received by the callee correctly.
- Argument list elements not passed in registers are passed on stack in the argument area (E) at the next available quad-word (64-bit), starting from the lowest address in the argument area.

Example:

```
typedef struct s_point {
  float x, y, z;
} point;
/* prototype of foo unknown, but type of p1,f1, etc. as before*/
foo(p1,f1,d1,f2,p2,p3,f3,d2);
```

Argument	Machine Resource
p1.x, p1.y	R2
p1.z	R3
f1	DR0 and R4
d1	DR2 and R5
f2	DR4 and R6
p2.x, p2.y	R7
p2.z	R8
p3.x, p3.y	R9
p3.z	(0,SP)

Table 7: Arguments to machine resources - unknown prototype

Argument	Machine Resource
f3	DR6 and (8,SP)
d2	DR8 and (16, SP)

Table 7: Arguments to machine resources - unknown prototype

Note: For this call to work correctly, the definition of `foo` *must not receive floating-point parameters as type float. The following definitions of* `foo` *would work:*

```
int foo (p1, f1, d1, f2, p2, p3, f3, d2)
point p1,p2,p3;
float f1,f2, f3;
double d1,d2;
{...}
```

or

```
int foo (point p1, double f1, double d1, double f2, point p2, point p3, double
f3, double d2)
{...}
```

but the prototype for `foo` given in *Callee known to be a non-variable-arguments function* would not work correctly as it receives `f1`, `f2` and `f3` as `float`, and the prototype for `foo` given in *Callee known to be a variable-arguments function* would not work correctly as it receives `f1` as `float`.

Further parameter passing examples appear in *Section A.2: Parameter passing - further examples* on page 38.

Implementation of argument list to processor mapping

The SHmedia instruction set of the SH-5 architecture provides support for mapping argument list elements which are 8 bytes long to machine resources.

However, the SHcompact instruction set of the SH-5 architecture does not provide support for this mapping.

- In SHcompact mode, only the lower 32 bits of a register can be read. Furthermore, some SHcompact instructions require the upper 32 bits of a source operand to be a sign extension of bit 31.
- In SHcompact mode, the lower 32 bits of a register can be written. However, some SHcompact instructions will also overwrite the upper 32 bits of the destination register as a sign extension of bit 31. Care must be taken when using these instructions that the upper 32 bits do not contain meaningful values.

Note: In particular that aggregate, double precision, and long long parameters that are passed in general purpose registers may use all 64 bits of the register: i.e. the upper 32 bits contain meaningful values, and they are not a sign extension of bit 31. Therefore care must be taken when accessing these parameters in SHcompact mode. Where necessary, the mapping of 8 byte long argument list elements to machine resources may be achieved by switching to SHmedia, performing the mapping and then switching back to SHcompact. Section A.1: Passing 64-bit parameters on page 33 gives an example implementation of this mapping.

3.3.2.3 Handling of variable-arguments function by the callee.

When the call is to a function with a variable number of arguments, the caller will pass the arguments in accordance with the rules outlined above. The called routine will allocate space and copy the registers R2-R9 to its own stack space (A). Because the function cannot tell in advance how many of the machine registers may be in use, it must save all the potential parameter registers to the stack. The implementation of the variable argument manipulation macros will use the memory copies of the parameters. The **va_arg** macro will address the parameters as an array indexed by the implicit counter to **va_arg**. The **va_start** macro is implemented by initializing the variable argument pointer with the address of the argument in the argument list which is the first variant argument. This is the address of the first parameter available through **va_arg**. Consequently, **va_arg** is implemented as returning the value at this pointer followed by an increase to address the next parameter in the variable argument list. **va_arg** must take into account padding inserted when the parameter is not an exact multiple of 8 bytes in length. All arguments smaller than 8 bytes are padded at the most significant end; arguments larger than 8 bytes are padded in the last argument element.

va_end serves no purpose except to make the variable argument unusable as a legal pointer.

Section A.3: Implementation of STDARG.H on page 43 gives an example implementation of the **va_start**, **va_arg** and **va_end** macros.

3.4 Symbol names

C identifiers which are used as global symbol names in the resulting object file must be prepended with an underscore. For example, the entry point of the function **foo()** will be represented by the symbol **_foo** in the object file.

3.5 Intrinsic functions

Compilers that provide intrinsic C functions to support specific SHmedia machine instructions should name these functions **sh_media_** followed by the instruction name in upper case. Dots in the instruction name should be replaced with underscores in the intrinsic name. For example, the intrinsic to support **MCNVS.LW** should be named

```
sh_media_MCNVS_LW
```

The intrinsic functions should be declared in the following header files:

ushmedia.h Intrinsics corresponding to SHmedia instructions that may be executed in both user and privileged mode (i.e. all the intrinsics described below except those in *Section 3.5.3*)

sshmedia.h Intrinsics corresponding to SHmedia instructions that may only be executed in privileged mode (i.e. the intrinsics described in *Section 3.5.3*)

shmedia.h All SHmedia instriniscs (may simply include both **ushmedia.h** and **sshmedia.h**).

The following sections list the recommended set of intrinsic functions.

Note: Some SHmedia instructions read and write their third operand, for example, FMAC.S reads and then writes its third operand. The intrinsics that correspond to these instructions do not overwrite the third operand, as the result is returned as the function value.

For example, the behavior of the FMAC.S intrinsic,

```
sh_media_FMAC_S(float fg, float fh, float fq)
```

is

```
return fq + fg * fh;
```

that is, `fq` is not overwritten by the intrinsic function.

The behavior of the SHmedia instruction can be obtained by assigning the result returned by the intrinsic to the same variable that is passed to the third parameter, for example,

```
v = sh_media_FMAC_S(fg, fh, v);
```

The instructions that have operands that behave in this way are: CMVEQ, CMVNE, FMAC.S, MCMV, MMACFX.WL, MMACNFX.WL, MMULSUM.WQ, MSAD.UBQ.

3.5.1 Multimedia instructions

```
__inline__ unsigned long long sh_media_MABS_L(unsigned long long mm)
__inline__ unsigned long long sh_media_MABS_W(unsigned long long mm)
__inline__ unsigned long long sh_media_MADD_L(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MADD_W(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MADDS_L(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MADDS_UB(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MADDS_W(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MCMPEQ_B(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MCMPEQ_L(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MCMPEQ_W(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MCMPGT_L(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MCMPGT_UB(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MCMPGT_W(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MCMV(unsigned long long mm, unsigned long
long mn, unsigned long long mw)
__inline__ unsigned long long sh_media_MCNVS_LW(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MCNVS_WB(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MCNVS_WUB(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MEXTR1(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MEXTR2(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MEXTR3(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MEXTR4(unsigned long long mm, unsigned
long long mn
)__inline__ unsigned long long sh_media_MEXTR5(unsigned long long mm, unsigned
long long mn)
```

```
__inline__ unsigned long long sh_media_MEXTR6(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MEXTR7(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MMACFX_WL(unsigned long long mm, unsigned
long long mn, unsigned long long mw)
__inline__ unsigned long long sh_media_MMACNFX_WL(unsigned long long mm,
unsigned long long mn, unsigned long long mw)
__inline__ unsigned long long sh_media_MMUL_L(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MMUL_W(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MMULFX_L(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MMULFX_W(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MMULFXRP_W(unsigned long long mm,
unsigned long long mn)
__inline__ unsigned long long sh_media_MMULHI_WL(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MMULLO_WL(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MMULSUM_WQ(unsigned long long mm,
unsigned long long mn, unsigned long long mw)
__inline__ unsigned long long sh_media_MPERM_W(unsigned long long mm, unsigned
int mn)
__inline__ unsigned long long sh_media_MSAD_UBQ(unsigned long long mm, unsigned
long long mn, unsigned long long mw)
__inline__ unsigned long long sh_media_MSHALDS_L(unsigned long long mm, unsigned
int mn)
__inline__ unsigned long long sh_media_MSHALDS_W(unsigned long long mm, unsigned
int mn)
__inline__ unsigned long long sh_media_MSHARD_L(unsigned long long mm, unsigned
int mn)
__inline__ unsigned long long sh_media_MSHARD_W(unsigned long long mm, unsigned
int mn)
__inline__ short sh_media_MSHARDS_Q(long long mm, unsigned int mn)
__inline__ unsigned long long sh_media_MSHFHI_B(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MSHFHI_L(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MSHFHI_W(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MSHFLO_B(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MSHFLO_L(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MSHFLO_W(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MSHLLD_L(unsigned long long mm, unsigned
int mn)
__inline__ unsigned long long sh_media_MSHLLD_W(unsigned long long mm, unsigned
int mn)
__inline__ unsigned long long sh_media_MSHLRD_L(unsigned long long mm, unsigned
int mn)
__inline__ unsigned long long sh_media_MSHLRD_W(unsigned long long mm, unsigned
int mn)
__inline__ unsigned long long sh_media_MSUB_L(unsigned long long mm, unsigned
long long mn)
```

```
__inline__ unsigned long long sh_media_MSUB_W(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MSUBS_L(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MSUBS_UB(unsigned long long mm, unsigned
long long mn)
__inline__ unsigned long long sh_media_MSUBS_W(unsigned long long mm, unsigned
long long mn)
```

3.5.2 Floating-point instructions

```
__inline__ double sh_media_FABS_D(double dg)
__inline__ float  sh_media_FABS_S(float fg)
__inline__ int    sh_media_FCMPUN_D(double dg, double dh)
__inline__ int    sh_media_FCMPUN_S(float fg, float fh)
__inline__ float sh_media_FCOSA_S(float fg)
__inline__ float  sh_media_FGETSCR(void)
__inline__ float  sh_media_FIPR_S(const void *fvg, const void *fvh)
```

Notionally: `const float fvg[4]`, `const float fvh[4]`.

`fvg` and `fvh` point to 8-byte aligned data.

$$f = \begin{bmatrix} g_0 & g_1 & g_2 & g_3 \end{bmatrix} \begin{bmatrix} h_0 \\ h_1 \\ h_2 \\ h_3 \end{bmatrix}$$

```
__inline__ float  sh_media_FMAC_S(float fg, float fh, float fq)
__inline__ long long sh_media_FMOV_DQ(double dg)
__inline__ float sh_media_FMOV_LS(int mm)
__inline__ double sh_media_FMOV_QD(long long mm)
__inline__ int sh_media_FMOV_SL(float fg)
__inline__ void   sh_media_FPUTSCR(float fg)
__inline__ float sh_media_FSINA_S(float fg)
__inline__ double sh_media_FSQRT_D(double dg)
__inline__ float  sh_media_FSQRT_S(float fg)
__inline__ float sh_media_FSRRA_S(float fg)
__inline__ void   sh_media_FTRV_S(const void *mtrxg, const void *fvh, void *fvf)
```

Notionally: `const float mtrxg[4][4]`, `const float fvh[4]`, `float fvf[4]`.

`mtrxg`, `fvh`, `fvf` point to 8-byte aligned data.

$$\begin{bmatrix} h_0 & h_1 & h_2 & h_3 \end{bmatrix} \begin{bmatrix} g_{0,0} & g_{0,1} & g_{0,2} & g_{0,3} \\ g_{1,0} & g_{1,1} & g_{1,2} & g_{1,3} \\ g_{2,0} & g_{2,1} & g_{2,2} & g_{2,3} \\ g_{3,0} & g_{3,1} & g_{3,2} & g_{3,3} \end{bmatrix} = \begin{bmatrix} f_0 & f_1 & f_2 & f_3 \end{bmatrix}$$

3.5.3 Control and configuration instructions

```
__inline__ unsigned long long sh_media_GETCON(unsigned int k)
__inline__ void      sh_media_PUTCON(unsigned long long mm, unsigned int k)
__inline__ unsigned long long sh_media_GETCFG(unsigned long long mm, int s)
s is the displacement operand, expressed in bytes.
__inline__ void     sh_media_PUTCFG(unsigned long long mm, int s, unsigned long
long mw)
s is the displacement operand, expressed in bytes.
__inline__ void      sh_media_SLEEP(void)
```

3.5.4 Misaligned access support instructions

```
__inline__ unsigned long long sh_media_LDHI_L(void *p, int s)
s is the displacement operand, expressed in bytes.
__inline__ unsigned long long sh_media_LDHI_Q(void *p, int s)
s is the displacement operand, expressed in bytes.
__inline__ unsigned long long sh_media_LDLO_L(void *p, int s)
s is the displacement operand, expressed in bytes.
__inline__ unsigned long long sh_media_LDLO_Q(void *p, int s)
s is the displacement operand, expressed in bytes.
__inline__ void      sh_media_STHI_L(void *p, int s, unsigned int mw)
s is the displacement operand, expressed in bytes.
__inline__ void      sh_media_STHI_Q(void *p, int s, unsigned long long mw)
s is the displacement operand, expressed in bytes.
__inline__ void      sh_media_STLO_L(void *p, int s, unsigned int mw)
s is the displacement operand, expressed in bytes.
__inline__ void      sh_media_STLO_Q(void *p, int s, unsigned long long mw)
s is the displacement operand, expressed in bytes.
```

3.5.5 Miscellaneous instructions

```
__inline__ unsigned char sh_media_NSB(long long mm)
__inline__ unsigned long long sh_media_BYTEREV(unsigned long long mm)
__inline__ unsigned long long sh_media_CMVEQ(unsigned long long mm, unsigned
long long mn, unsigned long long mw)
__inline__ unsigned long long sh_media_CMVNE(unsigned long long mm, unsigned
long long mn, unsigned long long mw)
__inline__ unsigned long long sh_media_ADDZ_L(unsigned int mm, unsigned int mn)
__inline__ void sh_media_NOP(void)
```

3.5.6 Synchronization instructions

```
__inline__ unsigned long long sh_media_SWAP_Q(void *mm, long long mn, unsigned
long long mw)
__inline__ void      sh_media_SYNCI(void)
__inline__ void      sh_media_SYNCO(void)
```

3.5.7 Cache instructions

```
__inline__ void sh_media_ALLOCO(void *mm, int s)
s is the displacement operand, expressed in bytes.
__inline__ void sh_media_ICBI(void *mm, int s)
s is the displacement operand, expressed in bytes.
__inline__ void sh_media_OCBI(void *mm, int s)
s is the displacement operand, expressed in bytes.
__inline__ void sh_media_OCBP(void *mm, int s)
s is the displacement operand, expressed in bytes.
__inline__ void sh_media_OCBWB(void *mm, int s)
s is the displacement operand, expressed in bytes.
```

```
__inline__ void sh_media_PREFI(void *mm, int s)
s is the displacement operand, expressed in bytes.
__inline__ void sh_media_PREFO(void *mm, int s)
s is the displacement operand, expressed in bytes.
This intrinsic should be implemented using
LD.B Rm,s,R63
```

3.5.8 Event handling instructions

```
__inline__ void sh_media_BRK(void)
__inline__ void sh_media_TRAPA(unsigned long long mm)
```

3.5.9 Unaligned load and store functions

```
__inline__ short sh_media_unaligned_LD_W(void *p)
__inline__ unsigned short sh_media_unaligned_LD_UW(void *p)
__inline__ int sh_media_unaligned_LD_L(void *p)
__inline__ long long sh_media_unaligned_LD_Q(void *p)
__inline__ void sh_media_unaligned_ST_W(void *p, unsigned int k)
__inline__ void sh_media_unaligned_ST_L(void *p, unsigned int k)
__inline__ void sh_media_unaligned_ST_Q(void *p, unsigned long long k)
```

3.5.10 Floating-point Functions

```
__inline__ void sh_media_FTRVADD_S(const void *mtrxg, const void *fvh, const
void *fvi, void *fvf)
```

Notionally: `const float mtrxg[4][4]`, `const float fvh[4]`, `const float fvi[4]`, `float fvf[4]`.

mtrxg, **fvh**, **fvi**, **fvf** point to 8-byte aligned data.

$$\begin{bmatrix} h_0 & h_1 & h_2 & h_3 \end{bmatrix} \begin{bmatrix} g_{0,0} & g_{0,1} & g_{0,2} & g_{0,3} \\ g_{1,0} & g_{1,1} & g_{1,2} & g_{1,3} \\ g_{2,0} & g_{2,1} & g_{2,2} & g_{2,3} \\ g_{3,0} & g_{3,1} & g_{3,2} & g_{3,3} \end{bmatrix} + \begin{bmatrix} i_0 & i_1 & i_2 & i_3 \end{bmatrix} = \begin{bmatrix} f_0 & f_1 & f_2 & f_3 \end{bmatrix}$$

```
__inline__ void sh_media_FTRVSUB_S(const void *mtrxg, const void *fvh, const
void *fvi, void *fvf)
```

Notionally:`const float mtrxg[4][4]`, `const float fvh[4]`, `const float fvi[4]`, `float fvf[4]`.

mtrxg, **fvh**, **fvi**, **fvf** point to 8-byte aligned data.

$$\begin{bmatrix} h_0 & h_1 & h_2 & h_3 \end{bmatrix} \begin{bmatrix} g_{0,0} & g_{0,1} & g_{0,2} & g_{0,3} \\ g_{1,0} & g_{1,1} & g_{1,2} & g_{1,3} \\ g_{2,0} & g_{2,1} & g_{2,2} & g_{2,3} \\ g_{3,0} & g_{3,1} & g_{3,2} & g_{3,3} \end{bmatrix} - \begin{bmatrix} i_0 & i_1 & i_2 & i_3 \end{bmatrix} = \begin{bmatrix} f_0 & f_1 & f_2 & f_3 \end{bmatrix}$$

_inline__ void sh_media_FVADD_S(const void *fvg, const void *fvh, void *fvf)

Notionally: **const float fvg[4]**, **const float fvh[4]**, **float fvf[4]**.

fvg, **fvh**, and **fvf** point to 8-byte aligned data.

$$\begin{bmatrix} g_0 & g_1 & g_2 & g_3 \end{bmatrix} + \begin{bmatrix} h_0 & h_1 & h_2 & h_3 \end{bmatrix} = \begin{bmatrix} f_0 & f_1 & f_2 & f_3 \end{bmatrix}$$

__inline__ void sh_media_FVSUB_S(const void *fvg, const void *fvh, void *fvf)

Notionally: **const float fvg[4]**, **const float fvh[4]**, **float fvf[4]**.

fvg, **fvh**, and **fvf** point to 8-byte aligned data.

$$\begin{bmatrix} g_0 & g_1 & g_2 & g_3 \end{bmatrix} - \begin{bmatrix} h_0 & h_1 & h_2 & h_3 \end{bmatrix} = \begin{bmatrix} f_0 & f_1 & f_2 & f_3 \end{bmatrix}$$

__inline__ void sh_media_FMTRXMUL_S(const void *mtrxg, const void *mtrxh, void *mtrxf)

Notionally: **const float mtrxg[4][4]**, **const float mtrxh[4][4]**, **float mtrxf[4][4]**.

mtrxg, **mtrxh**, **mtrxf** point to 8-byte aligned data.

$$\begin{bmatrix} g_{0,0} & g_{0,1} & g_{0,2} & g_{0,3} \\ g_{1,0} & g_{1,1} & g_{1,2} & g_{1,3} \\ g_{2,0} & g_{2,1} & g_{2,2} & g_{2,3} \\ g_{3,0} & g_{3,1} & g_{3,2} & g_{3,3} \end{bmatrix} \begin{bmatrix} h_{0,0} & h_{0,1} & h_{0,2} & h_{0,3} \\ h_{1,0} & h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,0} & h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,0} & h_{3,1} & h_{3,2} & h_{3,3} \end{bmatrix} = \begin{bmatrix} f_{0,0} & f_{0,1} & f_{0,2} & f_{0,3} \\ f_{1,0} & f_{1,1} & f_{1,2} & f_{1,3} \\ f_{2,0} & f_{2,1} & f_{2,2} & f_{2,3} \\ f_{3,0} & f_{3,1} & f_{3,2} & f_{3,3} \end{bmatrix}$$

__inline__ void sh_media_FMTRXMULADD_S(const void *mtrxg, const void *mtrxh, const void *mtrxi, void *mtrxf)

Notionally: **const float mtrxg[4][4]**, **const float mtrxh[4][4]**, **const float mtrxi[4][4]**, **float mtrxf[4][4]**.

mtrxg, **mtrxh**, **mtrxi**, **mtrxf** point to 8-byte aligned data.

$$\begin{bmatrix} g_{0,0} & g_{0,1} & g_{0,2} & g_{0,3} \\ g_{1,0} & g_{1,1} & g_{1,2} & g_{1,3} \\ g_{2,0} & g_{2,1} & g_{2,2} & g_{2,3} \\ g_{3,0} & g_{3,1} & g_{3,2} & g_{3,3} \end{bmatrix} \begin{bmatrix} h_{0,0} & h_{0,1} & h_{0,2} & h_{0,3} \\ h_{1,0} & h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,0} & h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,0} & h_{3,1} & h_{3,2} & h_{3,3} \end{bmatrix} + \begin{bmatrix} i_{0,0} & i_{0,1} & i_{0,2} & i_{0,3} \\ i_{1,0} & i_{1,1} & i_{1,2} & i_{1,3} \\ i_{2,0} & i_{2,1} & i_{2,2} & i_{2,3} \\ i_{3,0} & i_{3,1} & i_{3,2} & i_{3,3} \end{bmatrix} = \begin{bmatrix} f_{0,0} & f_{0,1} & f_{0,2} & f_{0,3} \\ f_{1,0} & f_{1,1} & f_{1,2} & f_{1,3} \\ f_{2,0} & f_{2,1} & f_{2,2} & f_{2,3} \\ f_{3,0} & f_{3,1} & f_{3,2} & f_{3,3} \end{bmatrix}$$

`__inline__ void sh_media_FMTRXMULSUB_S(const void *mtrxg, const void *mtrxh, const void *mtrxi, void *mtrxf)`

Notionally: **`const float mtrxg[4][4]`**, **`const float mtrxh[4][4]`**, **`const float mtrxi[4][4]`**, **`float mtrxf[4][4]`**.

`mtrxg`, **`mtrxh`**, **`mtrxi`**, **`mtrxf`** point to 8-byte aligned data.

$$\begin{bmatrix} g_{0,0} & g_{0,1} & g_{0,2} & g_{0,3} \\ g_{1,0} & g_{1,1} & g_{1,2} & g_{1,3} \\ g_{2,0} & g_{2,1} & g_{2,2} & g_{2,3} \\ g_{3,0} & g_{3,1} & g_{3,2} & g_{3,3} \end{bmatrix} \begin{bmatrix} h_{0,0} & h_{0,1} & h_{0,2} & h_{0,3} \\ h_{1,0} & h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,0} & h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,0} & h_{3,1} & h_{3,2} & h_{3,3} \end{bmatrix} - \begin{bmatrix} i_{0,0} & i_{0,1} & i_{0,2} & i_{0,3} \\ i_{1,0} & i_{1,1} & i_{1,2} & i_{1,3} \\ i_{2,0} & i_{2,1} & i_{2,2} & i_{2,3} \\ i_{3,0} & i_{3,1} & i_{3,2} & i_{3,3} \end{bmatrix} = \begin{bmatrix} f_{0,0} & f_{0,1} & f_{0,2} & f_{0,3} \\ f_{1,0} & f_{1,1} & f_{1,2} & f_{1,3} \\ f_{2,0} & f_{2,1} & f_{2,2} & f_{2,3} \\ f_{3,0} & f_{3,1} & f_{3,2} & f_{3,3} \end{bmatrix}$$

`__inline__ void sh_media_FVCOPY_S(const void *fvg, void *fvf)`

Notionally: **`const float fvg[4]`**, **`float fvf[4]`**.

`fvg`, **`fvf`** point to 8-byte aligned data.

$$\begin{bmatrix} g_0 & g_1 & g_2 & g_3 \end{bmatrix} \rightarrow \begin{bmatrix} f_0 & f_1 & f_2 & f_3 \end{bmatrix}$$

`__inline__ void sh_media_FMTRXCOPY_S(const void *mtrxg, void *mtrxf)`

Notionally: **`const float mtrxg[4][4]`**, **`float mtrxf[4][4]`**.

`mtrxg`, **`mtrxf`** point to 8-byte aligned data.

$$\begin{bmatrix} g_{0,0} & g_{0,1} & g_{0,2} & g_{0,3} \\ g_{1,0} & g_{1,1} & g_{1,2} & g_{1,3} \\ g_{2,0} & g_{2,1} & g_{2,2} & g_{2,3} \\ g_{3,0} & g_{3,1} & g_{3,2} & g_{3,3} \end{bmatrix} \rightarrow \begin{bmatrix} f_{0,0} & f_{0,1} & f_{0,2} & f_{0,3} \\ f_{1,0} & f_{1,1} & f_{1,2} & f_{1,3} \\ f_{2,0} & f_{2,1} & f_{2,2} & f_{2,3} \\ f_{3,0} & f_{3,1} & f_{3,2} & f_{3,3} \end{bmatrix}$$

`__inline__ void sh_media_FMTRXADD_S(const void *mtrxg, const void *mtrxh, void *mtrxf)`

Notionally: **`const float mtrxg[4][4]`**, **`const float mtrxh[4][4]`**, **`float mtrxf[4][4]`**.

`mtrxg`, **`mtrxh`**, **`mtrxf`** point to 8-byte aligned data.

$$\begin{bmatrix} g_{0,0} & g_{0,1} & g_{0,2} & g_{0,3} \\ g_{1,0} & g_{1,1} & g_{1,2} & g_{1,3} \\ g_{2,0} & g_{2,1} & g_{2,2} & g_{2,3} \\ g_{3,0} & g_{3,1} & g_{3,2} & g_{3,3} \end{bmatrix} + \begin{bmatrix} h_{0,0} & h_{0,1} & h_{0,2} & h_{0,3} \\ h_{1,0} & h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,0} & h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,0} & h_{3,1} & h_{3,2} & h_{3,3} \end{bmatrix} = \begin{bmatrix} f_{0,0} & f_{0,1} & f_{0,2} & f_{0,3} \\ f_{1,0} & f_{1,1} & f_{1,2} & f_{1,3} \\ f_{2,0} & f_{2,1} & f_{2,2} & f_{2,3} \\ f_{3,0} & f_{3,1} & f_{3,2} & f_{3,3} \end{bmatrix}$$

```
__inline__ void sh_media_FMTRXSUB_S(const void *mtrxg, const void *mtrxh, void
*mtrxf)
```

Notionally: **`const float mtrxg[4][4]`**, **`const float mtrxh[4][4]`**, **`float mtrxf[4][4]`**.

`mtrxg`, **`mtrxh`**, **`mtrxf`** point to 8-byte aligned data.

$$\begin{bmatrix} g_{0,0} & g_{0,1} & g_{0,2} & g_{0,3} \\ g_{1,0} & g_{1,1} & g_{1,2} & g_{1,3} \\ g_{2,0} & g_{2,1} & g_{2,2} & g_{2,3} \\ g_{3,0} & g_{3,1} & g_{3,2} & g_{3,3} \end{bmatrix} - \begin{bmatrix} h_{0,0} & h_{0,1} & h_{0,2} & h_{0,3} \\ h_{1,0} & h_{1,1} & h_{1,2} & h_{1,3} \\ h_{2,0} & h_{2,1} & h_{2,2} & h_{2,3} \\ h_{3,0} & h_{3,1} & h_{3,2} & h_{3,3} \end{bmatrix} = \begin{bmatrix} f_{0,0} & f_{0,1} & f_{0,2} & f_{0,3} \\ f_{1,0} & f_{1,1} & f_{1,2} & f_{1,3} \\ f_{2,0} & f_{2,1} & f_{2,2} & f_{2,3} \\ f_{3,0} & f_{3,1} & f_{3,2} & f_{3,3} \end{bmatrix}$$

3.6 Compiler support routines

To avoid name clashes, compiler support routines should be prefixed by:

___<*Compiler's unique id*>_

where

1 The first underscore is the underscore prepended to all C identifiers

2 The next two underscores shows that this identifier is reserved for the compiler (C language convention)

3 <*Compiler's unique ID*> specifies the compiler developer, or names derived from it. The following unique ids have been allocated:

`H`	Hitachi
`ST`	STMicroelectronics

For example, Hitachi's divide routine is called **`___H_div`**, and ST's divide routine is called **`___ST_div`**.

A Appendix

The appendix contains examples of various aspects of the ABI. It does not form part of the ABI itself.

Note: Certain aspects of the assembly syntax used in this section may differ depending upon the assembler used.

A.1 Passing 64-bit parameters

The SHcompact instruction set does not have visibility of the upper 32 bits of the general purpose registers, yet this ABI requires that some function arguments and results are passed in these upper bits.

This section discusses a scheme for handling such arguments and results in SHcompact.

The scheme described in this section is an example intended to guide implementors of this ABI, it is not mandated by the ABI.

A.1.1 General principles

When an SHcompact program needs to read or write the upper 32 bits of an integer register, it must switch to SHmedia in order to do so. The switch may either be performed in line, or by a call to a SHmedia service routine. Inline code is potentially faster, as it can be tuned to the particular requirements of the program, but it is considerably more bulky: it requires on average 9 bytes of code (four instructions plus a longword alignment) to switch from SHcompact to SHmedia, and 8 bytes of code (two instructions) to switch back from SHmedia to SHcompact.

In a mixed mode program, the main benefit of SHcompact is compact code size, so the scheme outlined here makes exclusive use of service routines to handle 64-bit parameters and results.

There are three areas to consider:

1 Receiving 64-bit parameters
2 Returning 64-bit results
3 Passing 64-bit arguments

Each of these is considered in turn.

A.1.2 Receiving 64-bit parameters

Long long and aggregate parameters occupy the upper as well as the lower, 32 bits of an integer parameter register. The simplest thing to do here is to store the register on the stack and then access the values from memory. This is necessary for variadic functions anyway, so it is a useful first base for handling wide parameters. Consider

```
void fn(int c, ...)
{
   ...
}
```

The following is a possible entry sequence for this function:

```
_fn:
   ADD    #-68,R15
   STS.L  PR,@-R15
   MOV.L  @(.L11,PC),R0
   JSR    @R0
   NOP
...
.L11:
   .long  ___push_int_args
```

where `__push_int_args` is a service routine written in SHmedia code:

```
___push_int_args:
   PTABS  R18, T0
   ST.Q   R15,  8, R2
   ST.Q   R15, 16, R3
   ST.Q   R15, 24, R4
   ST.Q   R15, 32, R5
   ST.Q   R15, 40, R6
   ST.Q   R15, 48, R7
   ST.Q   R15, 56, R8
   ST.Q   R15, 64, R9
   BLINK  T0, R63
```

There are a couple of points to note here:

- `PR` must be saved before calling the service routine, but it is saved at a lower address than the integer parameter registers, so that the saved integer parameter registers are contiguous with the stack parameters (this enables easier implementation of the **`va_arg`**`()` macro). `PR` only requires 4 stack bytes, but the service routine must store the argument registers at offsets a multiple of 8 bytes from `R15`, so `R15` has to be pulled down an extra 4 bytes (68 rather than 64). In this example, these 4 bytes are wasted, though they could be used to hold local variables.
- The service routine has saved all the integer parameter registers, even though it is not necessary to save the first register, because the first register does not contain a variadic parameter. It is possible to write variants of **`__push_int_args`** that only push the required number of parameter registers, for example, **`__push_7_int_args`** could push R3 through R9 only. The compiler knows how many registers need to be saved, and can call the appropriate service routine.
- In this and following examples, an SHcompact function call is represented as a MOV.L; JSR sequence. However, if the target function is close enough, this can be replaced by a BSR, and the literal pool entry containing the target function address can be removed. The BSR range is 4096 bytes.

The above service routine is also adequate for handling entry to functions that have 64-bit parameters, though it is inefficient when there are only a small number of 64-bit parameters. Consider

```
void fn(int a, int b, long long c, int d)
{
        ...
}
```

On entry, a is in R2, b in R3, c in R4, and d in R5. Only c is a 64-bit parameter. The upper 32 bits of c could be extracted by the code:

```
_fn:
   ... entry preamble
   MOV.L  @(.L11,PC),R0
   JSR    @R0
   NOP
   ... now c is in R4 (l.s. part) and R1 (m.s. part)
   ...
.L11:
   .long  ___unpack_reg_R4
```

where **__unpack_reg_R4** is a service routine written in SHmedia. It extracts the upper 32 bits of R4 to the lower 32 bits of R1 (a scratch register), and sign extends the lower 32 bits of R4.

```
___unpack_reg_R4:
   PTABS  R18,T0
   SHARI  R4,32,R1
   ADDI.L R4,0,R4
   BLINK  T0, R63
```

If there are a large number of 64-bit parameters, then it may be more efficient to store them to memory. Extracting them all to register pairs will greatly increase the register pressure and force stores anyway.

A single 64-bit parameter can be stored to memory as follows:

```
_fn:
   ... entry preamble
   MOV.L  @(.L11,PC),R0
   MOV  R15,R1
   ADD  #c_offset,R1
   JSR    @R0
   NOP
   ... now c is in the stack frame at R15+c_offset
   ...
.L11:
   .long  ___store64_reg_R4
```

where **___store64_reg_R4** is a service routine written in SHmedia. It stores the 64-bit value in R4 to memory at the address in R1.

```
___store64_reg_R4:
   PTABS  R18,T0
   ST.Q   R1,0,R4
   BLINK  T0,R63
```

It would be slightly more efficient to tailor a version of **___store64_reg_R4** to store the 64-bit register at an offset from the stack pointer. This saves one instruction at the call site, for example:

```
_fn:
   ... entry preamble
   MOV.L  @(.L11,PC),R0
   MOV  #c_offset,R1
   JSR    @R0
   NOP
   ... now c is in the stack frame at R15+c_offset
   ...
.L11:
   .long  ___store64_R15_reg_R4
   ...

___store64_R15_reg_R4:
   PTABS  R18,T0
   STX.Q  R15,R1,R4
   BLINK  T0,R63
```

A.1.3 Returning 64-bit results

Long long and aggregate results 8 bytes or less in length are returned in a 64-bit register. This can be achieved in SHcompact by loading the lower 32 bits of the result into the result register, the upper 32 bits of the result into a scratch register (for example, R1), and tail-calling a service routine that packs the upper 32 bits of the result into the result register.

Consider

```
struct s { int a; int b; };

struct s fn(void)
{
   struct s res;
   ... body of fn
   res.a = 1;
   res.b = 2;
   return res;
}
```

This can be coded as:

```
_fn:
   ... body of fn
   MOV.L   @(.L13,PC),R0
   MOV     #1,R2 ; lower 32 bits of result
   JMP     @R0   ; tail-call result packing function
   MOV     #2,R1 ; upper 32 bits of result
.L13:
   .long ___pack_reg_R2
```

where **__pack_reg_r2** is a service routine written in SHmedia that copies the bottom 32 bits of R1 (a scratch register) into the top 32 bits of R2:

```
___pack_reg_R2:
   PTABS R18, T0
   MSHFLO.L    R2, R1, R2
   BLINK       T0, R63
```

If the returned value is in memory at the time it needs to be returned, then it can be more efficient to tail-call a service routine to load it from memory instead, for example, for

```
struct s { int a; int b; } static_struct;

struct s fn(void)
{
   ... body of fn
   return static_struct;
}
```

can be coded as:

```
_fn:
   ... body of fn
   MOV.L   @(.L12,PC),R2
   MOV.L   @(.L13,PC),R0
   JMP     @R0
   NOP
L12:
   .long _static_struct
L13:
   .long ___load64_reg_R2
   ...
___load64_reg_R4:
   PTABS  R18,T0
   LD.Q   R2,0,R2
   BLINK  T0,R63
```

A.1.4 Passing 64-bit arguments

Long long arguments are passed in a 64-bit register, and aggregate parameters are passed in one or more 64-bit registers. They can be loaded by putting the lower 32 bits directly into the argument register, and the upper 32 bits into a scratch register, then calling a SHmedia service routine to copy the upper 32 bits into the argument register.

Consider

```
struct s { int a; int b; };

void fn(struct s);
...
void fn2(void)
{
   ...
   struct s arg;
   arg.a = 1;
   arg.b = 2;
   fn(arg);
   ...
}
```

This can be coded as:

```
_fn2:
...
   MOV.L   @(.L14,PC),R0
   MOV     #1,R2
   JSR     @R0
   MOV     #2,R1
   MOV.L   @(.L15,PC),R0
   JSR     @R0
   NOP
   ...
.L14:
   .long   ___pack_reg_R2
.L15:
   .long   _fn
```

where **___pack_reg_r2** is the same service routine as described earlier. Again, it may be more efficient in some circumstances to call **___load64_reg_r2** to load a 64-bit value from memory, rather than calling **___pack_reg_r2**.

Care should be taken to pack arguments into argument registers only after all function calls contained in the arguments and function designator have been performed (except for calls to service routines), as it is not possible in SHcompact to save 64-bit registers around function calls, except by storing them to memory, which will involve a switch to SHmedia both to store and to reload. Although the SHcompact register-register MOV instruction copies 64 bits from source to destination, there is no register that can be used to hold a 64-bit value across a function call (the upper 32 bits of all registers visible in SHcompact are not preserved across function calls).

Note: This also means that some optimizations are not possible.

Consider

```
void fn(long long);
...
long long ll;
fn(ll);
fn(ll);
```

Here it is not possible to perform common subexpression elimination on the packed value of `ll` and keep it in a register, as the register value is not preserved across the first call to **fn**.

A.2 Parameter passing - further examples

Example: Passing floating-point values in both floating-point and integer registers.

If a floating-point argument occurs within the first 8 argument elements, then it is passed both in a floating-point register and in an integer register. However, if the prototype is in scope and the floating-point argument does not correspond to a variadic parameter, then it need not be passed in an integer register, though an integer register is still allocated, but left unused.

Consider a call to:

```
void fn(int i1, double d1, int i2);
```

Table 8 illustrates how the arguments are passed.

Argument	Machine resource	
	Prototype in scope	No prototype in scope
i1	R2	R2
d1	DR0[a]	DR0 and R3
i2	R4	R4

Table 8:

a. A ‘hole’ is left in R3, which is unused

If the prototype is in scope and the floating-point argument does correspond to a variadic parameter, then it need not be passed in a floating-point register, and no floating-point register need be allocated.

Consider the call:

```
int i1, i2;
double d1;
fn(i1, d1, i2);
```

where fn has the prototype:

```
void fn(int i, ...);
```

Table 9 illustrates how the arguments are passed.

Argument	Machine resource	
	Prototype in scope	No prototype in scope
i1	R2	R2
d1	R3	DR0 and R3
i2	R4	R4

Table 9:

Example: Leaving ‘holes’ on the stack.

If a floating-point argument occurs after the first 8 argument elements are filled, then it is passed both in a floating-point register and on the stack. However, if the prototype is in scope and the floating-point argument does not correspond to a variadic parameter, then it need not be passed on the stack, though 8 bytes of stack are still allocated, but left unused.

Consider a call to

```
void fn(int i1, int i2, int i3, int i4, int i5, int i6,
        int i7, int i8, double d1, int i9);
```

Table 10 illustrates how the arguments are passed.

Argument	Machine resource	
	Prototype in scope	No prototype in scope
i1	R2	R2
i2	R3	R3
i3	R4	R4
i4	R5	R5
i5	R6	R6
i6	R7	R7
i7	R8	R8
i8	R9	R9
d1	DR0	[SP+0,SP+7] and DR0
i9	[SP+8,SP+15][a]	[SP+8,SP+15]

Table 10:

a. An 8 byte ‘hole’ is left on the stack, at (0,SP) through (7,SP).

Example: Passing floating-point values in integer registers when floating-point registers are exhausted.

If a floating-point argument occurs after all floating-point argument registers have been filled, but there are still integer argument registers available, then the argument is passed in the next available integer register.

Consider a call to:

```
void fn(double d1, double d2, double d3, double d4, double d5,
        double d6, double d7, double d8, double d9);
```

Table 11 illustrates how the arguments are passed.

Argument	Machine resource(s)	
	Prototype in scope	No prototype in scope
d1	DR0	DR0 and R2
d2	DR2	DR2 and R3
d3	DR4	DR4 and R4
d4	DR6	DR6 and R5
d5	DR8	DR8 and R6
d6	DR10	DR10 and R7
d7	R8	R8
d8	R9	R9

Table 11:

Argument	Machine resource(s)	
	Prototype in scope	No prototype in scope
d9	[SP+0,SP+7]	[SP+0,SP+7]

Table 11:

Example: Passing small aggregates

Aggregates of size less than 8 bytes are padded at the most significant end of the argument element.

Consider

```
struct s { short x, y;} num;
fn(num);
```

NUM is passed in R2 as illustrated in *Figure 5*:

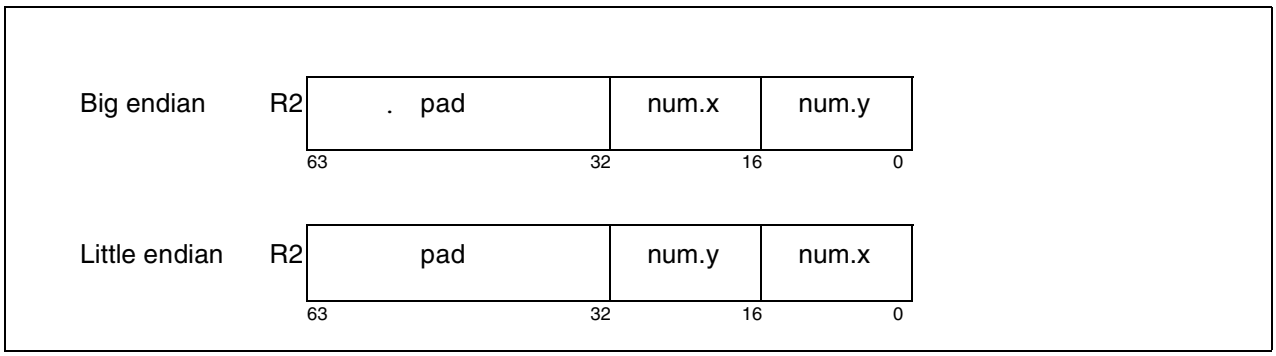

Figure 5:

Example: Passing large aggregates

Aggregates that are larger than 8 bytes are padded in the last argument element. If byte ordering is big endian, then the last argument element is padded at the least significant end; if byte ordering is little endian, then the last argument element is padded at the most significant end. Note that this means that for big endian byte ordering, aggregate padding occurs at different ends of an argument element for small (< 8 bytes) and large (> 8 byte) aggregates.

Consider

```
struct s{ int x, y, z; } coord;
fn(coord);
```

COORD is passed in R2 and R3, as illustrated in *Figure 6*.

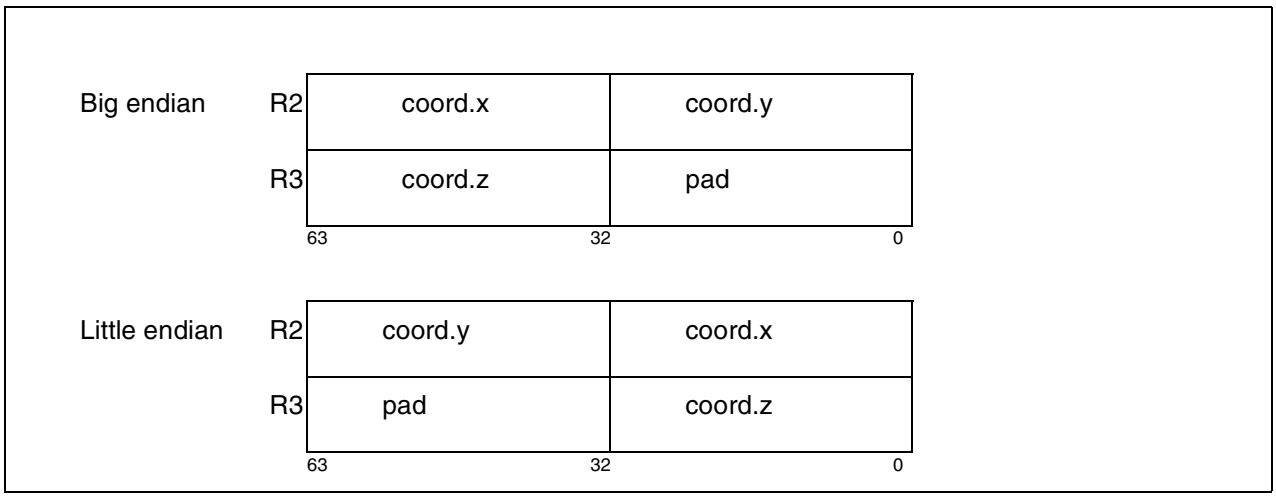

Figure 6:

Example: Argument straddling integer registers and stack

Aggregates are passed in integer registers when available, otherwise on the stack. If there are not enough integer registers available to hold the whole aggregate, then part of the aggregate will be passed in integer registers, and the remainder will be passed on the stack.

Consider

```
struct s { long long x, y, z; } coord64;
int i1, i2, i3, i4, i5, i6;
fn(i1,i2,i3,i4,i5,i6,coord64);
```

Table 12 illustrates how the arguments are passed.

Argument	Machine resource
i1	R2
i2	R3
i3	R4
i4	R5
i5	R6
i6	R7
coord.x	R8
coord.y	R9
coord.z	[SP+0,SP+7]

Table 12:

Example: Returning large aggregates

Aggregates that are larger than 8 bytes are not returned in registers. Instead the caller places in R2 the address of the returned value as an implicit extra argument. The callee writes the returned value to that address and ensures that the address is in R2 on return.

Consider

```
struct s { int x, y, z; } coord;
int val1, val2, val3;
coord = fn(val1, val2, val3);
```

Table 13 illustrates how the caller passes the arguments.

Argument	Machine resource
&coord	R2
val1	R3
val2	R4
val3	R5

Table 13:

Upon return, the callee will ensure that R2 contains &coord.

A.3 Implementation of STDARG.H

This is an example implementation of the STDARG.H macros, **va_start**, **va_arg**, and **va_end**.

Big-endian byte ordering

```
typedef char *va_list;
#define _VA_SIZE 8
#define _VA_DELTA (_VA_SIZE - 1)
#define _VA_MASK (~_VA_DELTA)
#define __va_pad(x)  (((x) + _VA_DELTA) & _VA_MASK)
#define __va_promote(type) (__va_pad((int)sizeof(type)))

#define va_start(ap,last) (ap =
(char*)__va_pad((int)((char*)&(last)+sizeof(last))))
#define va_arg(ap,type) (*(type*)((sizeof(type)<8)\
                      ?((ap+=__va_promote(type))-sizeof(type))\
                      :((ap+=__va_promote(type))-__va_promote(type))))
#define va_end(ap) (void)0
```

Little-endian byte ordering

```
typedef char *va_list;
#define va_end(ap) (void)0
#define _VA_SIZE 8
#define _VA_DELTA (_VA_SIZE - 1)
#define _VA_MASK (~_VA_DELTA)
#define __va_pad(x)  (((x) + _VA_DELTA) & _VA_MASK)
#define __va_promote(type) (__va_pad((int)sizeof(type)))

#define va_start(ap,last) (ap = ((char*)&(last) + __va_promote(last)))
#define va_arg(ap,type) (*(type*)((ap+=__va_promote(type))-__va_promote(type)))
#define va_end(ap) (void)0
```

A.4 Usage of R25

This section contains some examples of the permitted usage of register R25.

Consider the C code:

```
static_var = 1;
```

For this, the compiler can generate:

```
MOVI 1,R2
ST.L R26, _static_var-.data, R2
```

At link-time, a linker that performed code relaxation could determine whether the expression

```
_static_var - .data
```

fits in the immediate field of the ST.L instruction. If it does not,then at link time the ST.L instruction would be expanded to:

```
MOVI (_static_var - .data)>>16, R25
SHORI (_static_var - .data)&0xffff, R25
STX.L R26, R25, R2
```

Here the linker has used R25 in the expansion of the relocation of the ST.L instruction. This value of R25 is created just for this one expansion: it is not used again, so it is dead after the STX.L instruction.

As the expansion of a relocation can overwrite the value in R25, the assembler must assume that the value of R25 is changed by an instruction that is subject to relocation, and so it cannot place a value in R25 before a relocated instruction, and later use that value after the relocated instruction. So for example, the following is not permitted:

```
MOVI 23,R25
LD.L R26,_static_var-.data,R2   ; this instruction will be relocated
ADD  R2,R25,R2
```

because the relocation of the LD.L may overwrite the value in R25.

However, the assembler can still use R25 in places where the value does not need to be preserved across a relocation. For example, if the assembler allows a large immediate value in the ADDI instruction, then the code for

```
static_var += 100000;
```

could be

```
LD.L R26, _static_var-.data,R2   ; this instruction will be relocated
ADDI R2,100000,R2
ST.L R26, _static_var-.data,R2   ; this instruction will be relocated
```

Here it is permissible for the assembler to expand the ADDI to:

```
MOVI  (100000)>>16,R25
SHORI (100000)&0xffff, R25
ADD   R2,R25,R2
```

because the lifetime of this usage of R25 is from the MOVI to the ADD instruction only: it does not matter that the LD.L or the ST.L instructions overwrite R25.

So in the worst case the fully expanded sequence could be:

```
MOVI  (_static_var - .data)>>16, R25     ; Start of R25 life 1
SHORI (_static_var - .data)&0xffff, R25
LDX.L R26, R25, R2                       ; End of R25 life 1
MOVI  (100000)>>16,R25                   ; Start of R25 life 2
SHORI (100000)&0xffff,R25
ADD   R2,R25,R2                          ; End of R25 life 2
MOVI  (_static_var - .data)>>16, R25     ; Start of R25 life 3
SHORI (_static_var - .data)&0xffff, R25
STX.L R26, R25, R2                       ; End of R25 life 3
```

that is, there are three distinct lifetimes for R25. Life 1 and life 3 are created at link time, and life 2 is created at assembly time. It is safe for the assembler to create life 2, as it does not overlap life 1 or life 3.

SuperH, Inc.

Issued by the SuperH Documentation Group on behalf of SuperH, Inc.

 is a registered trademark of SuperH, Inc.

SuperH, Inc.
San Jose, U.S.A. - Bristol, United Kingdom - Tokyo, Japan

www.superh.com