Procedure Call Standard for the ARM® Architecture

Development systems Division

Compiler Tools Group

Document number: GENC-003534
Date of Issue: 30th October 2003
Author: Richard Earnshaw
Authorized by:

Abstract

This document describes the Procedure Call Standard use by the Application Binary Interface (ABI) for the ARM architecture.

Keywords

Procedure call function call, calling conventions, data layout

Licence

Proprietary notice

Contents

1 ABOUT THIS DOCUMENT 4

1.1 Change Control 4
1.1.1 Current Status and Anticipated Changes 4
1.1.2 Change History 4

1.2 References 4

1.3 Terms and Abbreviations 5

1.4 About the Licence to Use This Specification 6

1.5 Acknowledgements 6

2 SCOPE 7

3 INTRODUCTION 8

3.1 Design Goals 8

3.2 Conformance 8

4 DATA TYPES AND ALIGNMENT 9

4.1 Fundamental Data Types 9

4.2 Endianness and Byte Ordering 9

4.3 Composite Types 10
4.3.1 Aggregates 11
4.3.2 Unions 11
4.3.3 Arrays 11
4.3.4 Bit-fields 11
4.3.5 Homogeneous Aggregates 11

5 THE BASE PROCEDURE CALL STANDARD 12

5.1 Machine Registers 12
5.1.1 Co-processor Registers 13
5.1.1.1 VFP register usage conventions 13

5.2 Processes, Memory and the Stack 14
5.2.1 The Stack 14
5.2.1.1 Universal stack constraints 15
5.2.1.2 Stack constraints at a public interface 15

5.3 Subroutine Calls 15
5.3.1.1 Use of IP by the linker 16

5.4 Result Return 16

5.5 Parameter Passing 16

5.6 Interworking 18

6 THE STANDARD VARIANTS 20

6.1 VFP Register Arguments 20
6.1.1 Procedure Calling 20
6.1.1.1 Result return 20
6.1.1.2 Parameter passing 20

6.2 Read-Write Position Independence (RWPI) 21

6.3 Variant Compatibility 21
6.3.1 VFP and Base Standard Compatibility 21
6.3.2 RWPI and Base Standard Compatibility 21
6.3.3 VFP and RWPI Standard Compatibility 21

7 ARM C AND C++ LANGUAGE MAPPINGS 22

7.1 Data Types 22
7.1.1 Arithmetic Types 22
7.1.2 Pointer Types 23
7.1.3 Enumerated Types 23
7.1.4 Additional Types 24
7.1.5 Volatile Data Types 24
7.1.6 Structure, Union and Class Layout 24
7.1.7 Bit-fields 25
7.1.7.1 Bit-fields no larger than their container 25
7.1.7.2 Bit-field extraction expressions 26
7.1.7.3 Over-sized bit-fields 26
7.1.7.4 Combining bit-field and non-bit-field members 27
7.1.7.5 Volatile bit-fields—preserving number and width of container accesses 27

7.2 Argument Passing Conventions 27

1 ABOUT THIS DOCUMENT

1.1 Change Control

1.1.1 Current Status and Anticipated Changes

This document has been released publicly. Anticipated changes to this document include:

- Typographical corrections.
- Clarifications.
- Compatible extensions.

1.1.2 Change History

Issue	Date	By	Change
1.0	30th October 2003	Lee Smith	First public release.

1.2 References

This document refers to, and is referred to by, the following documents.

Ref	URL or other reference	Title
AAPCS		Procedure Call Standard for the ARM Architecture (*This document*)
AAELF		ELF for the ARM Architecture
BSABI		ABI for the ARM Architecture (Base Standard)
CPPABI		C++ ABI for the ARM Architecture
ARM ARM	ARM DDI 0100E ISBN 0 201 737191	The ARM Architecture Reference Manual, 2nd edition, edited by David Seal, published by Addison-Wesley.

1.3 Terms and Abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
ABI	Application Binary Interface: 1. The specifications to which an executable must conform in order to execute in a specific execution environment. For example, the *Linux ABI for the ARM Architecture*. 2. A particular aspect of the specifications to which independently produced relocatable files must conform in order to be statically linkable and executable. For example, the *C++ ABI for the ARM Architecture*, the *Run-time ABI for the ARM Architecture*, the *C Library ABI for the ARM Architecture.*
ARM-based	… based on the ARM architecture …
EABI	An ABI suited to the needs of embedded (sometimes called *free standing*) applications.

PCS	Procedure Call Standard.
AAPCS	Procedure Call Standard for the ARM Architecture (this standard).
APCS	ARM Procedure Call Standard (obsolete).
TPCS	Thumb Procedure Call Standard (obsolete).
ATPCS	ARM-Thumb Procedure Call Standard (precursor to this standard).
PIC, PID	Position-independent code, position-independent data.

Routine, subroutine	A fragment of program to which control can be transferred that, on completing its task, returns control to its caller at an instruction following the call. *Routine* is used for clarity where there are nested calls: a routine is the *caller* and a subroutine is the *callee*.
Procedure	A routine that returns no result value.
Function	A routine that returns a result value.
Activation stack, call-frame stack	The stack of routine activation records (call frames).
Activation record, call frame	The memory used by a routine for saving registers and holding local variables (usually allocated on a stack, once per activation of the routine).
Argument, Parameter	The terms *argument* and *parameter* are used interchangeably. They may denote a formal parameter of a routine given the value of the actual parameter when the routine is called, or an actual parameter, according to context.
Externally visible [interface]	[An interface] between separately compiled or separately assembled routines.
Variadic routine	A routine is variadic if the number of arguments it takes, and their type, is determined by the caller instead of the callee.

Term	Meaning
Global register	A register whose value is neither saved nor destroyed by a subroutine. The value may be updated, but only in a manner defined by the execution environment.
Program state	The state of the program's memory, including values in machine registers.
Scratch register, temporary register	A register used to hold an intermediate value during a calculation (usually, such values are not named in the program source and have a limited lifetime).
Variable register, v-register	A register used to hold the value of a variable, usually one local to a routine, and often named in the source code.

More specific terminology is defined when it is first used.

1.4 About the Licence to Use This Specification

Use of these *ABI for the ARM Architecture* specifications published by ARM is governed by the simple licence agreement shown on the cover page of this document, and on the cover page of each major component document. Without formalities or payment, you are licensed to use any IP rights ARM might hold in these ABI specifications for the purpose of producing products that comply with these ABI specifications.

Because these specifications may be updated by ARM without notice, we prefer that these specifications should not be copied, but that third parties should refer directly to them, in the same way that we refer directly to the specifications underpinning this ABI, such as the specifications of ELF, DWARF, and the generic C++ ABI.

1.5 Acknowledgements

This specification could not have been developed without contributions from, and the active support of, the following organizations. In alphabetical order: ARM, Intel, Metrowerks, Montavista, Nexus Electronics, PalmSource, Symbian, and Wind River.

2 SCOPE

The AAPCS defines how subroutines can be separately written, separately compiled, and separately assembled to work together. It describes a contract between a calling routine and a called routine that defines:

- Obligations on the caller to create a program state in which the called routine may start to execute.
- Obligations on the called routine to preserve the program state of the caller across the call.
- The rights of the called routine to alter the program state of its caller.

This standard specifies the base for a family of *Procedure Call Standard (PCS)* variants generated by choices that reflect alternative priorities among:

- Code size.
- Performance.
- Functionality (for example, ease of debugging, run-time checking, support for shared libraries).

Some aspects of each variant – for example the allowable use of R9 – are determined by the execution environment. Thus:

- It is possible for code complying strictly with the base standard to be PCS compatible with each of the variants.
- It is unusual for code complying with a variant to be compatible with code complying with any other variant.
- Code complying with a variant, or with the base standard, is not guaranteed to be compatible with an execution environment that requires those standards. An execution environment may make further demands beyond the scope of the procedure call standard.

This standard is presented in four sections that, after an introduction, specify:

- The layout of data.
- Layout of the stack and calling between functions with public interfaces.
- Variations available for processor extensions, or when the execution environment restricts the addressing model.
- The C and C++ language bindings for plain data types.

This specification *does not* standardize the representation of publicly visible C++-language entities that are not also C language entities (these are described in CPPABI) and it places no requirements on the representation of language entities that are not visible across public interfaces.

3 INTRODUCTION

The AAPCS embodies the fifth major revision of the APCS and third major revision of the TPCS. It forms part of the complete ABI specification for the ARM Architecture.

3.1 Design Goals

The goals of the AAPCS are to:

- Support Thumb-state and ARM-state equally.
- Support inter-working between Thumb-state and ARM-state.
- Support efficient execution on high-performance implementations of the ARM Architecture.
- Clearly distinguish between mandatory requirements and implementation discretion.
- Minimize the binary incompatibility with the ATPCS.

3.2 Conformance

The AAPCS defines how separately compiled and separately assembled routines can work together. There is an *externally visible interface* between such routines. It is common that not all the externally visible interfaces to software are intended to be *publicly visible* or open to arbitrary use. In effect, there is a mismatch between the machine-level concept of external visibility—defined rigorously by an object code format—and a higher level, application-oriented concept of external visibility—which is system-specific or application-specific.

Conformance to the AAPCS requires that[1]:

- At all times, stack limits and basic stack alignment are observed (*§5.2.1.1 Universal stack constraints*).
- At each call where the control transfer instruction is subject to a BL-type relocation at static link time, rules on the use of IP are observed (*§5.3.1.1 Use of IP by the linker*).
- The routines of each publicly visible interface conform to the relevant procedure call standard variant.
- The data elements[2] of each publicly visible interface conform to the data layout rules.

[1] This definition of conformance gives maximum freedom to implementers. For example, if it is known that both sides of an externally visible interface will be compiled by the same compiler, and that the interface will not be publicly visible, the AAPCS permits the use of private arrangements across the interface such as using additional argument registers or passing data in non-standard formats. Stack invariants must, nevertheless, be preserved because an AAPCS-conforming routine elsewhere in the call chain might otherwise fail. Rules for use of IP must be obeyed or a static linker might generate a non-functioning executable program.

Conformance at a publicly visible interface does not depend on what happens behind that interface. Thus, for example, a tree of non-public, non-conforming calls can conform because the root of the tree offers a publicly visible, conforming interface and the other constraints are satisfied.

[2] *Data elements* include: parameters to routines named in the interface, static data named in the interface, and all data addressed by pointer values passed across the interface.

4 DATA TYPES AND ALIGNMENT

4.1 Fundamental Data Types

Table 1, Byte size and byte alignment of fundamental data types shows the fundamental data types of the machine. A NULL pointer is always represented by all-bits-zero.

Type Class	Machine Type	Byte size	Byte alignment	Note
Character	Unsigned byte	1	1	
	Signed byte	1	1	
Integral	Unsigned half-word	2	2	
	Signed half-word	2	2	
	Unsigned word	4	4	
	Signed word	4	4	
	Unsigned double-word	8	8	
	Signed double-word	8	8	
Floating Point	Single precision (IEEE 754)	4	4	The encoding of floating point numbers is described in [ARM ARM] chapter C2, *VFP Programmer's Model*, *§2.1.1 Single-precision format*, and *§2.1.2 Double-precision format*.
	Double precision (IEEE 754)	8	8	
Pointer	Data pointer	4	4	Pointer arithmetic should be unsigned.
	Code pointer	4	4	Bit 0 of a code pointer indicates the target instruction set type (0 ARM, 1 Thumb).

Table 1, Byte size and byte alignment of fundamental data types

4.2 Endianness and Byte Ordering

From a software perspective, memory is an array of bytes, each of which is addressable.

This ABI supports two views of memory implemented by the underlying hardware.

- In a little-endian view of memory the least significant byte of a data object is at the lowest byte address the data object occupies in memory.
- In a big-endian view of memory the least significant byte of a data object is at the highest byte address the data object occupies in memory.

The least significant bit in an object is always designated as *bit 0*.

The mapping of a word-sized data object to memory is shown *in Figure 1, Memory layout of big-endian data object* and *Figure 2, Memory layout of little-endian data object*. All objects are pure-endian, so the mappings may be scaled accordingly for larger or smaller objects[1].

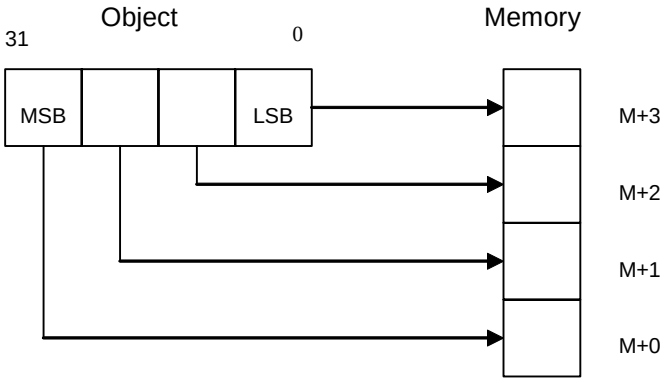

Figure 1, Memory layout of big-endian data object

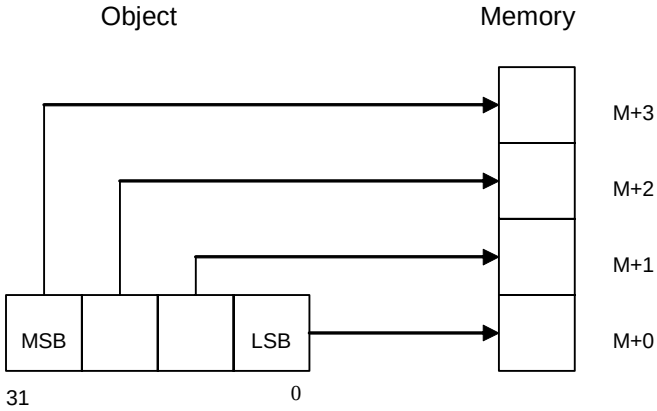

Figure 2, Memory layout of little-endian data object

4.3 Composite Types

A *Composite Type* is a collection of one or more Fundamental Data Types that are handled as a single entity at the procedure call level. A Composite Type can be any of:

- An *aggregate*, where the members are laid out sequentially in memory
- A *union*, where each of the members has the same address
- An *array*, which is a repeated sequence of some other type (its base type).

The definitions are recursive; that is, each of the types may contain a Composite Type as a member.

[1] The underlying hardware may not directly support a pure-endian view of data objects that are not naturally aligned.

4.3.1 Aggregates

- The alignment of an aggregate shall be the alignment of its most-aligned component.
- The size of an aggregate shall be the smallest multiple of its alignment that is sufficient to hold all of its members when they are laid out according to these rules.

4.3.2 Unions

- The alignment of a union shall be the alignment of its most-aligned component.
- The size of a union shall be the smallest multiple of its alignment that is sufficient to hold its largest member.

4.3.3 Arrays

- The alignment of an array shall be the alignment of its base type.
- The size of an array shall be the size of the base type multiplied by the number of elements in the array.

4.3.4 Bit-fields

A member of an aggregate that is a Fundamental Data Type may be subdivided into bit-fields; if there are unused portions of such a member that are sufficient to start the following member at its natural alignment then the following member may use the unallocated portion. For the purposes of calculating the alignment of the aggregate the type of the member shall be the Fundamental Data Type upon which the bit-field is based.[1] The layout of bit-fields within an aggregate is defined by the appropriate language binding.

4.3.5 Homogeneous Aggregates

A Homogeneous Aggregate is a Composite Type where all of the Fundamental Data Types that compose the type are the same.

[1] The intent is to permit the C construct `struct {int a:8; char b[7];}` to have size 8 and alignment 4.

5 THE BASE PROCEDURE CALL STANDARD

The base standard defines a machine-level, core-registers-only calling standard common to ARM and Thumb. It should be used for systems where there is no floating-point hardware, or where a high degree of inter-working with Thumb code is required.

5.1 Machine Registers

There are 16, 32-bit core (integer) registers visible to the ARM and Thumb instruction sets. These are labeled r0-r15 or R0-R15. Register names may appear in assembly language in either upper case or lower case. In this specification upper case is used when the register has a fixed role in the procedure call standard. *Table 2, Core registers and AAPCS usage* summarizes the uses of the core registers in this standard. In addition to the core registers there is one status register (CPSR) that is available for use in conforming code.

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Table 2, Core registers and AAPCS usage

The first four registers r0-r3 (a1-a4) are used to pass argument values into a subroutine and to return a result value from a function. They may also be used to hold intermediate values within a routine (but, in general, only *between* subroutine calls).

Register r12 (IP) may be used by a linker as a scratch register between a routine and any subroutine it calls (for details, see *§5.3.1.1, Use of IP by the linker*). It can also be used within a routine to hold intermediate values *between* subroutine calls.

The role of register r9 is platform specific. A virtual platform may assign any role to this register and must document this usage. For example, it may designate it as the static base (SB) in a position-independent data model, or it may designate it as the thread register (TR) in an environment with thread-local storage. The usage of this register may require that the value held is persistent across all calls. A virtual platform that has no need for such a special register may designate r9 as an additional callee-saved variable register, v6.

Typically, the registers r4-r8, r10 and r11 (v1-v5, v7 and v8) are used to hold the values of a routine's local variables. Of these, only v1-v4 can be used uniformly by the whole Thumb instruction set, but the AAPCS does not require that Thumb code only use those registers.

A subroutine must preserve the contents of the registers r4-r8, r10, r11 and SP (and r9 in PCS variants that designate r9 as v6).

In all variants of the procedure call standard, registers r12-r15 have special roles. In these roles they are labeled IP, SP, LR and PC.

The CPSR is a global register with the following properties:

- The N, Z, C, V and Q bits (bits 27-31) and the GE[3:0] bits (bits 16-19) are undefined on entry to or return from a public interface. The Q and GE[3:0] bits may only be modified when executing on a processor where these features are present.
- On ARM Architecture 6, the E bit (bit 8) can be used in applications executing in little-endian mode, or in big-endian-8 mode to temporarily change the endianness of data accesses to memory. An application must have a designated endianness and at entry to and return from any public interface the setting of the E bit must match the designated endianness of the application.
- The T bit (bit 5) and the J bit (bit 24) are the execution state bits. Only instructions designated for modifying these bits may change them.
- The A, I, F and M[4:0] bits (bits 0-7) are the privileged bits and may only be modified by applications designed to operate explicitly in a privileged mode.
- All other bits are reserved and must not be modified. It is not defined whether the bits read as zero or one, or whether they are preserved across a public interface.

5.1.1 Co-processor Registers

A machine's register set may be extended with additional registers that are accessed via instructions in the co-processor instruction space. To the extent that such registers are not used for passing arguments to and from subroutine calls the use of co-processor registers is compatible with the base standard. Each co-processor may provide an additional set of rules that govern the usage of its registers.

Note Even though co-processor registers are not used for passing arguments some elements of the run-time support for a language may require knowledge of all co-processors in use in an application in order to function correctly (for example, `setjmp()` in C and exceptions in C++).

5.1.1.1 VFP register usage conventions

The VFP co-processor has 32 single-precision registers, s0-s31, which may also be accessed as 16 double-precision registers, d0-d15 (with d0 overlapping s0, s1; d1 overlapping s2, s3; etc). In addition there are 3 or more system registers, depending on the implementation.

Registers s16-s31 (d8-d15) must be preserved across subroutine calls; registers s0-s15 (d0-d7) do not need to be preserved (and can be used for passing arguments or returning results in standard procedure-call variants).

The FPSCR is the only status register that may be accessed by conforming code. It is a global register with the following properties:

- The condition code bits (28-31) and the cumulative exception-status bits (0-4) are not preserved across a public interface.
- The exception-control bits (8-12), rounding mode bits (22-23) and flush-to-zero bits (24) may be modified by calls to specific support functions that affect the global state of the application.
- The length bits (16-18) and stride bits (20-21) must be zero on entry to and return from a public interface.
- All remaining bits must not be modified.

5.2 Processes, Memory and the Stack

The AAPCS applies to a single *thread of execution* or *process* (hereafter referred to as a process). A process has a *program state* defined by the underlying machine registers and the contents of the memory it can access. The memory a process can access, without causing a run-time fault, may vary during the execution of the process.

The memory of a process can normally be classified into five categories:

- code (the program being executed), which must be readable, but need not be writable, by the process.
- read-only static data.
- writable static data.
- the heap.
- the stack.

Writable static data may be further sub-divided into initialized, zero-initialized and uninitialized data. Except for the stack there is no requirement for each class of memory to occupy a single contiguous region of memory. A process must always have some code and a stack, but need not have any of the other categories of memory.

The heap is an area (or areas) of memory that are managed by the process itself (for example, with the C `malloc` function). It is typically used for the creation of dynamic data objects.

A conforming program must only execute instructions that are in areas of memory designated to contain code.

5.2.1 The Stack

The stack is a contiguous area of memory that may be used for storage of local variables and for passing additional arguments to subroutines when there are insufficient argument registers available.

The stack implementation is *full-descending*, with the current extent of the stack held in the register SP (r13). The stack will, in general, have both a *base* and a *limit* though in practice an application may not be able to determine the value of either.

The stack may have a fixed size or be dynamically extendable (by adjusting the stack-limit downwards).

The rules for maintenance of the stack are divided into two parts: a set of constraints that must be observed at all times, and an additional constraint that must be observed at a public interface.

5.2.1.1 Universal stack constraints

At all times the following basic constraints must hold:

- Stack-limit < SP <= stack-base. The stack pointer must lie within the extent of the stack.
- SP mod 4 = 0. The stack must at all times be aligned to a word boundary.
- A process may only access (for reading or writing) the closed interval of the entire stack delimited by [SP, stack-base – 1] (where SP is the value of register r13).

Note This implies that instructions of the following form can fail to satisfy the stack discipline constraints, even when `reg` points within the extent of the stack.

```
ldmxx    reg, {..., sp, ...}         // reg != sp
```

If execution of the instruction is interrupted after sp has been loaded, the stack extent will not be restored, so restarting the instruction might violate the third constraint.

5.2.1.2 Stack constraints at a public interface

The stack must also conform to the following constraint at a public interface:

- SP mod 8 = 0. The stack must be double-word aligned.

5.3 Subroutine Calls

Both the ARM and Thumb instruction sets contain a primitive subroutine call instruction, BL, which performs a branch-with-link operation. The effect of executing BL is to transfer the sequentially next value of the program counter—the *return address*—into the link register (LR) and the destination address into the program counter (PC). Bit 0 of the link register will be set to 1 if the BL instruction was executed from Thumb state, and to 0 if executed from ARM state. The result is to transfer control to the destination address, passing the return address in LR as an additional parameter to the called subroutine.

Control is returned to the instruction following the BL when the return address is loaded back into the PC (see *§5.6, Interworking*).

A subroutine call can be synthesized by any instruction sequence that has the effect:

```
        LR[31:1] ← return address
        LR[0]    ← code type at return address (0 ARM, 1 Thumb)
        PC       ← subroutine address
        ...
return address:
```

For example, in ARM-state, to call a subroutine addressed by r4 with control returning to the following instruction, do:

```
        MOV  LR, PC
        BX   r4
        ...
```

Note The equivalent sequence will not work from Thumb state because the instruction that sets LR does not copy the Thumb-state bit to LR[0].

In ARM Architecture v5 both ARM and Thumb state provide a BLX instruction that will call a subroutine addressed by a register and correctly sets the return address to the sequentially next value of the program counter.

5.3.1.1 Use of IP by the linker

Both the ARM- and Thumb-state BL instructions are unable to address the full 32-bit address space, so it may be necessary for the linker to insert a veneer between the calling routine and the called subroutine. Veneers may also be needed to support ARM-Thumb inter-working or dynamic linking. Any veneer inserted must preserve the contents of all registers except IP (r12); a conforming program must assume that a veneer that alters IP may be inserted at any branch instruction that is exposed to a relocation that supports inter-working or long branches.

Note Currently, `R_ARM_PC24` and `R_ARM_THUMB_PC22` (the BL-type relocations) are the only ELF relocation types with this property.

5.4 Result Return

The manner in which a result is returned from a function is determined by the type of that result.

For the base standard:

- A Fundamental Data Type that is smaller than 4 bytes is zero- or sign-extended to a word and returned in r0.
- A word-sized Fundamental Data Type (e.g., `int`, `float`) is returned in r0.
- A double-word sized Fundamental Data Type (e.g., `long long`, `double`) is returned in r0 and r1. R0 contains the low address half of the representation of the value in memory. Thus r0 contains the most significant half big-endian, the least significant half little-endian.
- A Composite Type not larger than 4 bytes is returned in r0. The format is as if the result had been stored in memory at a word-aligned address and then loaded into r0 with an LDR instruction. Any bits in r0 that lie outside the bounds of the result have unspecified values.
- A Composite Type larger than 4 bytes, or whose size cannot be determined statically by both caller and callee, is stored in memory at an address passed as an extra argument when the function was called.

5.5 Parameter Passing

The base standard provides for passing arguments in core registers (r0-r3) and on the stack. For subroutines that take a small number of parameters, only registers are used, greatly reducing the overhead of a call.

Parameter passing is defined as a two-level conceptual model

- A mapping from a source language argument onto a machine type
- The marshalling of machine types to produce the final parameter list

The mapping from the source language onto the machine type is specific for each language and is described separately (the C and C++ language bindings are described in *§7, ARM C and C++ language mappings*). The result is an ordered list of arguments that are to be passed to the subroutine.

In the following description there are assumed to be a number of co-processors available for passing and receiving arguments. The co-processor registers are divided into different classes. An argument may be a candidate for at most one co-processor register class. An argument that is suitable for allocation to a co-processor register is known as a Co-processor Register Candidate (CPRC).

In the base standard there are no arguments that are candidates for a co-processor register class.

A variadic function is always marshaled as for the base standard.

For a caller, sufficient stack space to hold stacked arguments is assumed to have been allocated prior to marshaling: in practice the amount of stack space required cannot be known until after the argument marshalling has been completed.

When a Composite Type argument is assigned to core registers (either fully or partially), the behavior is as if the argument had been stored to memory at a word-aligned (4-byte) address and then loaded into consecutive registers using a suitable load-multiple instruction.

Stage A – Initialization

This stage is performed exactly once, before processing of the arguments commences.

A.1 The Next Core Register Number (NCRN) is set to r0.

A.2.cp Co-processor argument register initialization is performed.

A.3 The next stacked argument address (NSAA) is set to the current stack-pointer value (SP).

A.4 If the subroutine is a function that returns a result in memory, then the address for the result is placed in r0 and the NCRN is set to r1.

Stage B – Pre-padding and extension of arguments

For each argument in the list the first matching rule from the following list is applied.

B.1 If the argument is a Composite Type whose size cannot be statically determined by both the caller and callee, the argument is copied to memory and the argument is replaced by a pointer to the copy.

B.2 If the argument is an integral Fundamental Data Type that is smaller than a word, then it is zero- or sign-extended to a full word and its size is set to 4 bytes.

B.3.cp If the argument is a CPRC then any preparation rules for that co-processor register class are applied.

B.4 If the argument is a Composite Type whose size is not a multiple of 4 bytes, then its size is rounded up to the nearest multiple of 4.

Stage C – Assignment of arguments to registers and stack

For each argument in the list the following rules are applied in turn until the argument has been allocated.

C.1.cp If the argument is a CPRC and there are sufficient unallocated co-processor registers of the appropriate class, the argument is allocated to co-processor registers.

C.2.cp If the argument is a CPRC then any co-processor registers in that class that are unallocated are marked as unavailable. The NSAA is adjusted upwards until it is correctly aligned for the argument and the argument is copied to the memory at the adjusted NSAA. The NSAA is further incremented by the size of the argument. The argument has now been allocated.

C.3 If the argument requires double-word alignment (8-byte), the NCRN is rounded up to the next even register number.

C.4 If the size in words of the argument is not more than r4 minus NCRN, the argument is copied into core registers, starting at the NCRN. The NCRN is incremented by the number of registers used. Successive registers hold the parts of the argument they would hold if its value were loaded into those registers from memory using an LDM instruction. The argument has now been allocated.

C.5 If the NCRN is less than r4 and the NSAA is equal to the SP, the argument is split between core registers and the stack. The first part of the argument is copied into the core registers starting at the NCRN up to and including r3. The remainder of the argument is copied onto the stack, starting at the NSAA. The NCRN is set to r4 and the NSAA is incremented by the size of the argument minus the amount passed in registers. The argument has now been allocated.

C.6 The NCRN is set to r4.

C.7 If the argument required double-word alignment (8-byte), then the NSAA is rounded up to the next double-word address.

C.8 The argument is copied to memory at the NSAA. The NSAA is incremented by the size of the argument.

It should be noted that the above algorithm makes provision for languages other than C and C++ in that it provides for passing arrays by value and for passing arguments of dynamic size. The rules are defined in a way that allows the caller to be always able to statically determine the amount of stack space that must be allocated for arguments that are not passed in registers, even if the function is variadic.

Several further observations can also be made:

- The initial stack slot address is the value of the stack pointer that will be passed to the subroutine. It may therefore be necessary to run through the above algorithm twice during compilation, once to determine the amount of stack space required for arguments and a second time to assign final stack slot addresses.
- A double-word aligned type will always start in an even-numbered register, or at a double-word aligned address on the stack even if it is not the first member of an aggregate.
- Arguments are allocated first to registers and only excess arguments are placed on the stack.
- Arguments that are Fundamental Data Types can either be entirely in registers or entirely on the stack.
- At most one argument can be split between registers and memory according to rule C.5.
- CPRCs may be allocated to co-processor registers or the stack – they may never be allocated to core registers.
- Since an argument may be a candidate for at most one class of co-processor register, then the rules for multiple co-processors (should they be present) may be applied in any order without affecting the behavior.
- An argument may only be split between core registers and the stack if all preceding CPRCs have been allocated to co-processor registers.

5.6 Interworking

The AAPCS requires that all sub-routine call and return sequences support inter-working between ARM and Thumb states. The implications on compiling for various ARM Architectures are as follows.

ARM v5 and ARM v6

Calls via function pointers should use one of the following, as appropriate:

```
blx  Rm    ; For normal sub-routine calls

bx   Rm    ; For tail calls
```

Calls to functions that use `bl<cond>`, b, or `b<cond>` will need a linker-generated veneer if a state change is required, so it may sometimes be more efficient to use a sequence that permits use of an unconditional `bl` instruction.

Return sequences may use load-multiple operations that directly load the PC or a suitable `bx` instruction.

The following traditional return must not be used if inter-working might be required.

```
mov  pc, Rm
```

ARM v4T

In addition to the constraints for ARM v5, the following additional restrictions apply to ARM v4T.

Calls using `bl` that involve a state change also require a linker-generated stub.

Calls via function pointers must use a sequence equivalent to the ARM-state code

```
mov  lr, pc
bx   Rm
```

However, this sequence does not work for Thumb state, so usually a `bl` to a veneer that does the `bx` instruction must be used.

Return sequences must restore any saved registers and then use a `bx` instruction to return to the caller.

ARM v4

The ARM v4 Architecture supports neither Thumb state nor the bx instruction, therefore it is not strictly compatible with the AAPCS.

It is recommended that code for ARM v4 be compiled using ARM v4T inter-working sequences but with all `bx` instructions subject to relocation by an `R_ARM_V4BX` relocation [AAELF]. A linker linking for ARM V4 can then change all instances of:

```
bx   Rm
```

Into:

```
mov  pc, Rm
```

But relocatable files remain compatible with this standard.

6 THE STANDARD VARIANTS

6.1 VFP Register Arguments

This variant alters the manner in which floating-point values are passed between a subroutine and its caller and allows significantly better performance when a VFP co-processor is present.

6.1.1 Procedure Calling

This section applies only to non-variadic functions. For a variadic function the base standard is always used both for argument passing and result return.

The set of call saved registers is the same as for the base standard (*§5.1.1.1, VFP register usage conventions*).

6.1.1.1 Result return

The manner in which a result is returned from a non-variadic function is changed as follows

- A single-precision floating point type is returned in s0.
- A double-precision floating point type is returned in d0.
- A homogeneous aggregate of a floating point type with one to four members is returned in s0-s*N* or d0-d*N* as appropriate to its type.

All other types are returned as for the base standard.

6.1.1.2 Parameter passing

For the VFP the following argument types are VFP CPRCs.

- - A single-precision floating-point type.
- - A double-precision floating-point type.
- - An Homogeneous Aggregate of a floating-point type with one to four members.

There is one VFP co-processor register class using registers s0-s15 (d0-d7) for passing arguments.

The following co-processor rules are defined for the VFP:

A.2.vfp The floating point argument registers are marked as unallocated.

B.3.vfp Nothing to do.

C.1.vfp If the argument is a VFP CPRC and there are sufficient consecutive VFP registers of the appropriate type unallocated then the argument is allocated to the lowest-numbered sequence of such registers.

C.2.vfp If the argument is a VFP CPRC then any VFP registers that are unallocated are marked as unavailable. The NSAA is adjusted upwards until it is correctly aligned for the argument and the argument is copied to the stack at the adjusted NSAA. The NSAA is further incremented by the size of the argument. The argument has now been allocated.

6.2 Read-Write Position Independence (RWPI)

Code compiled or assembled for execution environments that require read-write position independence (for example, the single address-space DLL-like model) use a static base to address writable data. Core register r9 is renamed as SB and used to hold the static base address: consequently this register may not be used for holding other values at any time[1].

6.3 Variant Compatibility

The variants described in *§6, The Standard Variants* can produce code that is incompatible with the base standard. Nevertheless, there still exist subsets of code that may be compatible across more than one variant. This section describes the theoretical levels of compatibility between the variants; however, whether a tool-chain must accept compatible objects compiled to different base standards, or correctly reject incompatible objects, is implementation defined.

6.3.1 VFP and Base Standard Compatibility

Code compiled for the VFP calling standard is compatible with the base standard (and *vice-versa*) if no floating-point arguments are used or floating-point results returned, or if the only routines that pass or return floating-point values are variadic routines

6.3.2 RWPI and Base Standard Compatibility

Code compiled for the base standard is compatible with the RWPI calling standard if it makes no use of register r9. However, a platform ABI may restrict further the subset of code that is usefully compatible.

6.3.3 VFP and RWPI Standard Compatibility

The VFP calling variant and RWPI addressing variant may be combined to create a third major variant. The appropriate combination of the rules described above will determine whether code is compatible.

[1] Although not mandated by this standard, compilers usually formulate the address of a static datum by loading the offset of the datum from SB, and adding SB to it. Usually, the offset is a 32-bit value loaded PC-relative from a literal pool. Usually, the literal value is subject to R_ARM_SBREL32-type relocation at static link time. The offset of a datum from SB is clearly a property of the layout of an executable, which is fixed at static link time. It does not depend on where the data is loaded, which is captured by the value of SB at run time.

7 ARM C AND C++ LANGUAGE MAPPINGS

This section describes how ARM compilers map C language features onto the machine-level standard. To the extent that C++ is a superset of the C language it also describes the mapping of C++ language features.

7.1 Data Types

7.1.1 Arithmetic Types

The mapping of C arithmetic types to Fundamental Data Types is shown in *Table 3, Mapping of C & C++ built-in data types*.

C/C++ Type	Machine Type	Notes
`char`	unsigned byte	LDRB is unsigned
`unsigned char`	unsigned byte	
`signed char`	signed byte	
`[signed] short`	signed halfword	
`unsigned short`	unsigned halfword	
`[signed] int`	signed word	
`unsigned int`	unsigned word	
`[signed] long`	signed word	
`unsigned long`	unsigned word	
`[signed] long long`	signed double-word	C99 Only
`unsigned long long`	unsigned double-word	C99 Only
`float`	single precision (IEEE 754)	
`double`	double precision (IEEE 754)	
`long double`	double precision (IEEE 754)	
`float _Imaginary`	single precision (IEEE 754)	C99 Only
`double _Imaginary`	double precision (IEEE 754)	C99 Only
`long double _Imaginary`	double precision (IEEE 754)	C99 Only
`float _Complex`	2 single precision (IEEE 754)	C99 Only. Layout is `struct {float re; float im;};`
`double _Complex`	2 double precision (IEEE 754)	C99 Only. Layout is `struct {double re; double im;};`

C/C++ Type	Machine Type	Notes
`long double _Complex`	2 double precision (IEEE 754)	C99 Only. Layout is `struct {long double re; long double im;};`
`_Bool/bool`	unsigned byte	C99/C++ Only
`wchar_t`	see text	built-in in C++, typedef in C, type is platform specific

Table 3, Mapping of C & C++ built-in data types

The preferred type of `wchar_t` is `unsigned word`. However, a virtual platform may elect to use `unsigned halfword` instead. A platform standard must document its choice.

7.1.2 Pointer Types

The container types for pointer types are shown in *Table 4, Pointer and reference types*. A C++ reference type is implemented as a pointer to the type.

Pointer Type	Machine Type	Notes
`T *`	data pointer	any data type T
`T (*F)()`	code pointer	any function type F
`T&`	data pointer	C++ reference

Table 4, Pointer and reference types

7.1.3 Enumerated Types

The type of the storage container for an enumerated type is the smallest integer type that contains all the enumerated values.

A conforming exported interface from a module shall not require support for long long enumeration types.

The container types for enumerators are shown in *Table 5, Enumerator container types*.

Lower bound	Upper bound	Container type
0…255	0…255	`unsigned char`
-128…-1	-128…127	`signed char`
0…65535	256…65535	`unsigned short`
-128…-1 -32768…-129	128…32767 -32768…32767	`short`
0…4294967295	65536…4294967295	`unsigned int`
-32768…-1 -2147483648…-32769	32768…2147483647 -2147483648…2147483647	`int`

Table 5, Enumerator container types

The size and alignment of an enumeration type shall be the size and alignment of the container type.

If a negative number is assigned to an enumeration type whose container type is unsigned the behavior is undefined.

7.1.4 Additional Types

Both C and C++ require that a system provide additional type definitions that are defined in terms of the base types. Normally these types are defined by inclusion of the appropriate header file. However, in C++ the underlying type of `size_t` can be exposed without the use of any header files simply by using `::operator new()`, and the definition of `va_list` has implications for the internal implementation in the compiler. An EABI conforming object must use the definitions shown in *Table 6, Additional data types*.

Typedef	Base type	Notes
`size_t`	`unsigned int`	For consistent C++ mangling of `::operator new()`
`va_list`	`struct __va_list { void *__ap; }`	A `va_list` may address any object in a parameter list. Consequently, the first object addressed may only have word alignment (all objects are at least word aligned), but any double-word aligned object will appear at the correct double-word alignment in memory.

Table 6, Additional data types

7.1.5 Volatile Data Types

A data type declaration may be qualified with the `volatile` type qualifier. The compiler may not remove any access to a volatile data type unless it can prove that the code containing the access will never be executed; however, a compiler may ignore a volatile qualification of an automatic variable whose address is never taken unless the function calls `setjmp()`. A volatile qualification on a structure or union shall be interpreted as applying the qualification recursively to each of the fundamental data types of which it is composed. Access to a volatile-qualified fundamental data type must always be made by accessing the whole type.

The behavior of assigning to or from an entire structure or union that contains volatile-qualified members is undefined. Likewise, the behavior is undefined if a cast is used to change either the qualification or the size of the type.

Not all ARM architectures provide for access to types of all widths; for example, prior to ARM Architecture 4 there were no instructions to access a 16-bit quantity, and similar issues apply to accessing 64-bit quantities. Further, the memory system underlying the processor may have a restricted bus width to some or all of memory. The only guarantee applying to volatile types in these circumstances are that each byte of the type shall be accessed exactly once for each access mandated above, and that any bytes containing volatile data that lie outside the type shall not be accessed. Nevertheless, if the compiler has an instruction available that will access the type exactly it should use it in preference to smaller or larger accesses.

7.1.6 Structure, Union and Class Layout

Structures and unions are laid out according to the Fundamental Data Types of which they are composed (see *§4.3, Composite Types*). All members are laid out in declaration order. Additional rules applying to C++ non-POD class layout are described in CPPABI.

7.1.7 Bit-fields

A bit-field may have any integral type (including enumerated and bool types). Bit-fields of non-enumerated types are treated as unsigned unless explicitly declared signed.

A sequence of bit-fields is laid out in the order declared using the rules below.

For each bit-field, the type of its container is its declared type and contributes to the alignment of the containing aggregate in the same way a plain (not bit-field) member of that type would, without exception for zero-sized or anonymous bit-fields.

Note The C++ standard states that an anonymous bit-field is not a member, so it is unclear whether or not an anonymous bit-field of non-zero size should contribute to the aggregate's alignment. However, a bit-field of zero size clearly must, and such a field must be anonymous. Thus the following example has the same size, but a different alignment, when `T` is `char`, `short`, and `int` (C++ allows `char :16`).

```
struct FourBytes { char a; T :16; char d;};
```

The contents of each bit-field is contained by exactly one instance of its container type.

Initially, we define the layout of fields that are no bigger than their container types.

7.1.7.1 Bit-fields no larger than their container

Let `F` be a bit-field whose address we wish to determine. We define the container address, `CA(F)`, to be the byte address

```
CA(F) = &(container(F));
```

This address will always be at the natural alignment of the container type, that is

```
CA(F) % sizeof(container(F)) == 0.
```

The bit-offset of `F` within the container, `K(F)`, is defined in an endian-dependent manner:

- For big-endian data types `K(F)` is the offset from the most significant bit of the container to the most significant bit of the bit-field.
- For little-endian data types `K(F)` is the offset from the least significant bit of the container to the least significant bit of the bit-field.

A bit-field can be extracted by loading its container, shifting and masking by amounts that depend on the byte order, `K(F)`, the container size, and the field width, then sign extending if needed.

The bit-address of `F`, `BA(F)`, can now be defined as:

```
BA(F) = CA(F) * 8 + K(F)
```

For a bit address `BA` falling in a container of width `C` and alignment `A` ($\leq$ `C`) (both expressed in bits), define the unallocated container bits (`UCB`) to be:

```
UCB(BA, C, A) = C - (BA % A)
```

We further define the truncation function

```
TRUNCATE(X,Y) = Y * ⌊X/Y⌋
```

That is, the largest integral multiple of `Y` that is no larger than `X`.

We can now define the next container bit address (`NCBA`) which will be used when there is insufficient space in the current container to hold the next bit-field as

```
NCBA(BA, A) = TRUNCATE(BA + A – 1, A)
```

At each stage in the laying out of a sequence of bit-fields there is:

- A current bit address (CBA)
- A container size, C, and alignment, A, determined by the type of the field about to be laid out (8, 16, 32, …)
- A field width, W (≤ C).

For each bit-field, F, in declaration order the layout is determined by

1 If the field width, W, is zero, set CBA = NCBA(CBA, A)

2 If W > UCB(CBA, C, A), set CBA = NCBA(CBA, A)

3 Assign BA(F) = CBA

4 Set CBA = CBA + W.

Note The EABI does not allow exported interfaces to contain packed structures or bit-fields. However a scheme for laying out packed bit-fields can be achieved by reducing the alignment, A, in the above rules to below that of the natural container type. ARMCC uses an alignment of A=8 in these cases, but GCC uses an alignment of A=1.

7.1.7.2 Bit-field extraction expressions

To access a field, F, of width W and container width C at the bit-address BA(F):

- Load the (naturally aligned) container at byte address TRUNCATE(BA(F), C) / 8 into a register R (or two registers if the container is 64-bits)
- Set Q = MAX(32, C)
- Little-endian, set R = (R << ((Q – W) – (BA MOD C))) >> (Q – W).
- Big-endian, set R = (R << (BA MOD C)) >> (Q – W).

The long long bit-fields use shifting operations on 64-bit quantities; it may often be the case that these expressions can be simplified to use operations on a single 32-bit quantity (but see *§7.1.7.5, Volatile bit-fields—preserving number and width of container accesses* for volatile bit-fields).

7.1.7.3 Over-sized bit-fields

C++ permits the width specification of a bit-field to exceed the container size, but specifies that any excess bits of the container size only contribute to padding: it does not say where the padding occurs. The padding rules below are selected on the basis of:

- Minimal additional padding.
- Facilitating access to the value with accesses that are naturally aligned for the type.
- The same sequence of fields and holes in both big- and little-endian systems.

Hence for a bit-field with width, W (> C), we reduce W according to the formula

```
W = MAX(CBA + W – NCBA(CBA, A), C)
```

and then apply steps 2–4 of *§7.1.7.1, Bit-fields no larger than their container* using the modified width. The effect of these rules is that excess bits are used in preference for pre-padding.

An oversized bit-field can be accessed simply by accessing its container type.

7.1.7.4 Combining bit-field and non-bit-field members

A bit-field container may overlap a non-bit-field member. For the purposes of determining the layout of bit-field members the CBA will be the address of the first unallocated bit after the preceding non-bit-field type.

Note Any tail-padding added to a structure that immediately precedes a bit-field member is part of the structure and must be taken into account when determining the CBA.

When a non-bit-field member follows a bit-field it is placed at the lowest acceptable address following the allocated bit-field.

Note When laying out fundamental data types it is possible to consider them all to be bit-fields with a width equal to the container size. The rules in *§7.1.7.1, Bit-fields no larger than their container* can then be applied to determine the precise address within a structure.

7.1.7.5 Volatile bit-fields—preserving number and width of container accesses

When a volatile bit-field is read, its container must be read exactly once using the access width appropriate to the type of the container.

When a volatile bit-field is written, its container must be read exactly once and written exactly once using the access width appropriate to the type of the container. The two accesses are not atomic.

Multiple accesses to the same volatile bit-field, or to additional volatile bit-fields within the same container may not be merged. For example, an increment of a volatile bit-field must always be implemented as two reads and a write.

Note Note the volatile access rules apply even when the width and alignment of the bit-field imply that the access could be achieved more efficiently using a narrower type. For a write operation the read must always occur even if the entire contents of the container will be replaced.

If the containers of two volatile bit-fields overlap then access to one bit-field will cause an access to the other. For example, in `struct S {volatile int a:8; volatile char b:2};` an access to a will also cause an access to b, but not vice-versa.

If the container of a non-volatile bit-field overlaps a volatile bit-field then it is undefined whether access to the non-volatile field will cause the volatile field to be accessed.

7.2 Argument Passing Conventions

The argument list for a subroutine call is formed by taking the user arguments in the order in which they are specified.

- For C, each argument is formed from the value specified in the source code, except that an array is passed by passing the address of its first element.
- For C++, an implicit `this` parameter is passed as an extra argument that immediately precedes the first user argument. Other rules for marshalling C++ arguments are described in CPPABI.
- For variadic functions, `float` arguments that match the ellipsis (…) are converted to type `double`.

The argument list is then processed according to the standard rules for procedure calls (see *§5.5, Parameter Passing*) or the appropriate variant.