Procedure Call Standard for the Arm Architecture

Release 2019Q4

Arm Ltd

Jan 31, 2020

IHI0042I_2019Q4

CONTENTS

IHI0042I_2019Q4

Document number: IHI 0042I, current through ABI release 2019Q4

Date of Issue: 30th January 2020

CHAPTER

ONE

PREAMBLE

1.1 Abstract

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

1.2 Keywords

Procedure call, function call, calling conventions, data layout

1.3 How to find the latest release of this specification or report a defect in it

Please check the Arm Developer site (https://developer.arm.com/products/software-development-tools/specifications) for a later release if your copy is more than one year old.

Please report defects in this specification to *arm* dot *eabi* at *arm* dot *com*.

1.4 Licence

1.5 Non-Confidential Proprietary Notice

1.6 Contents

Contents

IHI0042I_2019Q4

CHAPTER

TWO

ABOUT THIS DOCUMENT

2.1 Change Control

2.1.1 Current Status and Anticipated Changes

The following support level definitions are used by the Arm ABI specifications:

Release Arm considers this specification to have enough implementations, which have received sufficient testing, to verify that it is correct. The details of these criteria are dependent on the scale and complexity of the change over previous versions: small, simple changes might only require one implementation, but more complex changes require multiple independent implementations, which have been rigorously tested for cross-compatibility. Arm anticipates that future changes to this specification will be limited to typographical corrections, clarifications and compatible extensions.

Beta Arm considers this specification to be complete, but existing implementations do not meet the requirements for confidence in its release quality. Arm may need to make incompatible changes if issues emerge from its implementation.

Alpha The content of this specification is a draft, and Arm considers the likelihood of future incompatible changes to be significant.

All content in this document is at the **Release** quality level.

2.1.2 Change History

Issue	Date	By	Change
1.0	30th October 2003	LS	First public release.
2.0	24th March 2005	LS	Second public release.
2.01	5th July 2005	LS	Added clarifying remark following *Table 5* (page 33) – word-sized enumeration contains are `int` if possible (*Enumerated Types* (page 32))
2.02	4th August 2005	RE	Clarify that a callee may modify stack space used for incoming parameters.
2.03	7th October 2005	LS	Added notes concerning VFPv3 D16-D31 (*VFP register usage conventions* (page 20)); retracted requirement that plain bit-fields be unsigned by default (*Bit-fields* (page 34))
2.04	4th May 2006	RE	Clarified when linking may insert veneers that corrupt r12 and the condition codes (*Use of IP by the linker* (page 23)).
2.05	19th January 2007	RE	Update for the Advanced SIMD Extension.
2.06	2nd October 2007	RE	Add support for half-precision floating point.
A	25th October 2007	LS	Document renumbered (formerly GENC-003534 v2.06).

Continued on next page

Table 2.1 – continued from previous page

Issue	Date	By	Change
B	2nd April 2008	RE	Simplify duplicated text relating to VFP calling and clarify that homogeneous aggregates of containerized vectors are limited to four members in calling convention (*VFP co-processor register candidates* (page 28)).
C	10th October 2008	RE	Clarify that __va_list is in namespace std. Specify containers for oversized enums. State truth values for _Bool/bool. Clarify some wording with respect to homogeneous aggregates and argument marshalling of VFP CPRCs.
D	16th October 2009	LS	Re-wrote *Enumerated Types* (page 32) to better reflect the intentions for enumerated types in ABI-complying interfaces.
E 2.09	30th November 2012	AC	Clarify that memory passed for a function result may be modified at any point during the function call (*Result Return* (page 23)). Changed the illustrative source name of the half-precision float type from __f16 to __fp16 to match [ACLE[1]] (*Arithmetic Types* (page 31)). Re-wrote *APPENDIX Support for Advanced SIMD Extensions and MVE* (page 38) to clarify requirements on Advanced SIMD types.
F	24th October 2015	CR	*SIMD vector data types* (page 38), corrected the element counts of poly16x4_t and poly16x8_t. Added [u]int64x1_t, [u]int64x2_t, poly64x2_t. Allow half-precision floating point types as function parameter and return types, by specifying how half-precision floating point types are passed and returned in registers *Result Return* (page 23), *Parameter Passing* (page 24), *Mapping between registers and memory format* (page 28), *VFP co-processor register candidates* (page 28)). Added parameter passing rules for over-aligned types (*Composite Types* (page 15), *Parameter Passing* (page 24)).
2018Q4	21st December 2018	OS	In *Volatile bit-fields – preserving number and width of container accesses* (page 36), relaxed the rules regarding accesses to volatile bitfield members to be compatible with the C/C++ memory model. In *Stack probing* (page 22), relaxed the rules regarding stack accesses to permit stack probing. In *VFP register usage conventions* (page 20), corrected the rules regarding the values of the IDC and IDE bits of the FPSCR register on a public interface.
2019Q4	28th January 2020	TS	Be more specific on the use of frame pointers and frame records. (*The Frame Pointer* (page 22), *Machine Registers* (page 18)). Add description of half-precision Brain floating-point format (*Half-precision Floating Point* (page 14), *Arm Alternative Format Half-precision Floating Point values* (page 29), *Arithmetic Types* (page 31)). For clarity, renamed half-precision format 'Alternative' to 'Arm Alternative' (*Half-precision Floating Point* (page 14), *Arm Alternative Format Half-precision Floating Point values* (page 29), *Half-precision Format Compatibility* (page 30), *Table 3, Mapping of C & C++ built-in data types* (page 31)).

IHI0042I_2019Q4

2.2 References

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

Ref	External URL	Title
AAPCS (page 1)	This document	Procedure Call Standard for the Arm Architecture
AAELF		ELF for the Arm Architecture
BSABI		ABI for the Arm Architecture (Base Standard)
CPPABI		C++ ABI for the Arm Architecture
ARMARM[2]	Arm DDI 0100E, ISBN 0 201 737191 https://developer.arm.com/docs/ddi0100/latest/armv5-architecture-reference-manual	The Arm Architecture Reference Manual 2nd edition, edited by David Seal, published by Addison-Wessley.
	Arm DDI 0406 https://developer.arm.com/docs/ddi0406/c/arm-architecture-reference-manual-armv7-a-and-armv7-r-edition	Arm Architecture Reference Manual Arm v7-A and Arm v7-R edition
ACLE[3]	IHI 0053A	Arm C Language Extensions
GCPPABI[4]	http://itanium-cxx-abi.github.io/	Generic C++ ABI

2.3 Terms and Abbreviations

This document uses the following terms and abbreviations.

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.

[1] https://developer.arm.com/products/software-development-tools/compilers/arm-compiler-5/docs/101028/latest/1-preface
[2] https://developer.arm.com/docs/ddi0406/c/arm-architecture-reference-manual-armv7-a-and-armv7-r-edition
[3] https://developer.arm.com/products/software-development-tools/compilers/arm-compiler-5/docs/101028/latest/1-preface
[4] http://itanium-cxx-abi.github.io/cxx-abi/abi.html

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 concept:*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.

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).

Thumb-1 The variant of the Thumb instruction set introduced in Arm v4T and used in Arm v6-M and the Arm v8-M.Baseline variants of the architecture. It consists of instructions that are predominantly encoded with 16-bit opcodes.

Thumb-2 The variant of the Thumb instruction set introduced in Arm v6T2. It consists of a mix of instructions encoded with 16- and 32-bit opcodes.

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.

2.4 Your licence to use this specification

IMPORTANT: THIS IS A LEGAL AGREEMENT ("LICENCE") BETWEEN YOU (AN INDIVIDUAL OR SINGLE ENTITY WHO IS RECEIVING THIS DOCUMENT DIRECTLY FROM ARM LIMITED) ("LICENSEE") AND ARM LIMITED ("ARM") FOR THE SPECIFICATION DEFINED IMMEDIATELY BELOW. BY DOWNLOADING OR OTHERWISE USING IT, YOU AGREE TO BE BOUND BY ALL OF THE TERMS OF THIS LICENCE. IF YOU DO NOT AGREE TO THIS, DO NOT DOWNLOAD OR USE THIS SPECIFICATION.

"Specification" means, and is limited to, the version of the specification for the Applications Binary Interface for the Arm Architecture comprised in this document. Notwithstanding the foregoing, "Specification" shall not include (i) the implementation of other published specifications referenced in this Specification; (ii) any enabling technologies that may be necessary to make or use any product or portion thereof that complies with this Specification, but are not themselves expressly set forth in this Specification (e.g. compiler front ends, code generators, back ends, libraries or other compiler, assembler or linker technologies; validation or debug software or hardware; applications, operating system or driver software; RISC architecture; processor microarchitecture); (iii) maskworks and physical layouts of integrated circuit designs; or (iv) RTL or other high level representations of integrated circuit designs.

Use, copying or disclosure by the US Government is subject to the restrictions set out in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software – Restricted Rights at 48 C.F.R. 52.227-19, as applicable.

This Specification is owned by Arm or its licensors and is protected by copyright laws and international copyright treaties as well as other intellectual property laws and treaties. The Specification is licensed not sold.

1. Subject to the provisions of Clauses 2 and 3, Arm hereby grants to LICENSEE, under any intellectual property that is (i) owned or freely licensable by Arm without payment to unaffiliated third parties and (ii) either embodied in the Specification or Necessary to copy or implement an applications binary interface compliant with this Specification, a perpetual, non-exclusive, non-transferable, fully paid, worldwide limited licence (without the right to sublicense) to use and copy this Specification solely for the purpose of developing, having developed, manufacturing, having manufactured, offering to sell, selling, supplying or otherwise distributing products which comply with the Specification.
2. THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES EXPRESS, IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF SATISFACTORY QUALITY, MERCHANTABILITY, NONINFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE. THE SPECIFICATION MAY INCLUDE ERRORS. Arm RESERVES THE RIGHT TO INCORPORATE MODIFICATIONS TO THE SPECIFICATION IN LATER REVISIONS OF IT, AND TO MAKE IMPROVEMENTS OR CHANGES IN THE SPECIFICATION OR THE PRODUCTS OR TECHNOLOGIES DESCRIBED THEREIN AT ANY TIME.
3. This Licence shall immediately terminate and shall be unavailable to LICENSEE if LICENSEE or any party affiliated to LICENSEE asserts any patents against Arm, Arm affiliates, third parties who have a valid licence from Arm for the Specification, or any customers or distributors of any of them based upon a claim that a LICENSEE (or LICENSEE affiliate) patent is Necessary to implement the Specification. In this Licence; (i) "affiliate" means any entity controlling, controlled by or under common control with a party (in fact or in law, via voting securities, management control or otherwise) and "affiliated" shall be construed accordingly; (ii) "assert" means to allege infringement in legal or administrative proceedings, or proceedings before any other competent trade, arbitral or international authority; (iii) "Necessary" means with respect to any claims of any patent, those claims which, without the appropriate permission of the patent owner, will be infringed when implementing the Specification because no alternative, commercially reasonable, non-infringing way of implementing the Specification is known; and (iv) English law and the jurisdiction of the English courts shall apply to all aspects of this Licence, its interpretation and enforcement. The total liability of Arm and any of its suppliers and licensors under or in relation to this Licence shall be limited to the greater of the amount actually paid by LICENSEE for the Specification or US$10.00. The limitations, exclusions and disclaimers in this Licence shall apply to the maximum extent allowed

IHI0042I_2019Q4

by applicable law.

Arm Contract reference LEC-ELA-00081 V2.0 AB/LS (9 March 2005)

2.5 Acknowledgements

This specification has been developed with the active support of the following organizations. In alphabetical order: Arm, CodeSourcery, Intel, Metrowerks, Montavista, Nexus Electronics, PalmSource, Symbian, Texas Instruments, and Wind River.

CHAPTER

THREE

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.

IHI0042I_2019Q4

CHAPTER

FOUR

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.

4.1 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.

4.2 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[11]:

- At all times, stack limits and basic stack alignment are observed (*Universal stack constraints* (page 21)).
- 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 (*Use of IP by the linker* (page 23)).
- The routines of each publicly visible interface conform to the relevant procedure call standard variant.
- The data elements[12] of each publicly visible interface conform to the data layout rules.

[11] 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.

[12] *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.

CHAPTER

FIVE

DATA TYPES AND ALIGNMENT

5.1 Fundamental Data Types

Table 1, Byte size and byte alignment of fundamental data types (page 14) shows the fundamental data types (Machine Types) of the machine. A NULL pointer is always represented by all-bits-zero.

Table 5.1: Table 1, Byte size and byte alignment of fundamental data types

Type Class	Machine Type	Byte size	Byte align-ment	Note
Integral	Unsigned byte	1	1	Character
	Signed byte	1	1	
	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	Half precision	2	2	See *Half-precision Floating Point* (page 14).
	Single precision (IEEE 754)	4	4	The encoding of floating point numbers is described in [ARMARM[5]] 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	
Containterized vector	64-bit vector	8	8	See *Containerized Vectors* (page 15).
	128-bit vector	16	8	
Pointer	Data pointer	4	4	Pointer arithmetic should be unsigned. Bit 0 of a code pointer indicates the target instruction set type (0 Arm, 1 Thumb).
	Code pointer	4	4	

5.1.1 Half-precision Floating Point

Optional extensions to the Arm architecture provide hardware support for half-precision values. Three formats are currently supported:

1 - half-precision format specified in IEEE754-2008

2 - Arm Alternative format, which provides additional range but has no NaNs or Infinities.

3 - Brain floating-point format, which provides a dynamic range similar to the 32-bit floating-point format, but with less precision.

[5] https://developer.arm.com/docs/ddi0406/c/arm-architecture-reference-manual-armv7-a-and-armv7-r-edition

The first two formats are mutually exclusive. The base standard of the AAPCS specifies use of the IEEE754-2008 variant, and a procedure call variant that uses the Arm Alternative format is permitted.

5.1.2 Containerized Vectors

The content of a containerized vector is opaque to most of the procedure call standard: the only defined aspect of its layout is the mapping between the memory format (the way a fundamental type is stored in memory) and different classes of register at a procedure call interface. If a language binding defines data types that map directly onto the containerized vectors it will define how this mapping is performed.

5.2 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 *Memory layout of big-endian data object* (page 15) and *Memory layout of little-endian data object* (page 16). All objects are pure-endian, so the mappings may be scaled accordingly for larger or smaller objects[13].

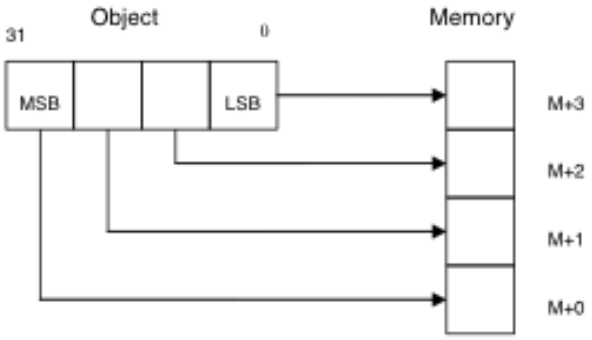

Fig. 5.1: Memory layout of big-endian data object

5.3 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.

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

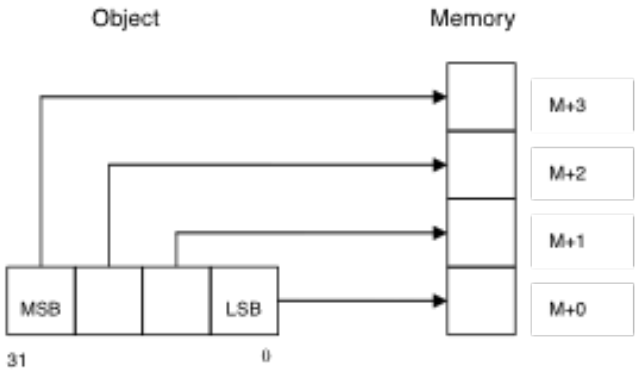

Fig. 5.2: Memory layout of little-endian data object

- The *member alignment* of an element of a composite type is the alignment of that member after the application of any language alignment modifiers to that member
- The *natural alignment* of a composite type is the maximum of each of the member alignments of the 'top-level' members of the composite type i.e. before any alignment adjustment of the entire composite is applied

5.3.1 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.

5.3.2 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.

5.3.3 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.

5.3.4 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.[14] The layout of bit-fields within an aggregate is defined by the appropriate language binding.

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

5.3.5 Homogeneous Aggregates

A Homogeneous Aggregate is a Composite Type where all of the Fundamental Data Types that compose the type are the same. The test for homogeneity is applied after data layout is completed and without regard to access control or other source language restrictions.

An aggregate consisting of containerized vector types is treated as homogeneous if all the members are of the same size, even if the internal format of the containerized members are different. For example, a structure containing a vector of 8 bytes and a vector of 4 half-words satisfies the requirements for a homogeneous aggregate.

A Homogeneous Aggregate has a Base Type, which is the Fundamental Data Type of each *Element*. The overall size is the size of the Base Type multiplied by the number of Elements; its alignment will be the alignment of the Base Type.

IHI0042I_2019Q4

CHAPTER

SIX

THE BASE PROCEDURE CALL STANDARD

The base standard defines a machine-level, core-registers-only calling standard common to the Arm and Thumb instruction sets. 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.

6.1 Machine Registers

The Arm architecture defines a core instruction set plus a number of additional instructions implemented by co-processors. The core instruction set can access the core registers and co-processors can provide additional registers which are available for specific operations.

6.1.1 Core 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* (page 18) 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.

Table 6.1: Table 2, Core registers and AAPCS usage

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	FP	Frame Pointer or 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.

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 *Use of IP by the linker* (page 23)). It can also be used within a routine to hold intermediate values between subroutine calls.

In some variants r11 (FP) may be used as a frame pointer in order to chain frame activation records into a linked list.

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.

Handling values larger than 32 bits

Fundamental types larger than 32 bits may be passed as parameters to, or returned as the result of, function calls. When these types are in core registers the following rules apply:

- A double-word sized type is passed in two consecutive registers (e.g., r0 and r1, or r2 and r3). The content of the registers is as if the value had been loaded from memory representation with a single `LDM` instruction.
- A 128-bit containerized vector is passed in four consecutive registers. The content of the registers is as if the value had been loaded from memory with a single `LDM` instruction.

6.1.2 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

IHI0042I_2019Q4

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++).

VFP register usage conventions

The VFP-v2 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. VFP-v3 adds 16 more double-precision registers d16-d31, but there are no additional single-precision counterparts. The Advanced SIMD Extension and the M-profile vector Extension (MVE) use the VFP register set. The Advanced SIMD Extension uses the double-precision registers for 64-bit vectors and further defines quad-word registers (with q0 overlapping d0, d1; and q1 overlapping d2, d3; etc) for 128-bit vectors. MVE uses 128-bit vectors in the same quad-word registers.

Registers s16-s31 (d8-d15, q4-q7) must be preserved across subroutine calls; registers s0-s15 (d0-d7, q0-q3) do not need to be preserved (and can be used for passing arguments or returning results in standard procedure-call variants). Registers d16-d31 (q8-q15), if present, do not need to be preserved.

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

- The condition code bits (28-31), the cumulative saturation (QC) bit (27) and the cumulative exception-status bits (0-4 and 7) are not preserved across a public interface.
- The exception-control bits (8-12 and 15), 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) must be 0b100 when using M-profile Vector Extension, 0b000 when using VFP vector mode and otherwise preserved across a public interface.
- The stride bits (20-21) must be zero on entry to and return from a public interface.
- 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.

VPR is a global register with the following properties:

- The VPT mask bits (16-23) must be zero on entry to and return from a public interface.
- The predication bits (0-15) are not preserved across a public interface.
- 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.

6.2 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.

IHI0042I_2019Q4

- 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.

6.2.1 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.

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 store data in 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.

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.

IHI0042I_2019Q4

Stack probing

In order to ensure stack integrity a process may emit stack probes immediately prior to allocating additional stack space (moving SP from SP_old to SP_new). Stack probes must be in the region of [SP_new, SP_old - 1] and may be either read or write operations. The minimum interval for stack probing is defined by the target platform but must be a minimum of 4KBytes. No recoverable data can be saved below the currently allocated stack region.

The Frame Pointer

A platform may require the construction of a list of stack frames describing the current call hierarchy in a program.

Each frame shall link to the frame of its caller by means of a Frame Record of two 32-bit values on the stack. The frame record for the innermost frame (belonging to the most recent routine invocation) shall be pointed to by the Frame Pointer register (FP). The lowest addressed word shall point to the previous frame record and the highest addressed word shall contain the value passed in LR on entry to the current function. The end of the frame record chain is indicated by the address zero in the address for the previous frame. The location of the frame record within a stack frame is not specified. The frame pointer register must not be updated until the new frame record has been fully constructed.

Note: There will always be a short period during construction or destruction of each frame record during which the frame pointer will point to the caller's record.

A platform shall mandate the minimum level of conformance with respect to the maintenance of frame records. The options are, in decreasing level of functionality:

- It may require the frame pointer to address a valid frame record at all times, except that small subroutines which do not modify the link register may elect not to create a frame record
- It may require the frame pointer to address a valid frame record at all times, except that any subroutine may elect not to create a frame record
- It may permit the frame pointer register to be used as a general-purpose callee-saved register, but provide a platform-specific mechanism for external agents to reliably locate the chain of frame records
- It may elect not to maintain a frame chain and to use the frame pointer register as a general-purpose callee-saved register.

Note: Unlike the APCS and its variants, the same frame pointer register is used for both the Arm and Thumb ISAs (including the Thumb-1 variant), this ensures that the frame chain can be constructed even when generating code that interworks between both the Arm and Thumb instruction sets. It is expected that Thumb-1 code will rarely, if ever, want to create stack frames - the choice of a high register therefore ensures that such code can conform minimally to the requirements of having a valid value stored in the frame pointer register without noticably reducing the number of registers available to normal code.

The AAPCS does not specify where, within a function's stack frame record, the frame chain data structure resides. This permits implementors the freedom to use whatever location will result in the most efficient code needed to establish the frame chain record. As a result, even in Thumb-1, the overhead for establishing the frame will rarely exceed three additional instructions in the function entry sequence and two additional instructions in the return sequence.

IHI0042I_2019Q4

6.3 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 *Interworking* (page 26)).

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.

6.3.1 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) and the condition code flags; 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: `R_ARM_CALL`, `R_ARM_JUMP24`, `R_ARM_PC24`, `R_ARM_THM_CALL`, `R_ARM_THM_JUMP24` and `R_ARM_THM_JUMP19` are examples of the ELF relocation types with this property. See [AAELF] for full details.

6.4 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 Half-precision Floating Point Type is returned in the least significant 16 bits of r0.

IHI0042I_2019Q4

- 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` and 64-bit containerized vectors) is returned in r0 and r1.
- A 128-bit containerized vector is returned in r0-r3.
- 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 (*Parameter Passing* (page 24), *rule A.4* (page 25)). The memory to be used for the result may be modified at any point during the function call.

6.5 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 *Arm C and C++ Language Mappings* (page 31)). 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. A callee can modify any stack space used for receiving parameter values from the caller.

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).

Continued on next page

IHI0042I_2019Q4

Table 6.2 – continued from previous page

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. If the argument is a Half-precision Floating Point Type its size is set to 4 bytes as if it had been copied to the least significant bits of a 32-bit register and the remaining bits filled with unspecified values.
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.
B.5	If the argument is an alignment adjusted type its value is passed as a copy of the actual value. The copy will have an alignment defined as follows. • For a Fundamental Data Type, the alignment is the natural alignment of that type, after any promotions. • For a Composite Type, the alignment of the copy will have 4-byte alignment if its natural alignment is <= 4 and 8-byte alignment if its natural alignment is >= 8 The alignment of the copy is used for applying marshaling rules.

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.

Continued on next page

Table 6.4 – continued from previous page

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 core 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* (page 25).
- 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.

6.6 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.

CHAPTER

SEVEN

THE STANDARD VARIANTS

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.

7.1 VFP and SIMD vector 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, the Advanced SIMD Extension or the M-profile Vector Extension is present.

7.1.1 Mapping between registers and memory format

Values passed across a procedure call interface in VFP registers are laid out as follows:

- A half precision floating point type is passed as if it were loaded from its memory format into the least significant 16 bits of a single precision register.
- A single precision floating point type is passed as if it were loaded from its memory format into a single precision register with `VLDR`.
- A double precision floating point type is passed as if it were loaded from its memory format into a double precision register with `VLDR`.
- A 64-bit containerized vector type is passed as if it were loaded from its memory format into a 64-bit vector register (D*n*) with `VLDR`.
- A 128-bit containerized vector type is passed as if it were loaded from its memory format into a 128-bit vector register (Q*n*) with a single `VLDM` of the two component 64-bit vector registers (for example, `VLDM r0,{d2,d3}` would load q1).

7.1.2 Procedure Calling

The set of call saved registers is the same as for the base standard (*VFP register usage conventions* (page 20)).

VFP co-processor register candidates

For the VFP the following argument types are VFP CPRCs.

- A half-precision floating-point type.
- A single-precision floating-point type.
- A double-precision floating-point type.
- A 64-bit or 128-bit containerized vector type.

- A Homogeneous Aggregate with a Base Type of a single- or double-precision floating-point type with one to four Elements.
- A Homogeneous Aggregate with a Base Type of 64-bit containerized vectors with one to four Elements.
- A Homogeneous Aggregate with a Base Type of 128-bit containerized vectors with one to four Elements.

Note: There are no VFP CPRCs in a variadic procedure.

Result return

Any result whose type would satisfy the conditions for a VFP CPRC is returned in the appropriate number of consecutive VFP registers starting with the lowest numbered register (s0, d0, q0).

All other types are returned as for the base standard.

Parameter passing

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.

Note that the rules require the 'back-filling' of unused co-processor registers that are skipped by the alignment constraints of earlier arguments. The back-filling continues only so long as no VFP CPRC has been allocated to a slot on the stack.

7.2 Arm Alternative Format Half-precision Floating Point values

Code may be compiled to use the Arm Alternative format Half-precision values. The rules for passing and returning values will either use the Base Standard rules or the VFP and SIMD vector register rules.

7.3 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[15].

[15] 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

7.4 Variant Compatibility

The variants described in *The Standard Variants* (page 28) 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.

7.4.1 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 or containerized vector arguments or results are used, or if the only routines that pass or return such values are variadic routines.

7.4.2 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.

7.4.3 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.

7.4.4 Half-precision Format Compatibility

The set of values that can be represented in Arm Alternative format differs from the set that can be represented in IEEE754-2008 format rendering code built to use either format incompatible with code that uses the other. Never-the-less, most code will make no use of either format and will therefore be compatible with both variants.

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.

IHI0042I_2019Q4

CHAPTER

EIGHT

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.

8.1 Data Types

8.1.1 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* (page 31).

Table 8.1: 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
`__fp16`	half precision (IEEE754-2008 or Arm Alternative)	Arm extension documented in [ACLE[6]]. In a variadic function call this will be passed as a double-precision value.
`__bf16`	half precision Brain floating-point format	Arm extension documented in [ACLE[7]].
`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; };`

Continued on next page

Table 8.1 – continued from previous page

C/C++ Type	Machine Type	Notes
`double _Complex`	2 double precision (IEEE 754)	C99 Only. Layout is `struct { double re; double im; };`
`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. False has value 0 and True has value 1.
`wchar_t`	see text	built-in in C++, typedef in C, type is platform specific

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

8.1.2 Pointer Types

The container types for pointer types are shown in *Table 4, Pointer and reference types* (page 32). A C++ reference type is implemented as a pointer to the type.

Table 8.2: Table 4, Pointer and reference types

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

8.1.3 Enumerated Types

This ABI delegates a choice of representation of enumerated types to a platform ABI (whether defined by a standard or by custom and practice) or to an interface contract if there is no defined platform ABI.

The two permitted ABI variants are:

- An enumerated type normally occupies a word (`int` or `unsigned int`). If a word cannot represent all of its enumerated values the type occupies a double word (`long long` or `unsigned long long`).
- The type of the storage container for an enumerated type is the smallest integer type that can contain all of its enumerated values.

When both the signed and unsigned versions of an integer type can represent all values, this ABI recommends that the unsigned type should be preferred (in line with common practice).

Discussion

The definition of enumerated types in the C and C++ language standards does not define a binary interface and leaves open the following questions.

- Does the container for an enumerated type have a fixed size (as expected in most OS environments) or is the size no larger than needed to hold the values of the enumeration (as expected by most embedded users)?
- What happens when a (strictly, non-conforming) enumerated value (e.g. MAXINT+1) overflows a fixed-size (e.g. `int`) container?

[6] https://developer.arm.com/products/software-development-tools/compilers/arm-compiler-5/docs/101028/latest/1-preface
[7] https://developer.arm.com/products/software-development-tools/compilers/arm-compiler-5/docs/101028/latest/1-preface

- Is a value of enumerated type (after any conversion required by C/C++) signed or unsigned?

In relation to the last question the C and C++ language standards state:

- **[C]** Each enumerated type shall be compatible with an integer type. The choice of type is implementation-defined, but *shall be capable of representing the values of all the members of the enumeration*.
- **[C++]** An enumerated type is **not** an integral type but ... An rvalue of... enumeration type (7.2) can be converted to an rvalue of the first of the following types that can represent all the values of its underlying type: `int`, `unsigned int`, `long`, or `unsigned long`.

Under this ABI, these statements allow a header file that describes the interface to a portable binary package to force its clients, in a portable, strictly-conforming manner, to adopt a 32-bit signed (`int/long`) representation of values of enumerated type (by defining a negative enumerator, a positive one, and ensuring the range of enumerators spans more than 16 bits but not more than 32).

Otherwise, a common interpretation of the binary representation must be established by appealing to a platform ABI or a separate interface contract.

8.1.4 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 AAPCS conforming object must use the definitions shown in *Table 5, Additional data types* (page 33).

Table 8.3: Table 5, 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. In C++, `__va_list` is in namespace `std`.

8.1.5 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.

8.1.6 Structure, Union and Class Layout

Structures and unions are laid out according to the Fundamental Data Types of which they are composed (see *Composite Types* (page 15)). All members are laid out in declaration order. Additional rules applying to C++ non-POD class layout are described in [CPPABI] and [GCPPABI[8]].

8.1.7 Bit-fields

A bit-field may have any integral type (including enumerated and bool types).

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 if its size is no larger than the size of its declared type.
- The largest integral type no larger than its size if its size is larger than the size of its declared type (see *Over-sized bit-fields* (page 36)).

The container type 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 an aggregate's alignment. Under this ABI it does.

The content 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.

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.

[8] http://itanium-cxx-abi.github.io/cxx-abi/abi.html

- 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 * $\lfloor$X/Y$\rfloor$

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 ($\leq$ 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 AAPCS 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.

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 *Volatile bit-fields – preserving number and width of container accesses* (page 36)).

Over-sized bit-fields

C++ permits the width specification of a bit-field to exceed the container size and the rules for allocation are given in [GCPPABI[9]]. Using the notation described above, the allocation of an over-sized bit-field of width `W`, for a container of width `C` and alignment `A` is achieved by:

- Selecting a new container width `C'` which is the width of the fundamental integer data type with the largest size less than or equal to `W`. The alignment of this container will be `A'`. Note that `C' >= C` and `A' >= A`.
- If `C' > UCB(CBA, C', A')` setting `CBA = NCBA(CBA, A')`. This ensures that the bit-field will be placed at the start of the next container type.
- Allocating a normal (undersized) bit-field using the values (`C`, `C'`, `A'`) for (`W`, `C`, `A`).
- Setting `CBA = CBA + W  C`.

Note: Although standard C++ does not have a `long long` data type, this is a common extension to the language. To avoid the presence of this type changing the layout of oversized bit-fields the above rules are described in terms of the fundamental machine types (*Fundamental Data Types* (page 14)) where a 64-bit integer data type always exists.

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

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 *Bit-fields no larger than their container* (page 34) can then be applied to determine the precise address within a structure.

Volatile bit-fields – preserving number and width of container accesses

When a volatile bit-field is read, and its container does not overlap with any non-bit-field member, 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, and its container does not overlap with any non-bit-field member, 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.

Note: This ABI does not place any restrictions on the access widths of bit-fields where the container overlaps with a non-bit-field member. This is because the C/C++ memory model defines these as being separate memory locations, which can be accessed by two threads simultaneously. For this reason, compilers must be permitted to use a narrower memory access width (including splitting the access into multiple instructions) to avoid writing to a different memory location. For example, in `struct S { int`

[9] http://itanium-cxx-abi.github.io/cxx-abi/abi.html

`a:24; char b; }`; a write to `a` must not also write to the location occupied by `b`, this requires at least two memory accesses in all current Arm architectures.

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.

8.2 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 *Parameter Passing* (page 24)) or the appropriate variant.

IHI0042I_2019Q4

CHAPTER

NINE

APPENDIX SUPPORT FOR ADVANCED SIMD EXTENSIONS AND MVE

9.1 Introduction

The Advanced SIMD and M-profile Vector Extension to the Arm architecture add support for processing short vectors. Because the C and C++ languages do not provide standard types to represent these vectors, access to them is provided by a vendor extension. The status of this appendix is normative in respect of public binary interfaces, i.e. the calling convention and name mangling of functions which use these types. In other respects it is informative.

9.2 SIMD vector data types

Access to the SIMD vector data types is obtained by including either of the two following header files: `arm_neon.h`, `arm_mve.h`. These headers provide the following features:

- They provide a set of user-level type names that map onto short vector types
- They provide prototypes for intrinsic functions that map onto the Advanced SIMD and M-profile Vector Extension(MVE) intruction sets respectively.

Note: The intrinsic functions are beyond the scope of this specification. Details of the usage of the user-level types (e.g. initialization, and automatic conversions) are also beyond the scope of this specification. For further details see [ACLE[10]].

Note: The user-level types are listed in *Table 6: Advanced SIMD Extension only vector data types using 64-bit containerized vectors* (page 39) and *Table 7: SIMD vector data types using 128-bit containerized vectors* (page 40). The types have 64-bit alignment and map directly onto the containerized vector fundamental data types. The memory format of the containerized vector is defined as loading the specified registers from an array of the Base Type using the Fill Operation and then storing that value to memory using a single `VSTM` of the loaded 64-bit (D) registers.

MVE only allows 128-bit vector types and it uses unsigned integer vectors to represent polynomials.

The tables also list equivalent structure types to be used for name mangling. Whether these types are actually defined by an implementation is unspecified.

---

[10] https://developer.arm.com/products/software-development-tools/compilers/arm-compiler-5/docs/101028/latest/1-preface

Table 9.1: Table 6: Advanced SIMD Extension only vector data types using 64-bit containerized vectors

User type name	Equivalent type name for mangling	Elements	Base type	Fill operation
`int8x8_t`	`struct __simd64_int8_t`	8	signed byte	`VLD1.8 {Dn}, [Rn]`
`int16x4_t`	`struct __simd64_int16_t`	4	signed half-word	`VLD1.16 {Dn}, [Rn]`
`int32x2_t`	`struct __simd64_int32_t`	2	signed word	`VLD1.32 {Dn}, [Rn]`
`int64x1_t`	`struct __simd64_int64_t`	1	signed double-word	`VLD1.64 {Dn}, [Rn]`
`uint8x8_t`	`struct __simd64_uint8_t`	8	unsigned byte	`VLD1.8 {Dn}, [Rn]`
`uint16x4_t`	`struct __simd64_uint16_t`	4	unsigned half-word	`VLD1.16 {Dn}, [Rn]`
`uint32x2_t`	`struct __simd64_uint32_t`	2	unsigned word	`VLD1.32 {Dn}, [Rn]`
`uint64x1_t`	`struct __simd64_uint64_t`	1	unsigned double-word	`VLD1.64 {Dn}, [Rn]`
`float16x4_t`	`struct __simd64_float16_t`	4	half-precision float	`VLD1.16 {Dn}, [Rn]`
`float32x2_t`	`struct __simd64_float32_t`	2	single-precision float	`VLD1.32 {Dn}, [Rn]`
`poly8x8_t`	`struct __simd64_poly8_t`	8	8-bit polynomial over GF(2)	`VLD1.8 {Dn}, [Rn]`
`poly16x4_t`	`struct __simd64_poly16_t`	4	16-bit polynomial over GF(2)	`VLD1.16 {Dn}, [Rn]`

Table 9.2: Table 7: SIMD vector data types using 128-bit containerized vectors

User type name	Equivalent type name for mangling	Elements	Base type	Fill operation
int8x16_t	struct __simd128_int8_t	16	signed byte	VLD1.8 {Qn}, [Rn]
int16x8_t	struct __simd128_int16_t	8	signed half-word	VLD1.16 {Qn}, [Rn]
int32x4_t	struct __simd128_int32_t	4	signed word	VLD1.32 {Qn}, [Rn]
int64x2_t	struct __simd128_int64_t	2	signed double-word	VLD1.64 {Qn}, [Rn]
uint8x16_t	struct __simd128_uint8_t	16	unsigned byte	VLD1.8 {Qn}, [Rn]
uint16x8_t	struct __simd128_uint16_t	8	unsigned half-word	VLD1.16 {Qn}, [Rn]
uint32x4_t	struct __simd128_uint32_t	4	unsigned word	VLD1.32 {Qn}, [Rn]
uint64x2_t	struct __simd128_uint64_t	2	unsigned double-word	VLD1.64 {Qn}, [Rn]
float32x4_t	struct __simd128_float32_t	4	single-precision float	VLD1.32 {Qn}, [Rn]
poly8x16_t	struct __simd128_poly8_t	16	8-bit polynomial over GF(2)	VLD1.8 {Qn}, [Rn]
poly16x8_t	struct __simd128_poly16_t	8	16-bit polynomial over GF(2)	VLD1.16 {Qn}, [Rn]
poly64x2_t	struct __simd128_poly64_t	2	64-bit polynomial over GF(2)	VLD1.64 {Qn}, [Rn]

9.2.1 C++ Mangling

For C++ the mangled name for parameters is as though the equivalent type name was used. For example,

```
void f(int8x8_t)
```

is mangled as

```
_Z1f15__simd64_int8_t
```