



**Procedure Call Standard for the
ARM 64-bit Architecture
(AArch64)
with SVE support**

Document number:

ARM_100986_0000_00_en (SVEpcs 00bet1)

Date of Issue:

Based on: ARM IHI 0055B, current through AArch64 ABI release 1.0

30th March 2017

Abstract

This document describes the Procedure Call Standard use by the Application Binary Interface (ABI) for the ARM 64-bit architecture. [It includes support for the Scalable Vector Extension \(SVE\).](#)

Keywords

Procedure call function call, calling conventions, data layout

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

Please check the *ARM Information Center* (<http://infocenter.arm.com/>) for a later release if your copy is more than 3 months old (navigate to the *Software Development Tools* section, *Application Binary Interface for the ARM Architecture* subsection). Please report defects in this specification to *arm dot eabi* at *arm dot com*.

Licence

THE TERMS OF YOUR ROYALTY FREE LIMITED LICENCE TO USE THIS ABI SPECIFICATION ARE GIVEN IN SECTION **1.4, *Your licence to use this specification*** (ARM contract reference **LEC-ELA-00081 V2.0**). PLEASE READ THEM CAREFULLY.

BY DOWNLOADING OR OTHERWISE USING THIS SPECIFICATION, YOU AGREE TO BE BOUND BY ALL OF ITS TERMS. IF YOU DO NOT AGREE TO THIS, DO NOT DOWNLOAD OR USE THIS SPECIFICATION.

THIS ABI SPECIFICATION IS PROVIDED “AS IS” WITH NO WARRANTIES (SEE SECTION 1.4 FOR DETAILS).

Proprietary notice

ARM, Thumb, RealView, ARM7TDMI and ARM9TDMI are registered trademarks of ARM Limited. The ARM logo is a trademark of ARM Limited. ARM9, ARM926EJ-S, ARM946E-S, ARM1136J-S ARM1156T2F-S ARM1176JZ-S Cortex, and Neon are trademarks of ARM Limited. All other products or services mentioned herein may be trademarks of their respective owners.

Contents

1	About this document	5
1.1	Change Control	5
1.1.1	Current Status and Anticipated Changes	5
1.1.2	Change History	5
1.2	References	5
1.3	Terms and Abbreviations	5
1.4	Your licence to use this specification	7
2	Scope	9
3	Introduction	10
3.1	Design Goals	10
3.2	Conformance	10
4	Data Types and alignment	11
4.1	Fundamental Data Types	11
4.1.1	Half-precision Floating Point	12
4.1.2	Short Vectors	12
4.1.3	Scalable Vectors	12
4.1.4	Scalable Predicates	13
4.1.5	Pointers	13
4.2	Byte Order (“Endianness”)	13
4.3	Composite Types	14
4.3.1	Aggregates	14
4.3.2	Unions	14
4.3.3	Arrays	14
4.3.4	Bit-fields	14
4.3.5	Homogeneous Aggregates	14
4.3.6	Scalable-Vector Aggregates (SVAs)	15
5	The Base PROCEDURE CALL Standard	16
5.1	Machine Registers	16
5.1.1	General-purpose Registers	16
5.1.2	SIMD and Floating-Point Registers	17
5.1.3	Scalable Vector Registers	17
5.1.4	Scalable Predicate Registers	18
5.2	Processes, Memory and the Stack	18
5.2.1	Memory Addresses	18
5.2.2	The Stack	18
5.2.3	The Frame Pointer	19
5.3	Subroutine Calls	19
5.4	Parameter Passing	20
5.4.1	Variadic Subroutines	20
5.4.2	Parameter Passing Rules	20
5.5	Result Return	22
5.6	Interworking	22
6	The Standard Variants	23
6.1	Half-precision Format Compatibility	23

6.2	Sizeof(long), sizeof(wchar_t)	23
7	ARM C and C++ language mappings	24
7.1	Data Types	24
7.1.1	Arithmetic Types	24
7.1.2	Pointer Types	25
7.1.3	Enumerated Types	25
7.1.4	Additional Types	26
7.1.5	Volatile Data Types	26
7.1.6	Structure, Union and Class Layout	26
7.1.7	Bit-fields	26
7.2	Argument Passing Conventions	29
Appendix A	C and C++ Support for SIMD Extensions	30
Appendix B	C and C++ Support for Scalable Vectors	32
Appendix C	C++ Mangling	33
Appendix D	Variable Argument Lists	34
D.1	Register Save Areas	34
D.2	The va_list type	34
D.3	The va_start() macro	34
D.4	The va_arg() macro	35

1 ABOUT THIS DOCUMENT

1.1 Change Control

1.1.1 Current Status and Anticipated Changes

This document's status is released for the base specification and is in beta for the SVE specification. Clarifications, extensions and minor changes should be expected for the base specification, while larger changes are possible but unlikely for the SVE specification.

1.1.2 Change History

Issue	Date	By	Change
00Bet3	25 th November 2011	RE	Beta release
1.0	22 nd May 2013	RE	First public release
SVEpcs 00bet1	30th March 2017	RS	First public beta release of the SVE specification

1.2 References

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

Ref	URL or other reference	Title
AAPCS64	IHI 0056C	Procedure Call Standard for the ARM 64-bit Architecture
CPPABI64	IHI 0059	C++ ABI for the ARM 64-bit Architecture
GC++ABI	http://mentorembdedd.github.io/cxx-abi/abi.html	Generic C++ ABI

1.3 Terms and Abbreviations

The *ABI for the ARM 64-bit Architecture* uses the following terms and abbreviations.

Term	Meaning
A32	The instruction set named <i>ARM</i> in the ARMv7 architecture; A32 uses 32-bit fixed-length instructions.
A64	The instruction set available when in AArch64 state.
AAPCS64	Procedure Call Standard for the ARM 64-bit Architecture (AArch64)
AArch32	The 32-bit general-purpose register width state of the ARMv8 architecture, broadly compatible with the ARMv7-A architecture.
AArch64	The 64-bit general-purpose register width state of the ARMv8 architecture.

Term	Meaning
ABI	Application Binary Interface: <ol style="list-style-type: none"> 1. The specifications to which an executable must conform in order to execute in a specific execution environment. For example, the <i>Linux ABI for the ARM Architecture</i>. 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 <i>C++ ABI for the ARM Architecture</i>, <i>ELF for the ARM Architecture</i>, ...
ARM-based	... based on the ARM architecture ...
Floating point	Depending on context floating point means or qualifies: (a) floating-point arithmetic conforming to IEEE 754 2008; (b) the ARMv8 floating point instruction set; (c) the register set shared by (b) and the ARMv8 SIMD instruction set.
Q-o-I	Quality of Implementation – a quality, behavior, functionality, or mechanism not required by this standard, but which might be provided by systems conforming to it. Q-o-I is often used to describe the tool-chain-specific means by which a standard requirement is met.
SIMD	Single Instruction Multiple Data – A term denoting or qualifying: (a) processing several data items in parallel under the control of one instruction; (b) the ARM v8 SIMD instruction set; (c) the register set shared by (b) and the ARMv8 floating point instruction set.
SIMD and floating point	The ARM architecture’s SIMD and Floating Point architecture comprising the floating point instruction set, the SIMD instruction set and the register set shared by them.
<u>SVE</u>	<u>Scalable Vector Extension</u>
T32	The instruction set named Thumb in the ARMv7 architecture; T32 uses 16-bit and 32-bit instructions.

This document uses the following terms and abbreviations.

Term	Meaning
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. <i>Routine</i> is used for clarity where there are nested calls: a routine is the <i>caller</i> and a subroutine is the <i>callee</i> .
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).
PIC, PID	Position-independent code, position-independent data.
Argument, Parameter	The terms <i>argument</i> and <i>parameter</i> 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, Caller-saved 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). If a function needs to preserve the value held in such a register over a call to another function, then the calling function must save and restore the value.
Callee-saved register	A register whose value must be preserved over a function call. If the function being called (the callee) needs to use the register, then it is responsible for saving and restoring the old value.
SVR4	System V Revision 4. A variant of the Unix Operating System. Although this specification refers to SVR4, many other operating systems, such as Linux or BSD use similar rules.
Platform	A program execution environment such as that defined by an operating system or run-time environment. A platform defines the specific variant of the ABI and may impose additional constraints. Linux is a platform in this sense.

More specific terminology is defined when it is first used.

1.4 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 by applicable law.

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

2 SCOPE

The AAPCS64 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, or between a routine and its execution environment, 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.
- Obligations on all routines to preserve certain global invariants.

This standard specifies the base for a family of *Procedure Call Standard (PCS)* variants generated by choices that reflect arbitrary, but historically important choices, among:

- Byte order.
- Size of long int and wchar_t and the format of half-precision floating-point values (here we define a SVR4-like variant and a Windows-like variant only – see sections 6 and 7 for details).
- Whether floating-point operations use floating-point hardware resources or are implemented by calls to integer-only routines¹.

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 *CPPABI64*) and it places no requirements on the representation of language entities that are not visible across public interfaces.

¹ This base standard requires that AArch64 floating-point resources be used by floating-point operations and floating-point parameter passing. However, it is acknowledged that operating system code often prefers not to perturb the floating-point state of the machine and to implement its own limited use of floating-point in integer-only code: such code is permitted, but not conforming.

3 INTRODUCTION

The AAPCS64 is the first revision of Procedure Call standard for the ARM 64-bit Architecture. It forms part of the complete ABI specification for the ARM 64-bit Architecture.

3.1 Design Goals

The goals of the AAPCS64 are to:

- Support efficient execution on high-performance implementations of the ARM 64-bit Architecture.
- Clearly distinguish between mandatory requirements and implementation discretion.

3.2 Conformance

The AAPCS64 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 AAPCS64 requires that¹:

- At all times, stack limits and basic stack alignment are observed (§5.2.2.1 *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 IPO and IP1 are observed (§5.3.1.1 *Use of IPO and IP1 by the linker*).
- The routines of each publicly visible interface conform to the relevant procedure call standard variant.
- The data elements² 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 AAPCS64 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 AAPCS64-conforming routine elsewhere in the call chain might otherwise fail. Rules for use of IPO and IP1 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 pointers 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 (Machine Types) of the machine.

Type Class	Machine Type	Byte size	Natural Alignment (bytes)	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	
	Unsigned quad-word	16	16	
	Signed quad-word	16	16	
Floating Point	Half precision	2	2	See §4.1.1, <i>Half-precision Floating Point</i> .
	Single precision	4	4	IEEE 754-2008
	Double precision	8	8	
	Quad precision	16	16	
Short vector	64-bit vector	8	8	See §4.1.2, <i>Short Vectors</i> .
	128-bit vector	16	16	
Scalable Vector	VGx64-bit vector of 8-bit elements	VGx8	16	See §4.1.3, Scalable Vectors
	VGx64-bit vector of 16-bit elements			

Type Class	Machine Type	Byte size	Natural Alignment (bytes)	Note
	VG×64-bit vector of 32-bit elements			
	VG×64-bit vector of 64-bit elements			
Scalable Predicate	VG×8-bit predicate	VG	2	See §4.1.4, Scalable Predicates
Pointer	Data pointer	8	8	See §4.1.5, Pointers.
	Code pointer	8	8	

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

4.1.1 Half-precision Floating Point

The architecture provides hardware support for half-precision values. Two formats are currently supported: the format specified in IEEE 754-2008 and an Alternative format that provides additional range but has no NaNs or Infinities. This base standard of the AAPCS64 specifies two variants:

- The SVR4-like variant uses the IEEE 754-2008 defined format.
- The Windows-like variant uses ...[TBC]

4.1.2 Short Vectors

A short vector is a machine type that is composed of repeated instances of one fundamental integral or floating-point type. It may be 8 or 16 bytes in total size. A short vector has a *base type* that is the fundamental integral or floating-point type from which it is composed, but its alignment is always the same as its total size. The number of elements in the short vector is always such that the type is fully packed. For example, an 8-byte short vector may contain 8 unsigned byte elements, 4 unsigned half-word elements, 2 single-precision floating-point elements, or any other combination where the product of the number of elements and the size of an individual element is equal to 8. Similarly, for 16-byte short vectors the product of the number of elements and the size of the individual elements must be 16.

Elements in a short vector are numbered such that the lowest numbered element (element 0) occupies the lowest numbered bit (bit zero) in the vector and successive elements take on progressively increasing bit positions in the vector. When a short vector transferred between registers and memory it is treated as an opaque object. That is a short vector is stored in memory as if it were stored with a single `STR` of the entire register; a short vector is loaded from memory using the corresponding `LDR` instruction. On a little-endian system this means that element 0 will always contain the lowest addressed element of a short vector; on a big-endian system element 0 will contain the highest-addressed element of a short vector.

A language binding may define extended types that map directly onto short vectors. Short vectors are not otherwise created spontaneously (for example because a user has declared an aggregate consisting of eight consecutive byte-sized objects).

4.1.3 Scalable Vectors

[Like a short vector \(§4.1.2\), a scalable vector is a machine type that is composed of repeated instances of one fundamental integral or floating-point type. The number of bytes in the vector is always \$VG \times 8\$, where \$VG\$ is a runtime value determined by the execution environment. \$VG\$ is an even integer greater than or equal to 2; the ABI does not define an upper bound. \$VG\$ is the same for all scalable vector and predicate types.](#)

Each element of a scalable vector has a zero-based index. When stored in memory, the elements are placed in index order, so that element N comes before element $N+1$. The layout of each individual element is the same as if it were scalar. When stored in a scalable vector register, element 0 contains bit 0 of the corresponding short vector register (note that this holds even for big-endian systems).

4.1.4 Scalable Predicates

A scalable predicate is a machine type that is composed of VG bytes, with one bit for each byte of a scalable vector.

Each bit of a scalable predicate has a zero-based index. When stored in memory, index 0 is placed in the least significant bit of the first byte, index 1 is stored in the next significant bit, and so on.

4.1.5 Pointers

Code and data pointers are 64-bit unsigned types. A NULL pointer is always represented by all-bits-zero.

All 64 bits in a pointer are always significant. When tagged addressing is enabled, a tag is part of a pointer's value for the purposes of pointer arithmetic. The result of subtracting or comparing two pointers with different tags is unspecified. See also §5.2.1, below.

4.2 Byte Order (“Endianness”)

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 objects¹.

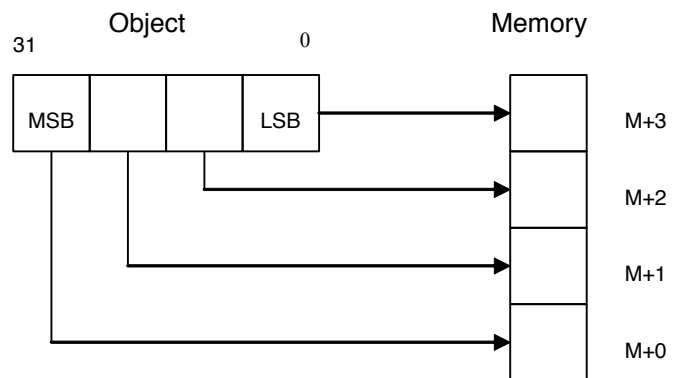


Figure 1, Memory layout of big-endian data object

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

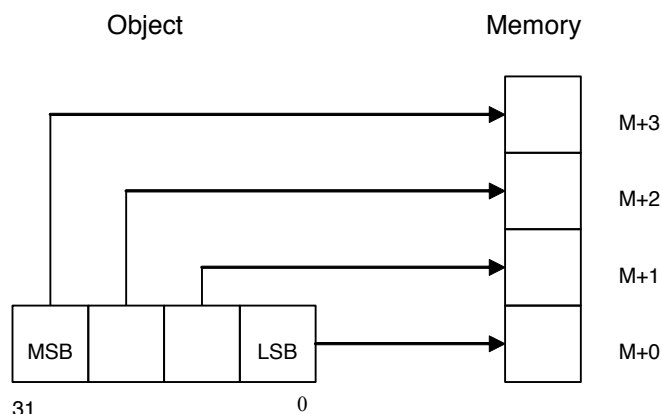


Figure 2, Memory layout of little-endian data object

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 (possibly with inter-member padding)
- 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.

4.3.1 Aggregates

- The alignment of an aggregate shall be the alignment of its most-aligned member.
- The size of an aggregate shall be the smallest multiple of its alignment that is sufficient to hold all of its members.

4.3.2 Unions

- The alignment of a union shall be the alignment of its most-aligned member.
- 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 of the members that compose the type are the same. The test for homogeneity is applied after data layout is completed and without

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

regard to access control or other source language restrictions. Note that for short-vector types the fundamental types are 64-bit vector and 128-bit vector; the type of the elements in the short vector does not form part of the test for homogeneity.

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

4.3.5.1 Homogeneous Floating-point Aggregates (HFA)

A Homogeneous Floating-point Aggregate (HFA) is a Homogeneous Aggregate with a Fundamental Data Type that is a Floating-Point type and at most four uniquely addressable members.

4.3.5.2 Homogeneous Short-Vector Aggregates (HVA)

A Homogeneous Short-Vector Aggregate (HVA) is a Homogeneous Aggregate with a Fundamental Data Type that is a Short-Vector type and at most four uniquely addressable members.

4.3.6 Scalable-Vector Aggregates (SVAs)

A Scalable Vector Aggregate is an aggregate in which the Fundamental Data Type of each member is a Scalable Vector Type or a Scalable Predicate Type. The aggregate does not need to be homogeneous. For example, an aggregate that contains a scalable vector of 8-bit elements, a scalable predicate, and a scalable vector of 16-bit elements is a Scalable Vector Aggregate.

5 THE BASE PROCEDURE CALL STANDARD

The base standard defines a machine-level calling standard for the A64 instruction set. It assumes the availability of the vector registers for passing floating-point and SIMD arguments. Application code is expected to conform to one of the two defined major variants of it (SVR4-like or Windows-like).

5.1 Machine Registers

The ARM 64-bit architecture defines two mandatory register banks: a general-purpose register bank which can be used for scalar integer processing and pointer arithmetic; and a SIMD and Floating-Point register bank. [In addition, the architecture defines an optional set of scalable vector registers that overlap the SIMD and Floating-Point register bank, accompanied by a set of scalable predicate registers.](#)

5.1.1 General-purpose Registers

There are thirty-one, 64-bit, general-purpose (integer) registers visible to the A64 instruction set; these are labeled r0-r30. In a 64-bit context these registers are normally referred to using the names x0-x30; in a 32-bit context the registers are specified by using w0-w30. Additionally, a stack-pointer register, SP, can be used with a restricted number of instructions. 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 this procedure call standard. *Table 2, General purpose registers and AAPCS64 usage* summarizes the uses of the general-purpose registers in this standard. In addition to the general-purpose registers there is one status register (NZCV) that may be set and read by conforming code.

Register	Special	Role in the procedure call standard
SP		The Stack Pointer.
r30	LR	The Link Register.
r29	FP	The Frame Pointer
r19...r28		Callee-saved registers
r18		The Platform Register, if needed; otherwise a temporary register. See notes.
r17	IP1	The second intra-procedure-call temporary register (can be used by call veneers and PLT code); at other times may be used as a temporary register.
r16	IP0	The first intra-procedure-call scratch register (can be used by call veneers and PLT code); at other times may be used as a temporary register.
r9...r15		Temporary registers
r8		Indirect result location register
r0...r7		Parameter/result registers

Table 2, General purpose registers and AAPCS64 usage

The first eight registers, r0-r7, are used to pass argument values into a subroutine and to return result values from a function. They may also be used to hold intermediate values within a routine (but, in general, only *between* subroutine calls).

Registers r16 (IP0) and r17 (IP1) 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 IP0 and IP1 by the linker*). They can also be used within a routine to hold intermediate values *between* subroutine calls.

The role of register r18 is platform-specific. If a platform ABI has need of a dedicated general purpose register to carry inter-procedural state (for example, the thread context) then it should use this register for that purpose. If the platform ABI has no such requirements, then it should use r18 as an additional temporary register. The platform ABI specification must document the usage for this register.

Note Software developers creating platform-independent code are advised to avoid using r18 if at all possible. Most compilers provide a mechanism to prevent specific registers from being used for general allocation; portable hand-coded assembler should avoid it entirely. It should not be assumed that treating the register as callee-saved will be sufficient to satisfy the requirements of the platform. Virtualization code must, of course, treat the register as they would any other resource provided to the virtual machine.

A subroutine invocation must preserve the contents of the registers r19-r29 and SP.

In all variants of the procedure call standard, registers r16, r17, r29 and r30 have special roles. In these roles they are labeled IP0, IP1, FP and LR when being used for holding addresses (that is, the special name implies accessing the register as a 64-bit entity).

Note The special register names (IP0, IP1, FP and LR) should be used only in the context in which they are special. It is recommended that disassemblers always use the architectural names for the registers.

The NZCV register is a global condition flag register with the following properties:

- The N, Z, C and V flags are undefined on entry to and return from a public interface.

5.1.2 SIMD and Floating-Point Registers

The ARM 64-bit architecture also has a further thirty-two registers, v0-v31, which can be used by SIMD and Floating-Point operations. The precise name of the register will change indicating the size of the access.

Note Unlike in AArch32, in AArch64 the 128-bit and 64-bit views of a SIMD and Floating-Point register do not overlap multiple registers in a narrower view, so q1, d1 and s1 all refer to the same entry in the register bank.

The first eight registers, v0-v7, are used to pass argument values into a subroutine and to return result values from a function. They may also be used to hold intermediate values within a routine (but, in general, only *between* subroutine calls).

Registers v8-v15 must be preserved by a callee across subroutine calls; the remaining registers (v0-v7, v16-v31) do not need to be preserved (or should be preserved by the caller). Additionally, only the bottom 64-bits of each value stored in v8-v15 need to be preserved¹; it is the responsibility of the caller to preserve larger values.

The FPSR is a status register that holds the cumulative exception bits of the floating-point unit. It contains the fields IDC, IXC, UFC, OFC, DZC, IOC and QC. These fields are not preserved across a public interface and may have any value on entry to a subroutine.

The FPCR is used to control the behavior of the floating-point unit. It is a global register with the following properties.

- 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.
- 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.

5.1.3 Scalable Vector Registers

The ARM 64-bit architecture also defines an optional set of thirty-two scalable vector registers, z0-z31. Each register extends the corresponding SIMD and Floating-Point register so that it can hold the contents of a single Scalable Vector Type (§4.1.3). That is, scalable vector register z0 is an extension of SIMD and Floating-Point register v0.

z0-z7 are used to pass scalable vector arguments to a subroutine, and to return scalable vector results from a function. If a subroutine takes arguments in scalable vector or predicate registers, or if it is a function that returns

¹ This includes double-precision or smaller floating-point values and 64-bit short vector values

results in such registers, it must ensure that the entire contents of z8-z31 are preserved across the call. In other cases it need only preserve the low 64 bits of z8-z15, as described in §5.1.2.

5.1.4 Scalable Predicate Registers

The ARM 64-bit architecture defines an optional set of sixteen scalable predicate registers p0-p15. These registers are available if and only if the scalable vector registers are available (§5.1.3). Each register can store the contents of a Scalable Predicate Type (§4.1.4).

p0-p3 are used to pass scalable predicate arguments to a subroutine and to return scalable predicate results from a function. If a subroutine takes arguments in scalable vector or predicate registers, or if it is a function that returns results in these registers, it must ensure that p4-p15 are preserved across the call. In other cases it need not preserve any scalable predicate register contents.

5.2 Processes, Memory and the Stack

The AAPCS64 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 one or more 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 Memory Addresses

The address space may consist of one or more disjoint regions. No region may span address zero (although one region may start at zero).

The use of tagged addressing is platform-specific. When tagged addressing is disabled, all 64 bits of a pointer are passed to the address translation system. When tagged addressing is enabled, the top eight bits of a pointer are ignored for the purposes of address translation. See also §4.1.5, above.

5.2.2 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 special-purpose register SP. In general, the stack will have both a *base* and a *limit*, although 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.

5.2.2.1 Universal stack constraints

At all times the following basic constraints must hold:

- $\text{Stack-limit} < \text{SP} \leq \text{stack-base}$. The stack pointer must lie within the extent of the stack.
- A process may only access (for reading or writing) the closed interval of the entire stack delimited by $[\text{SP}, \text{stack-base} - 1]$.

Additionally, at any point at which memory is accessed via SP, the hardware requires that

- $\text{SP} \bmod 16 = 0$. The stack must be quad-word aligned.

5.2.2.2 Stack constraints at a public interface

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

- $\text{SP} \bmod 16 = 0$. The stack must be quad-word aligned.

5.2.3 The Frame Pointer

Conforming code shall construct a linked list of stack-frames. Each frame shall link to the frame of its caller by means of a frame record of two 64-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 double-word shall point to the previous frame record and the highest addressed double-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.

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 detect this condition.
- It may elect not to maintain a frame chain and to use the frame pointer register as a general-purpose callee-saved register.

5.3 Subroutine Calls

The A64 instruction set contains primitive subroutine call instructions, BL and BLR, 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. The effect of executing BLR is similar except that the new PC value is read from the specified register.

5.3.1.1 Use of IP0 and IP1 by the linker

The A64 branch instructions are unable to reach every destination in the address space, so it may be necessary for the linker to insert a veneer between a calling routine and a called subroutine. Veneers may also be needed to support dynamic linking. Any veneer inserted must preserve the contents of all registers except IP0, IP1 (r16, r17) and the condition code flags; a conforming program must assume that a veneer that alters IP0 and/or IP1 may be inserted at any branch instruction that is exposed to a relocation that supports long branches.

Note R_AARCH64_CALL26, and R_AARCH64_JUMP26 are the ELF relocation types with this property.

5.4 Parameter Passing

The base standard provides for passing arguments in general-purpose registers (r0-r7), SIMD/floating-point registers (v0-v7), [scalable vector registers \(z0-z7, overlaid on v0-v7\)](#), [scalable predicate registers \(p0-p3\)](#) and on the stack. For subroutines that take a small number of small parameters, only registers are used.

5.4.1 Variadic Subroutines

A Variadic subroutine is a routine that takes a variable number of parameters. The full parameter list is known by the caller, but the callee only knows a minimum number of arguments will be passed and will determine the additional arguments based on the values passed in other arguments. The two classes of arguments are known as *Named* arguments (these form the minimum set) and *Anonymous* arguments (these are the optional additional arguments).

In this standard a non-variadic subroutine can be considered to be identical to a variadic subroutine that takes no optional arguments.

5.4.2 Parameter Passing Rules

Parameter passing is defined as a two-level conceptual model:

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

The mapping from a source language type onto a 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.

For a caller, sufficient stack space to hold stacked argument values is assumed to have been allocated prior to marshaling, but in practice, the amount of stack space required cannot be known until after the argument marshaling has been completed. A callee is permitted to modify any stack space used for receiving parameter values from the caller.

Stage A – Initialization

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

- A.1 The Next General-purpose Register Number (NGRN) is set to zero.
- [A.2](#) The Next SIMD₂ Floating-point [and Scalable Vector](#) Register Number (NSRN) is set to zero.
- [A.3](#) [The Next Scalable Predicate Register Number \(NPRN\) is set to zero.](#)
- A.4 The next stacked argument address (NSAA) is set to the current stack-pointer value (SP).

Stage B – Pre-padding and extension of arguments

For each argument in the list, the first matching rule from the following list is applied. If no rule matches the argument is used unmodified.

- B.1 [If the argument type is a Scalable Vector, a Scalable Predicate, or a Scalable Vector Aggregate, no change is made at this stage.](#)
- B.2 If the argument type is a Composite Type whose size cannot be statically determined by both the caller and the callee, the argument is copied to memory and the argument is replaced by a pointer to the copy. (There are no such types in C/C++ but they exist in other languages or in language extensions).
- B.3 If the argument type is an HFA or an HVA, then the argument is used unmodified.
- B.4 If the argument type is a Composite Type that is larger than 16 bytes, then the argument is copied to memory allocated by the caller and the argument is replaced by a pointer to the copy.
- B.5 If the argument type is a Composite Type, then the size of the argument is rounded up to the nearest multiple of 8 bytes.

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. When an argument is assigned to a register, any unused bits in the register have an unspecified value. When an argument is assigned to a stack slot any unused padding bytes have an unspecified value.

- C.1 If the argument is a Half-, Single-, Double- or Quad- precision Floating-point or Short Vector Type and the NSRN is less than 8, then the argument is allocated to the least significant bits of register $v[\text{NSRN}]$. The NSRN is incremented by one. The argument has now been allocated.
- C.2 If the argument is an HFA or an HVA and there are sufficient unallocated SIMD and Floating-point registers ($\text{NSRN} + \text{number of members} \leq 8$), then the argument is allocated to SIMD and Floating-point Registers (with one register per member of the HFA or HVA). The NSRN is incremented by the number of registers used. The argument has now been allocated.
- C.3 If the argument is an HFA or an HVA then the NSRN is set to 8 and the size of the argument is rounded up to the nearest multiple of 8 bytes.
- C.4 If the argument is an HFA, an HVA, a Quad-precision Floating-point or Short Vector Type then the NSAA is rounded up to the larger of 8 or the Natural Alignment of the argument's type.
- C.5 If the argument is a Half- or Single- precision Floating-Point type, then the size of the argument is set to 8 bytes. The effect is as if the argument had been copied to the least significant bits of a 64-bit register and the remaining bits filled with unspecified values.
- C.6 If the argument is an HFA, an HVA, a Half-, Single-, Double- or Quad- precision Floating-point or Short Vector Type, then the argument is copied to memory at the adjusted NSAA. The NSAA is incremented by the size of the argument. The argument has now been allocated.
- C.7 If the argument is a Scalable Vector Type, if the argument is named, and if the NSRN is less than 8, then the argument is allocated to $z[\text{NSRN}]$. The NSRN is incremented by one. The argument has now been allocated.
- C.8 If the argument is a Scalable Predicate Type, the argument is named, and the NPRN is less than 4, then the argument is allocated to $p[\text{NPRN}]$. The NPRN is incremented by one. The argument has now been allocated.
- C.9 If the argument is a Scalable Vector Aggregate that consists of NV Scalable Vector Types and NP Scalable Predicate Types, if the argument is named, if $\text{NSRN} + NV \leq 8$, and if $\text{NPRN} + NP \leq 4$, then the Scalable Vector Types are allocated in order to $z[\text{NSRN}] \dots z[\text{NSRN} + NV - 1]$ inclusive and the Scalable Predicate Types are allocated in order to $p[\text{NPRN}] \dots p[\text{NPRN} + NP - 1]$ inclusive. The NSRN is incremented by NV and the NPRN is incremented by NP . The argument has now been allocated.
- C.10 If the argument is a Scalable Vector Type, a Scalable Predicate Type or a Scalable Vector Aggregate that has not been allocated by the rules above, then the argument is copied to memory allocated by the caller and the argument is replaced by a pointer to the copy (as for B.4 above). The argument is then allocated according to the rules below.
- C.11 If the argument is an Integral or Pointer Type, the size of the argument is less than or equal to 8 bytes and the NGRN is less than 8, the argument is copied to the least significant bits in $x[\text{NGRN}]$. The NGRN is incremented by one. The argument has now been allocated.
- C.12 If the argument has an alignment of 16 then the NGRN is rounded up to the next even number.
- C.13 If the argument is an Integral Type, the size of the argument is equal to 16 and the NGRN is less than 7, the argument is copied to $x[\text{NGRN}]$ and $x[\text{NGRN} + 1]$. $x[\text{NGRN}]$ shall contain the lower addressed double-word of the memory representation of the argument. The NGRN is incremented by two. The argument has now been allocated.
- C.14 If the argument is a Composite Type and the size in double-words of the argument is not more than 8 minus NGRN, then the argument is copied into consecutive general-purpose registers, starting at $x[\text{NGRN}]$. The argument is passed as though it had been loaded into the registers from a double-word-aligned address with an appropriate sequence of LDR instructions loading consecutive registers from memory (the contents of any unused parts of the registers are unspecified by this standard). The NGRN is incremented by the number of registers used. The argument has now been allocated.
- C.15 The NGRN is set to 8.
- C.16 The NSAA is rounded up to the larger of 8 or the Natural Alignment of the argument's type.

- C.17 If the argument is a composite type, then the argument is copied to memory at the adjusted NSAA. The NSAA is incremented by the size of the argument. The argument has now been allocated.
- C.18 If the size of the argument is less than 8 bytes then the size of the argument is set to 8 bytes. The effect is as if the argument was copied to the least significant bits of a 64-bit register and the remaining bits filled with unspecified values.
- C.19 The argument is copied to memory at the adjusted NSAA. The NSAA is incremented by the size of the argument. The argument has now been allocated.

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 routine is variadic.

Several further observations can also be made:

- The address of the first stacked argument is defined to be the initial value of SP. Therefore, the total amount of stack space needed by the caller for argument passing cannot be determined until all the arguments in the list have been processed.
- Floating-point and short vector types are passed in SIMD and Floating-point registers or on the stack; never in general-purpose registers (except when they form part of a small structure that is neither an HFA nor an HVA).
- Unlike in the 32-bit AAPCS, named integral values must be narrowed by the callee rather than the caller.
- Unlike in the 32-bit AAPCS, half-precision floating-point values can be passed directly (and HFAs of half-precision floats are also permitted).
- Any part of a register or a stack slot that is not used for an argument (padding bits) has unspecified content at the callee entry point.
- The rules here do not require narrow arguments to subroutines to be widened. However, a language may require widening in some or all circumstances (for example, in C, unprototyped and variadic functions require single-precision values to be converted to double-precision and char and short values to be converted to int).
- HFAs and HVAs are special cases of a composite type. If they are passed as parameters in registers, then each uniquely addressable element goes in its own register. However, if they are not allocated to registers then they are always passed on the stack (never in general-purpose registers) and they are laid out in exactly the same way as any other composite.
- Both before and after the layout of each argument, then NSAA will have a minimum alignment of 8.

5.5 Result Return

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

- If the type, *T*, of the result of a function is such that


```
void func(T arg)
```

 would require that *arg* be passed as a value in a register (or set of registers) according to the rules in §5.4 *Parameter Passing*, then the result is returned in the same registers as would be used for such an argument.
- Otherwise, the caller shall reserve a block of memory of sufficient size and alignment to hold the result. The address of the memory block shall be passed as an additional argument to the function in *x8*. The callee may modify the result memory block at any point during the execution of the subroutine (there is no requirement for the callee to preserve the value stored in *x8*).

5.6 Interworking

Interworking between the 32-bit AAPCS and the AAPCS64 is not supported within a single process. (In AArch64, all inter-operation between 32-bit and 64-bit machine states takes place across a change of exception level.)

6 THE STANDARD VARIANTS

6.1 Half-precision Format Compatibility

The set of values that can be represented in 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. However, most code will make no use of either format and will therefore be compatible with both variants.

6.2 `sizeof(long)`, `sizeof(wchar_t)`

See section 7.1.1.

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	
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 or signed double-word	see text
unsigned long	unsigned word or unsigned double-word	see text
[signed] long long	signed double-word	C99 Only
unsigned long long	unsigned double-word	C99 Only
__int128	signed quad-word	ARM extension (used for LDXP/STXP)
__uint128	unsigned quad-word	ARM extension (used for LDXP/STXP)
__fp16	half precision (IEEE754-2008 format or Alternative Format)	ARM extension.
float	single precision (IEEE 754)	
double	double precision (IEEE 754)	
long double	quad precision (IEEE 754-2008)	
float _Imaginary	single precision (IEEE 754)	C99 Only
double _Imaginary	double precision (IEEE 754)	C99 Only
long double _Imaginary	quad precision (IEEE 754-2008)	C99 Only

C/C++ Type	Machine Type	Notes
<code>float _Complex</code>	2 single precision (IEEE 754)	C99 Only. Layout is <pre>struct {float re; float im;};</pre>
<code>double _Complex</code>	2 double precision (IEEE 754)	C99 Only. Layout is <pre>struct {double re; double im;};</pre>
<code>long double _Complex</code>	2 quad precision (IEEE 754-2008)	C99 Only. Layout is <pre>struct {long double re; long double im;};</pre>
<code>_Bool/bool</code>	unsigned byte	C99/C++ Only. False has value 0 and True has value 1.
<code>wchar_t</code>	unsigned short or unsigned int	built-in in C++, typedef in C, type is platform-specific; see text

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

In the SVR4-like variant of this standard the type of `wchar_t` is unsigned int and the machine types of `long` and `unsigned long` are double-word and unsigned double-word respectively (LP64 model).

In the Windows-like variant of this standard the type of `wchar_t` is unsigned short and the machine types of `long` and `unsigned long` are word and unsigned word respectively (LLP64 model).

A platform ABI may specify a different combination of primitive variants but we discourage this.

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
<code>T *</code>	data pointer	any data type T
<code>T (*F) ()</code>	code pointer	any function type F
<code>T&</code>	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 a word (`int` or `unsigned int`) for all enumeration types. The container type shall be `unsigned int` unless that is unable to represent all the declared values in the enumerated type.

If the set of values in an enumerated type cannot be represented using either `int` or `unsigned int` as a container type, and the language permits extended enumeration sets, then a `long long` or `unsigned long long` container may be used. If all values in the enumeration are in the range of `unsigned long long`, then the container type is `unsigned long long`, otherwise the container type is `long long`.

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 unsigned container 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 AAPCS64 conforming object must use the definitions shown in *Table 5, Additional data types*.

Typedef	Base type	Notes
<code>size_t</code>	unsigned long or unsigned long long	For consistent C++ mangling of <code>::operator new()</code> . On systems with a 32-bit long, the Base type shall be unsigned long long.
<code>va_list</code>	<pre>struct __va_list { void *__stack; void *__gr_top; void *__vr_top; int __gr_offs; int __vr_offs; }</pre>	A <code>va_list</code> may address any object in a parameter list. In C++, <code>__va_list</code> is in namespace <code>std</code> . See <i>Appendix D Variable Argument Lists</i> .

Table 5, 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.

The memory system underlying the processor may have a restricted bus width to some or all of the memory. The only guarantee applying to volatile types in these circumstances is 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, a compiler shall use an instruction that will access the type exactly.

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 *CPPABI64*.

7.1.7 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 §7.1.7.3, *Over-sized bit-fields*).

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.

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:

$$CA(F) = \&(\text{container}(F));$$

This address will always be at the Natural Alignment of the container type, that is:

$$CA(F) \% \text{sizeof}(\text{container}(F)) == 0.$$

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:

$$\text{TRUNCATE}(X, Y) = Y * \lfloor X/Y \rfloor$$

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:

$$NCBA(BA, A) = \text{TRUNCATE}(BA + A - 1, A)$$

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 AAPCS64 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 $\text{TRUNCATE}(BA(F), C) / 8$ into a 64-bit register R
- Set $Q = \text{MAX}(64, C)$
- Little-endian, set $R = (R \ll ((Q - W) - (BA \text{ MOD } C))) \gg (Q - W)$.
- Big-endian, set $R = (R \ll (Q - C + (BA \text{ MOD } C))) \gg (Q - W)$.

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 and the rules for allocation are given in [GC++ABI]. 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 $C' \geq C$ and $A' \geq A$.
- If $C' > \text{UCB}(CBA, C', A')$ setting $CBA = \text{NCBA}(CBA, A')$. 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 $CBA = CBA + W - C$.

Each segment of 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++, an implicit `this` parameter is passed as an extra argument that immediately precedes the first user argument. Other rules for marshaling C++ arguments are described in *CPPABI64*.
- For unprototyped (i.e. pre-ANSI or K&R C) and variadic functions, in addition to the normal conversions and promotions, arguments of type `__fp16` are converted to type `double`.
- The rules for passing Scalable Vector Types, Scalable Predicate Types and Scalable Vector Aggregates depend on whether the arguments are named. It is an error to pass such values to an unprototyped function.

The argument list is then processed according to the standard rules for procedure calls (see §5.4, *Parameter Passing*) or the appropriate variant.

APPENDIX A C AND C++ SUPPORT FOR SIMD EXTENSIONS

The AArch64 architecture supports a number of short-vector operations. To facilitate accessing these types from C and C++ a number of extended types need to be added to the language.

Following the conventions used for adding types to C99 a number of additional types (internal types) are defined unconditionally. To facilitate use in applications a header file is also defined (`arm_neon.h`) that maps these internal types onto more user-friendly names. These types are listed in *Table 6: Short vector extended types*.

The header file `arm_neon.h` also defines a number of intrinsic functions that can be used with the types defined below. The list of intrinsic functions and their specification is beyond the scope of this document.

Internal type	<code>arm_neon.h</code> type	Base Type	Elements
<code>__Int8x8_t</code>	<code>int8x8_t</code>	signed byte	8
<code>__Int16x4_t</code>	<code>int16x4_t</code>	signed half-word	4
<code>__Int32x2_t</code>	<code>int32x2_t</code>	signed word	2
<code>__UInt8x8_t</code>	<code>uint8x8_t</code>	unsigned byte	8
<code>__UInt16x4_t</code>	<code>uint16x4_t</code>	unsigned half-word	4
<code>__UInt32x2_t</code>	<code>uint32x2_t</code>	unsigned word	2
<code>__Float16x4_t</code>	<code>float16x4_t</code>	half-precision float	4
<code>__Float32x2_t</code>	<code>float32x2_t</code>	single-precision float	2
<code>__Poly8x8_t</code>	<code>poly8x8_t</code>	unsigned byte	8
<code>__Poly16x4_t</code>	<code>poly16x4_t</code>	unsigned half-word	4
<code>__Int8x16_t</code>	<code>int8x16_t</code>	signed byte	16
<code>__Int16x8_t</code>	<code>int16x8_t</code>	signed half-word	8
<code>__Int32x4_t</code>	<code>int32x4_t</code>	signed word	4
<code>__Int64x2_t</code>	<code>int64x2_t</code>	signed double-word	2
<code>__UInt8x16_t</code>	<code>uint8x16_t</code>	unsigned byte	16
<code>__UInt16x8_t</code>	<code>uint16x8_t</code>	unsigned half-word	8
<code>__UInt32x4_t</code>	<code>uint32x4_t</code>	unsigned word	4
<code>__UInt64x2_t</code>	<code>uint64x2_t</code>	unsigned double-word	2
<code>__Float16x8_t</code>	<code>float16x8_t</code>	half-precision float	8
<code>__Float32x4_t</code>	<code>float32x4_t</code>	single-precision float	4
<code>__Float64x2_t</code>	<code>float64x2_t</code>	double-precision float	2
<code>__Poly8x16_t</code>	<code>poly8x16_t</code>	unsigned byte	16
<code>__Poly16x8_t</code>	<code>poly16x8_t</code>	unsigned half-word	8

Internal type	arm_neon.h type	Base Type	Elements
<code>__Poly64x2_t</code>	<code>poly64x2_t</code>	unsigned double-word	2

Table 6: Short vector extended types

APPENDIX B C AND C++ SUPPORT FOR SCALABLE VECTORS

As an optional extension, the AArch64 architecture supports a number of scalable-vector operations. To facilitate accessing these operations from C and C++ a number of extended types need to be added to the language.

Following the conventions used for adding types to C99, and the conventions established in *Appendix A: C and C++ Support for SIMD Extensions*, a number of additional types (internal types) are defined unconditionally. To facilitate use in applications a header file `arm_sve.h` is also defined that maps these internal types onto more user-friendly names. These types are listed in *Table 7: Scalable Vector and Predicate Types*. `__SVBool_t` is a Scalable Predicate Type; the rest are Scalable Vector Types.

For each scalable vector type `svBASE_t`, `arm_sve.h` also defines tuples of 2, 3 and 4 Scalable Vector Types called `svBASEx2_t`, `svBASEx3_t` and `svBASEx4_t` respectively. The exact definition of these types depends on the implementation, but each `svBASExN_t` must be a Scalable Vector Aggregate that contains *N* members of type `svBASE_t`.

The header file `arm_sve.h` also defines a number of intrinsic functions that can be used with these vector and predicate types. The list of intrinsic functions and their specification is beyond the scope of this document.

Internal type	arm_sve.h type	Base Type	Elements
<code>__SVInt8_t</code>	<code>svint8_t</code>	signed byte	<code>VGx8</code>
<code>__SVUInt8_t</code>	<code>svuint8_t</code>	unsigned byte	<code>VGx8</code>
<code>__SVInt16_t</code>	<code>svint16_t</code>	signed half-word	<code>VGx4</code>
<code>__SVUInt16_t</code>	<code>svuint16_t</code>	unsigned half-word	<code>VGx4</code>
<code>__SVFloat16_t</code>	<code>svfloat16_t</code>	half-precision float	<code>VGx4</code>
<code>__SVInt32_t</code>	<code>svint32_t</code>	signed word	<code>VGx2</code>
<code>__SVUInt32_t</code>	<code>svuint32_t</code>	unsigned word	<code>VGx2</code>
<code>__SVFloat32_t</code>	<code>svfloat32_t</code>	single-precision float	<code>VGx2</code>
<code>__SVInt64_t</code>	<code>svint64_t</code>	signed double-word	<code>VG</code>
<code>__SVUInt64_t</code>	<code>svuint64_t</code>	unsigned double-word	<code>VG</code>
<code>__SVFloat64_t</code>	<code>svfloat64_t</code>	double-precision float	<code>VG</code>
<code>__SVBool_t</code>	<code>svbool_t</code>	single bit (fully packed into <code>VG</code> bytes)	<code>VGx8</code>

Table 7: Scalable Vector and Predicate Types

APPENDIX C C++ MANGLING

For C++ mangling purposes the user-friendly names [described in Appendix A: C and C++ Support for SIMD Extensions](#) and [Appendix B: C and C++ Support for Scalable Vectors](#) are treated as though the equivalent internal name was specified. Thus the function

```
void f(int8x8_t)
```

is mangled as

```
_Z1fu10__Int8x8_t
```

[The SVE tuple types are mangled using their `arm_sve.h` names \(`svBASExN_t`\).](#)

APPENDIX D VARIABLE ARGUMENT LISTS

Languages such as C and C++ permit routines that take a variable number of arguments (that is, the number of parameters is controlled by the caller rather than the callee). Furthermore, they may then pass some or even all of these parameters as a block to further subroutines to process the list. If a routine shares any of its optional arguments with other routines, then a parameter control block needs to be created as specified in §7.1.4 *Additional Types*. The remainder of this appendix is informative.

D.1 Register Save Areas

The prologue of a function which accepts a variable argument list and which invokes the `va_start` macro is expected to save the incoming argument registers to two register save areas within its own stack frame: one area to hold the 64-bit general registers `xn-x7`, the other to hold the 128-bit FP/SIMD registers `vn-v7`. Only parameter registers beyond those which hold the named parameters need be saved, and if a function is known never to accept parameters in registers of that class, then that register save area may be omitted altogether. In each area the registers are saved in ascending order. The memory format of FP/SIMD registers save area must be as if each register were saved using the integer `str` instruction for the entire (ie Q) register.

D.2 The `va_list` type

The `va_list` type may refer to any parameter in a parameter list, which depending on its type and position in the argument list may be in one of three memory locations: the current function's general register argument save area, its FP/SIMD register argument save area, or the calling function's outgoing stack argument area.

```
typedef struct __va_list {
    void *__stack;    // next stack param
    void *__gr_top;  // end of GP arg reg save area
    void *__vr_top;  // end of FP/SIMD arg reg save area
    int  __gr_offs;  // offset from __gr_top to next GP register arg
    int  __vr_offs;  // offset from __vr_top to next FP/SIMD register arg
} va_list;
```

D.3 The `va_start()` macro

The `va_start` macro shall initialize the fields of its `va_list` argument as follows, where `named_gr` represents the number of general registers known to hold named incoming arguments and `named_vr` the number of FP/SIMD registers known to hold named incoming arguments.

- `__stack`: set to the address following the last (highest addressed) named incoming argument on the stack, rounded upwards to a multiple of 8 bytes, or if there are no named arguments on the stack, then the value of the stack pointer when the function was entered.
- `__gr_top`: set to the address of the byte immediately following the general register argument save area, the end of the save area being aligned to a 16-byte boundary.
- `__vr_top`: set to the address of the byte immediately following the FP/SIMD register argument save area, the end of the save area being aligned to a 16-byte boundary.
- `__gr_offs`: set to $0 - ((8 - \text{named_gr}) * 8)$.
- `__vr_offs`: set to $0 - ((8 - \text{named_vr}) * 16)$.

If it is known that a `va_list` structure is never used to access arguments that could be passed in the FP/SIMD argument registers, then no FP/SIMD argument registers need to be saved, and the `__vr_top` and `__vr_offs` fields initialised to zero. Furthermore, if in this case the general register argument save area is located immediately below the value of the stack pointer on entry, then the `__stack` field may set to the address of the anonymous argument in the general register argument save area and the `__gr_top` and `__gr_offs` fields also

set to zero, permitting a simplified implementation of `va_arg` which simply advances the `__stack` pointer through the argument save area and into the incoming stacked arguments. This simplification may not be used in the reverse case where anonymous arguments are known to be in FP/SIMD registers but not in general registers.

Although this standard does not mandate a particular stack frame organisation beyond what is required to meet the stack constraints described in §5.2.2 *The Stack*, Figure 3, *Example stack frame layout* illustrates one possible stack layout for a variadic routine which invokes the `va_start` macro.

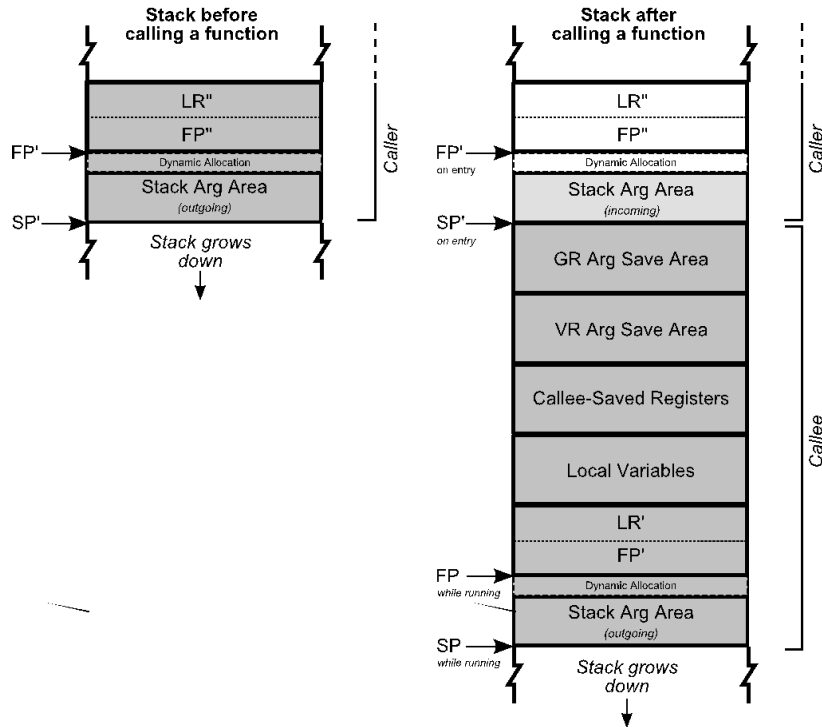


Figure 3, Example stack frame layout

Focussing on just the top of callee’s stack frame, Figure 4, *The va_list* illustrates graphically how the `__va_list` structure might be initialised by `va_start` to identify the three potential locations of the next anonymous argument.

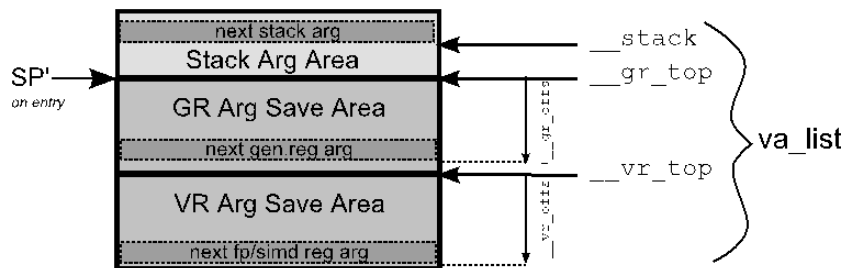


Figure 4, The va_list

D.4 The `va_arg()` macro

The algorithm to implement the generic `va_arg(ap, type)` macro is then most easily described using a C-like “pseudocode”, as follows:

```

type va_arg(va_list ap, type)
{
    int nreg, offs;

```

```

    if (type passed in general registers) {
        offs = ap.__gr_offs;
        if (offs >= 0)
            goto on_stack;          // reg save area empty
        if (alignof(type) > 8)
            offs = (offs + 15) & -16; // round up
        nreg = (sizeof(type) + 7) / 8;
        ap.__gr_offs = offs + (nreg * 8);
        if (ap.__gr_offs > 0)
            goto on_stack;          // overflowed reg save area
#ifdef BIG_ENDIAN
        if (classof(type) != "aggregate" && sizeof(type) < 8)
            offs += 8 - sizeof(type);
#endif
        return *(type *) (ap.__gr_top + offs);
    } else if (type is an HFA or an HVA) {
        type ha; // treat as "struct {ftype field[n];}"
        offs = ap.__vr_offs;
        if (offs >= 0)
            goto on_stack;          // reg save area empty
        nreg = sizeof(type) / sizeof(ftype);
        ap.__vr_offs = offs + (nreg * 16);
        if (ap.__vr_offs > 0)
            goto on_stack;          // overflowed reg save area
#ifdef BIG_ENDIAN
        if (sizeof(ftype) < 16)
            offs += 16 - sizeof(ftype);
#endif
        for (i = 0; i < nreg; i++, offs += 16)
            ha.field[i] = *((ftype *) (ap.__vr_top + offs));
        return ha;
    } else if (type passed in fp/simd registers) {
        offs = ap.__vr_offs;
        if (offs >= 0)
            goto on_stack;          // reg save area empty
        nreg = (sizeof(type) + 15) / 16;
        ap.__vr_offs = offs + (nreg * 16);
        if (ap.__vr_offs > 0)
            goto on_stack;          // overflowed reg save area
#ifdef BIG_ENDIAN
        if (classof(type) != "aggregate" && sizeof(type) < 16)
            offs += 16 - sizeof(type);
#endif
        return *(type *) (ap.__vr_top + offs);
    }
}

on_stack:
    intptr_t arg = ap.__stack;
    if (alignof(type) > 8)
        arg = (arg + 15) & -16;
    ap.__stack = (void *) ((arg + sizeof(type) + 7) & -8);
#ifdef BIG_ENDIAN
    if (classof(type) != "aggregate" && sizeof(type) < 8)
        arg += 8 - sizeof(type);
#endif
    return *(type *) arg;
}

```

If *type* is a Scalable Vector Type, a Scalable Predicate Type, or a Scalable Vector Aggregate, the pseudo-code above should be used to obtain a *type* *, which should then be dereferenced to get the required value.

It is expected that the implementation of the `va_arg` macro will be specialized by the compiler for the type, size and alignment of the type. By way of example the following sample code illustrates one possible expansion of `va_arg(ap, int)`, where register `x0` holds a pointer to `va_list ap`, and the argument is returned in register `w1`. Further optimizations are possible.

```

        ldr    w1, [x0, #__gr_offs]    // get register offset
        tbz   w1, #31, stack         // reg save area empty?
        adds  w2, w1, #8              // advance to next register offset
        str   w2, [x0, #__gr_offs]    // save next register offset
        bgt   on_stack               // just overflowed reg save area?
        ldr   x2, [x0, #__gr_top]     // get top of save area
#ifdef BIG_ENDIAN
        add   w1, w1, #4              // adjust offset to low 32 bits
#endif
        ldr   w1, [x2, w1, sxtw]      // load arg
        b     done
on_stack:
        ldr   x2, [x0, #__stack]      // get stack slot pointer
#ifdef BIG_ENDIAN
        ldr   w1, [x2, #4]            // load low 32 bits
        add   x2, #8                  // advance to next stack slot
#else
        ldr   w1, [x2], #8            // load low 32 bits and advance stack slot
#endif
        str   x2, [x0, #__stack]      // save next stack slot pointer
done:

```