Morello Supplement to the Procedure Call Standard for the Arm 64-bit Architecture (AArch64)

Release October 2020

Arm Ltd

Oct 1, 2020

102205_0001_00_en

CONTENTS

Document ID: 102205

Date of Issue: 1st October 2020

CHAPTER

ONE

CHANGE CONTROL

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.

This document is a draft and all content is at the **Alpha** quality level.

1.2 Change history

Issue	Date	Change
00alpha	1st October 2020	Alpha release

CHAPTER

TWO

LICENSE

2.1 CC-BY-SA-4.0 and Patent Grant

This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Grant of Patent License. Subject to the terms and conditions of this license (both the Public License and this Patent License), each Licensor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Licensed Material, where such license applies only to those patent claims licensable by such Licensor that are necessarily infringed by their contribution(s) alone or by combination of their contribution(s) with the Licensed Material to which such contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counter-claim in a lawsuit) alleging that the Licensed Material or a contribution incorporated within the Licensed Material constitutes direct or contributory patent infringement, then any licenses granted to You under this license for that Licensed Material shall terminate as of the date such litigation is filed.

2.1.1 About the license

As identified more fully in the *CC-BY-SA-4.0 and Patent Grant* (page 3) section, this project is licensed under CC-BY-SA-4.0 along with an additional patent license. The language in the additional patent license is largely identical to that in Apache-2.0 (specifically, Section 3 of Apache-2.0 as reflected at https://www.apache.org/licenses/LICENSE-2.0) with two exceptions.

First, several changes were made related to the defined terms so as to reflect the fact that such defined terms need to align with the terminology in CC-BY-SA-4.0 rather than Apache-2.0 (e.g., changing "Work" to "Licensed Material").

Second, the defensive termination clause was changed such that the scope of defensive termination applies to "any licenses granted to You" (rather than "any patent licenses granted to You"). This change is intended to help maintain a healthy ecosystem by providing additional protection to the community against patent litigation claims.

2.1.2 Contributions

Contributions to this project are licensed under an inbound=outbound model such that any such contributions are licensed by the contributor under the same terms as those in the *CC-BY-SA-4.0 and Patent Grant* (page 3) section.

2.1.3 Trademark Notice

The text of and illustrations in this document are licensed by Arm under a Creative Commons Attribution–Share Alike 4.0 International license ("CC-BY-SA-4.0"), with an additional clause on patents.

2.1.4 Copyright Notice

CHAPTER

THREE

PREFACE

3.1 Abstract

This document describes the Morello additions to the Procedure Call Standard use by the Application Binary Interface (ABI) for the Arm 64-bit architecture.

3.2 Keywords

Procedure call, function call, calling conventions, data layout

3.3 References

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

Ref	URL or other reference	Title
AAPCS64-morello (page 1)	This document	Morello Supplement to the Procedure Call Standard for the Arm 64-bit Architecture
AAPCS64	IHI 005D	Procedure Call Standard for the Arm 64-bit Architecture

3.4 Terms and Abbreviations

The ABI for the Arm 64-bit Architecture uses the following terms and abbreviations in addition to the terms and abbreviations used in the AAPCS64 document.

AAPCS64-cap The pure capability Procedure Call Standard for the Arm 64-bit Architecture (AArch64)

C64 Execution state where PSTATE.C64 is set.

A64 Execution state where PSTATE.C64 is cleared.

Capability The capability data type is an unforgeable token of authority which provides a foundation for fine grained memory protection and strong compartmentalisation.

Deriving a capability A capability value CV2 is said to be derived from a capability value CV1 when CV2 is a copy of CV1 with optionally removed permissions and/or optionally narrowed bounds (base increased or limit reduced).

More specific terminology is defined when it is first used.

3.5 Acknowledgements

CHAPTER

FOUR

SCOPE

This document extends the AAPCS64 calling convention described in the AAPCS64 document in order to add support for capabilities, and adds an additional Procedure Calling Standard (AAPCS64-cap). AAPCS64-cap is identical to AAPCS64, except for the differences documented here.

CHAPTER

FIVE

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.

The AAPCS64-cap is a Procedure Call standard for the Arm 64-bit Architecture designed to implement a software environment where all memory accesses are performed using capabilities.

5.1 Design Goals

AAPCS64 and AAPCS64-cap have the same design goals as described in the AAPCS64 document.

An additional design goal for the capability-aware AAPCS64 is to support interworking with legacy AAPCS64 code, following only the Procedure Calling Standard described in the AAPCS64 document.

5.2 Conformance

The AAPCS64 and AAPCS64-cap have the same conformance rules as defined in AAPCS64 , with the additional requirement that:

- At each call where the control transfer instruction is subject to a BL-type relocation at static link time, rules on the use of r16 and r17 (CIP0, CIP1, IP0, IP1) are observed (*Use of CIP0 and CIP1 by the linker* (page 12)).

CHAPTER

SIX

DATA TYPES AND ALIGNMENT

6.1 Fundamental Data Types

Table 6.1, Byte size and byte alignment of Morello-specific fundamental data types shows the additional fundamental data types (Machine Types) of the machine that are available in AAPCS64 and AAPCS64-cap, in addition to the fundamental data types shown in the AAPCS64 document.

Table 6.1: Byte size and byte alignment of Morello-specific fundamental data types

Type Class	Machine Type	Byte size	Natural Alignment (bytes)	Note
Capability	Data capability	16	16	See *Capabilities* (page 8)
	Code capability	16	16	

6.2 Capabilities

Capabilities are 129-bit types which encode a 64 bit value and extra information such as the bounds of the allocation the value is addressing and access permissions. Each capability has a single-bit tag associated with it that is tracked by the hardware, used to guarantee that the capability is unforgeable. The size of the capability in memory is 16 bytes, with the tag bit being stored separately at the corresponding capability tag location. A capability must be stored in memory at a 16-byte aligned address.

A NULL capability is represented by a capability with all bits zero, and the tag bit cleared.

A comparison between two capabilities will be performed by comparing the value fields of the two capabilities.

CHAPTER

SEVEN

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.

7.1 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.
- A SIMD and Floating-Point register bank.

7.1.1 General-purpose Registers

There are thirty-one, 128 bit (with one additional tag bit), general-purpose (capability) registers visible to the A64 and C64 states; these are labelled 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.
- In a capability context the registers are specified by using c0-c30.

Additionally, a stack-pointer register, SP in a 64-bit context or CSP in a capability context, 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 when the register has a fixed role in this procedure call standard, upper case is used.

- Table 7.1, General purpose registers and AAPCS64-cap usage summarize the uses of the general-purpose registers for AAPCS64-cap.
- Table 7.2, General purpose registers and AAPCS64 usage summarize the uses of the general-purpose registers for AAPCS64.

Table 7.1: General purpose registers and AAPCS64-cap usage

Register	Special	Role in AAPCS64-cap
r31	CSP	The Capability Stack Pointer.
r30	CLR	The Capability Link Register.
r29	CFP	The Capability Frame Pointer
r19-r28		Registers r19-r28 (c19-c28) are callee-saved
r18		The Platform Register, if needed; otherwise a temporary register. See notes.
r17	CIP1	The second intra-procedure-call temporary register (can be used by call veneers and PLT code)
r16	CIP0	The first intra-procedure-call scratch register (can be used by call veneers and PLT code)
r9-r15		Temporary registers
r8		The capability indirect result location register
r0-r7		Parameter/result registers

Table 7.2: General purpose registers and AAPCS64 usage

Register	Special	Role in AAPCS64
r31	SP	The Stack Pointer.
r30	LR	The Link Register.
r29	FP	The Frame Pointer
r19-r28		The lower 64 bits of the registers (x19-x28) is callee-saved
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)
r16	IP0	The first intra-procedure-call scratch register (can be used by call veneers and PLT code)
r9-r15		Temporary registers
r8		The indirect result location register
r0-r7		Parameter/result registers

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/CIP0) and r17 (IP1/CIP1) may be used by a linker as a scratch register between a routine and any subroutine it calls (for details, see aapcs64-section5-3-1). 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.

In AAPCS64-cap a subroutine invocation must preserve the contents of the registers r19-r29 and CSP. All 128 bits and the tag bit of each value stored in r19-r29 must be preserved.

In AAPCS64 a subroutine invocation must preserve the contents of the lower 64 bits of registers r19-r29 and SP. There is no requirement to preserve the tag bit.

Note: In AAPCS64 c19-c30 and CSP are not callee-saved, although x19-x30 and SP are callee-saved. It is therefore the responsibility of the caller to save any of the c19-c30 and CSP registers before any call, if these registers are used by the caller.

In all variants of the procedure call standard, registers r16, r17, r29 and r30 have special roles. In these roles they are labelled 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/CIP0, IP1/CIP1, FP/CFP and LR/CLR) 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.

7.2 Processes, Memory and the Stack

7.2.1 The Stack in AAPCS64-cap

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

The stack implementation is full-descending, with the current extent of the stack held in the special-purpose register CSP. The stack will have both a base and a limit, and an application can get these values by observing the base and limit of CSP.

The size of the stack is fixed in AAPCS64-cap. This size is encoded in CSP.

AAPCS64-cap has the same rules and constraints for maintenance of the stack as AAPCS64, with the following additional constraints:

- CSP must be a valid capability with the tag set, zero type (unsealed), and the bounds set to stack-base and stack-limit. In this case, stack-base and stack-limit are defined as being the bounds of the CSP capability. The values of stack-base and stack-limit are constrained such that they can form the upper and lower bound of a representable capability.
- CSP must have the Load, Store, LoadCap, StoreCap and MutableLoad permission bits set.
- CSP must have enough permission to be used to store capabilities derived from CSP. This means that CSP should either have at least one of the Global and StoreLocalCap permissions.

7.2.2 The Frame Pointer in AAPCS64-cap

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 capability values on the stack. The frame record for the inner-most frame (belonging to the most recent routine invocation) shall be pointed to by the Capability Frame Pointer register (CFP). The lowest addressed capability shall point to the previous frame record and the highest addressed capability shall contain the value passed in CLR on entry to the current function. The end of the frame record chain is indicated by the NULL capability 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, with the same choices as for AAPCS64.

7.3 Subroutine Calls

The A64 and C64 states contain 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 or CLR) and the destination address into the program counter. The effect of executing BLR is similar except that the new PC value is constructed from the specified register.

7.3.1 Use of CIP0 and CIP1 by the linker

The A64 and C64 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 and interworking between A64 and C64. Any veneer inserted must preserve the contents of all registers except CIP0, CIP1 (r16, r17) and the condition code flags; a conforming program must assume that a veneer that alters CIP0 and/or CIP1 may be inserted at any branch instruction that is exposed to a relocation that supports long branches.

7.4 Parameter Passing

The base standard provides for passing arguments in general-purpose registers (r0-r7), SIMD/floating-point registers (v0-v7) and on the stack. For subroutines that take a small number of small parameters, only registers are used.

7.4.1 Parameter Passing Rules

The parameter passing rules are modified from those shown in the AAPCS64 document to take into account capabilities. The marshalling of machine types is the same for AAPCS64 and AAPCS64-cap.

The differences in language bindings used for AAPCS64 and AAPCS64-cap are described in *Types Varying by Data Model and Procedure Calling Standard* (page 16).

Stage A – Initialization

A.1	The Next General-purpose Register Number (NGRN) is set to zero.
A.2	The Next SIMD and Floating-point Register Number (NSRN) is set to zero.
A.3	The next stacked argument address (NSAA) is set to the current stack-pointer value (SP).

Stage B – Pre-padding and extension of arguments

B.1	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 in AAPCS64 or a capability to the copy in AAPCS64-cap. (There are no such types in C/C++ but they exist in other languages or in language extensions).
B.2	If the argument type is an HFA or an HVA, then the argument is used unmodified.
B.3	If the argument type is a Composite Type which does not contain capabilities 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 in AAPCS64 or a capability to the copy in AAPCS64-cap.
B.4	If the argument type is a Composite Type then the size of the argument is rounded up to the nearest multiple of 8 bytes.
B.5	If the argument is a Composite Type containing Capabilities and the size is larger than 32 bytes or there are addressable members which are not Capabilities that overlap bytes 8-15 or 24-31 of the argument (if such bytes exist) then the argument is copied to memory allocated by the caller and the argument is replaced by a pointer to the copy in AAPCS64 or a capability to a copy in AAPCS64-cap.
B.6	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 8-byte alignment if its natural alignment is <= 8 and 16-byte alignment if its natural alignment is >= 16. The alignment of the copy is used for applying marshalling rules.

Stage C – Assignment of arguments to registers and stack

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[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 (NSRN + number of members <= 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 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 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[NGRN]. The NGRN is incremented by one. The argument has now been allocated.
C.8	If the argument is a Capability Type or a Composite Type containing Capabilities, and the size of the argument in quad words is less than 8 minus NGRN, the argument is passed as though it had been loaded into capability registers starting from a 16-byte aligned address with an appropriate sequence of capability loading instructions loading consecutive capability values from memory, starting from c[NGRN]. The NGRN is incremented by the number of capability registers used to hold the argument. The argument has now been allocated.
C.9	If the argument is not a Capability Type and is not a Composite Type containing Capability Types and has an alignment of 16 then the NGRN is rounded up to the next even number.
C.10	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[NGRN] and x[NGRN+1]. x[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.11	If the argument is a Composite Type which does not contain Capability Types 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[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.12	The NGRN is set to 8.
C.13	The NSAA is rounded up to the larger of 8 or the Natural Alignment of the argument's type.
C.14	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.

Continued on next page

Table 7.3 – continued from previous page

C.15	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.16	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.

7.5 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 *Parameter Passing* (page 12), 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 in AAPCS64 and c8 in AAPCS64-cap. 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 r8).

7.6 Interworking

Interworking between the 32-bit AAPCS and the AAPCS64 or AAPCS64-cap 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).

Interworking between data model variants of AAPCS64 (although technically possible) is not defined within a single process.

Interworking between AAPCS64 and AAPCS64-cap is not supported.

Interworking between A64 and C64 states is supported. The linker will insert a veneer at direct branches between different states. The veneer will perform both the state switch and range extensions. It is the responsibility of the callee to switch state on return.

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 Types Varying by Data Model and Procedure Calling Standard

The AAPCS64-cap uses different language mappings for any C/C++ Type that would be a code or data pointer in AAPCS64, while keeping other data types the same.

These differences, and new Morello-specific data types are shown in Table 8.1, C/C++ type variants by data model and PCS.

Table 8.1: C/C++ type variants by data model and PCS

C/C++ Type	Machine Type		Notes
	AAPCS64-cap	AAPCS64	
`T *`	Data Capability	64-bit data pointer	Any data type `T`
`T (*F)()`	Code Capability	64-bit code pointer	Any function type `F`
`T&`	Data Capability	64-bit data pointer	C++ reference
`T * __capability`	Data Capability	64-bit data pointer	Any data type `T`
`T (* __capability F)()`	Code Capability	64-bit code pointer	Any function type `F`
`T& __capability`	Data Capability	64-bit data pointer	C++ reference

8.1.2 Definition of va_list

The definition of va_list has implications for the internal implementation in the compiler. An AAPCS64 or AAPCS64-cap conforming object must use the definitions shown in Table 8.2, Definition of va_list.

Table 8.2: Definition of va_list

Typedef	Base type	Notes

`va_list`	`struct __va_list {`
	`  void *__stack;`
	`  void *__gr_top;`
	`  void *__vr_top;`
	`  int  __gr_offs;`
	`  int  __vr_offs;`
	`}`
		A `va_list` may address any object in a parameter list. In C++, `__va_list` is in namespace std. See aapcs64-appendixb. Variable Argument Lists. Note that __stack, __gr__top and __vr_top are capabilities in AAPCS64-cap.

CHAPTER

NINE

APPENDIX 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. The remainder of this appendix is informative.

9.1 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 three register save areas within its own stack frame: one area to hold the 64-bit general registers xn-x7, a second area to hold the 128-bit FP/SIMD registers vn-v7 and a third area to hold the capability registers cn-c7. 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 the first two areas 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.

The third register save area is located immediately above the general-purpose register area. The start of this area must be 16-byte aligned. It will contain the values of the capability registers cn-c7, stored starting at the address __gr_top from the highest numbered register to the lowest number register. All registers saved in this area must have their corresponding x sub-register stored in the general-purpose register save area as well. The capability register save area can be omitted if no capability arguments are used.

9.2 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;
```

In AAPCS64-cap the stack, gr_top and vr_top fields of va_list are capabilities, while in the AAPCS64 they are pointers.

9.3 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 - named_gr) * 8)`.
- `__vr_offs`: set to `0 - ((8 - 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 the NULL capability and zero respectively. 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 be 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 the NULL capability and 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.

9.4 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 (!containsCapabilities(type)) {
          if (alignof(type) > 8)
              offs = (offs + 15) & -16;   // round up
          nreg = (sizeof(type) + 7) / 8;
          ap.__gr_offs = offs + (nreg * 8);
        } else {
          offs = (offs + 7) & -8;// round up
          nreg = sizeof(type) / sizeof(void * __capability);
          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
        if (containsCapabilities(type)) {
          // Types containing capabilities are passed in capability
```

```
          // registers. Capability registers are stored in reverse
          // order.
          type T;
          for (unsigned reg = 0; reg < nreg; ++reg) {
            index = (reg + 1) * 16;
            ((void *__capability *)&T)[reg] =
                *(void *__capability)(ap.__gr_top - 2 * offs - index;
          }
          return T;
        }
        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;
}
```