Morello Extensions to DWARF for the Arm® 64-bit Architecture (AArch64)

2024Q3

Date of Issue: 5th September 2024

1 Preamble

1.1 Morello alpha

This document is an alpha proposal for Morello extensions to DWARF for AArch64.

1.2 Abstract

This document describes the use of the Morello extensions to the DWARF debug table format in the Application Binary Interface (ABI) for the Arm 64-bit architecture.

1.3 Keywords

DWARF, DWARF 3.0, DWARF 5.0, use of DWARF format

1.4 Latest release, feedback and defects report

Please check Application Binary Interface for the Arm® Architecture for the latest release of this document.

Please give feedback and report defects in this specification to the issue tracker page on GitHub.

1.5 Licence

1.6 About the license

As identified more fully in the Licence 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.

1.7 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 Licence section.

1.8 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. The Arm trademarks featured here are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved. Please visit https://www.arm.com/company/policies/trademarks for more information about Arm’s trademarks.

1.9 Copyright

Copyright (c) 2020-2024, Arm Limited and its affiliates. All rights reserved.

Contents

2 About this document

2.1 Change control

2.1.1 Current status and anticipated changes

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

Release

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

Beta

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

Alpha

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

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

2.1.2 Change history

If there is no entry in the change history table for a release, there are no changes to the content of the document for that release.

Issue	Date	Change
00alpha	September 2020	Alpha release.
2020Q4	21st December 2020	Document released on Github.

2.2 References

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

Ref	External reference or URL	Title
MDWARF64	This document	DWARF supplement for Morello
AADWARF64		DWARF for the Arm 64-bit Architecture (AArch64)
AAPCS64		Procedure Call Standard for the Arm 64-bit Architecture (AArch64)
GDWARF	http://dwarfstd.org/Dwarf3Std.php	DWARF 3.0, the generic debug table format.
GDWARF5	http://dwarfstd.org/Dwarf5Std.php	DWARF Debugging Information Format, Version 5

2.3 Terms and abbreviations

This ABI document uses the following terms and abbreviations:

A64

The instruction set that is available in AArch64 state.

AAPCS64

Procedure Call Standard for the Arm 64-bit Architecture (AArch64).

AArch64

The 64-bit general-purpose register width state of the Armv8 architecture.

ABI

Application Binary Interface:

1. The specifications to which an executable must conform in order to execute in a specific execution environment. For example, the Linux ABI for the Arm Architecture.
2. A particular aspect of the specifications to which independently produced relocatable files must conform in order to be statically linkable and executable. For example, the C++ ABI for the Arm Architecture, ELF for the Arm Architecture, ...

C64

The instruction set available when the Morello extensions are used.

3 Overview

This specification only provides the Morello-specific extensions to the base DWARF specification for the Arm 64-bit Architecture (AArch64), and is expected to be used together with AADWARF64.

4 Arm-specific DWARF definitions

4.1 DWARF register names

This specification adds DWARF register numbers and names for the capability registers.

Mapping from DWARF register numbers to Morello capability registers

DWARF register number	AArch64 register name	Description
198-228	C0-C30	Tagged 128+1-bit capability registers (Note 1, Note 2)
229	CSP	Tagged 128+1-bit capability stack pointer register (Note 1, Note 2)
230	PCC	Tagged 128+1-bit program counter capability register (Note 1, Note 2)
231	DDC	Tagged 128+1-bit default data capability register (Note 1, Note 2)
232	Reserved	
233	Reserved	

Notes

1. The architecture defines the following register overlaps:

    - General registers (X registers) overlap with the capability registers (C registers). A given X register is mapped to the low 64 bits of the corresponding C register.
    - Stack pointer (register SP) overlaps with the capability stack pointer (register CSP). Register SP is mapped to the low 64 bits of the CSP register.
    - Program counter (register PC) overlaps with the program counter capability (register PCC). Register PC is mapped to the low 64 bits of the PCC register.

    The DWARF call frame instructions do not explicitly specify the size of a register. This is implicit in the definition of the register. As a consequence, the overlapping registers have been allocated separate DWARF register number ranges which have their own definition for the size of these registers.

    When searching the call frame information table for one of these registers a consumer must take into account the aliasing between them and their overlapping registers.

2. Capability registers are stored in memory including their tag bit. When debug information describes that a capability register is saved at some address, a consumer must use a proper load of a capability from memory to restore its value.

    Capabilities are always stored in the little-endian byte order. This implies that if a given capability register C is stored in memory on a big-endian system, its corresponding X part is not stored in the natural byte order but as little-endian too.

4.2 Canonical frame address

The term Canonical Frame Address (CFA) is defined in GDWARF, §6.4, Call Frame Information. This ABI adopts the typical definition of CFA given there.

- The CFA is the value of the stack pointer at the call site in the previous frame.

A subroutine can define the CFA as either a 64-bit address or a 129-bit capability depending on whether the subroutine uses register SP as the stack pointer or register CSP, respectively.

The AAPCS64 document (AAPCS64, §5.2.2 Stack) describes what stack pointer is used by subroutines conforming to AAPCS64 and AAPCS64-cap:

- A subroutine conforming to AAPCS64 uses register SP as the stack pointer. The CFA in the subroutine is then defined as a 64-bit address, typically as `SP+<offset>` or `X29+<offset>`.
- A subroutine conforming to AAPCS64-cap uses register CSP as the stack pointer. The CFA in the subroutine is defined as a 129-bit capability, typically as `CSP+<offset>` or `C29+<offset>`.

If a subroutine defines the CFA as a 64-bit address, then only the lowest 64 bits from the value stored in register CSP in the previous frame can be restored. The consumer should treat the remaining bits as undefined.

4.3 Common information entries

4.3.1 Augmentation characters

This specification adds one CIE augmentation character that might appear as part of a CIE augmentation string:

- The 'C' character indicates that the default unwind rules for this CIE should be initialized in accordance with the pure capability procedure call standard.

4.3.2 Return address register

A subroutine can return to either a 64-bit address or a 129-bit capability depending on whether the subroutine expects the return address to be stored in register LR or CLR, respectively.

The AAPCS64 document (AAPCS64, §5.6 Function returns and the link register) describes where the return address for subroutines conforming to AAPCS64 and AAPCS64-cap is stored:

- A subroutine conforming to AAPCS64 expects the return address to be stored in register LR. The `return_address_register` field in the CIE is then set to a column for a 64-bit register, typically to the LR column directly.
- A subroutine conforming to AAPCS64-cap expects the return address to be stored in register CLR. The `return_address_register` field in the CIE is set to a column for a 129-bit capability register, typically to the CLR column.

If a subroutine uses a 64-bit return address register then it must also guarantee that it does not modify the upper 65 bits of register PCC. The value of PCC in the previous frame can then be restored by a consumer by using the PCC from the current frame and replacing the lowest 64 bits by the calculated value of the return address register in the previous frame.

4.4 DWARF attributes

4.4.1 Address classes

GDWARF, §5.3, Type Modifier Entries, describes attribute `DW_AT_address_class` that denotes how objects having the given pointer or reference type ought to be dereferenced. Section §2.12, Segmented Addresses then describes that the set of permissible values is specific to each target architecture.

This ABI uses the common address class value `DW_ADDR_none` and defines one Morello-specific value `DW_ADDR_capability`.

Morello DWARF address class codes

Address class name	Value
DW_ADDR_none	0x0
DW_ADDR_capability	0x1

1. `DW_ADDR_none`

   The `DW_ADDR_none` value is defined in GDWARF, §2.12, Segmented Addresses and means that no address class has been specified.

2. `DW_ADDR_capability`

   The `DW_ADDR_capability` value indicates that the type is an address capability and can be dereferenced as such.

4.4.2 Base type encodings

GDWARF, §5.1, Base Type Entries, describes attribute `DW_AT_encoding` that denotes how a base type is encoded and is to be interpreted.

This ABI uses two vendor-defined base-type encodings `DW_ATE_CHERI_signed_intcap` and `DW_ATE_CHERI_unsigned_intcap`.

Morello DWARF base type encoding values

Base type encoding name	Value
DW_ATE_CHERI_signed_intcap	0xa0
DW_ATE_CHERI_unsigned_intcap	0xa1

1. `DW_ATE_CHERI_signed_intcap`

   The `DW_ATE_CHERI_signed_intcap` encoding describes a signed integer/capability type `__intcap_t`.

2. `DW_ATE_CHERI_unsigned_intcap`

   The `DW_ATE_CHERI_unsigned_intcap` encoding describes an unsigned integer/capability type `__uintcap_t`.

4.4.3 Size attributes for capabilities

A size attribute of an entry that describes a capability type should record the untagged memory size of the data object. This typically means that such an entry should have the `DW_AT_byte_size` attribute with value set to 16.

Given that the formal size of a capability is 16 bytes, consumers of DWARF will have to implement ways to read the additional tag bit when required. This is implementation specific.

When inspecting capabilities, a debugger might choose to display only the untagged, 16-byte value, and provide the possibility for the user to further query the tag only if interested. In such a case, the value and the tag can be read independently of each other. Alternatively, it is perfectly valid for an implementation to always read and display the full 129 bits of the capability.

5 APPENDIX Supplementary material

The status of this appendix is informative.

5.1 Capability type examples

5.1.1 Capability pointers and references

The capability pointer type in example fragment Capability pointer: C++ source can be described in DWARF as shown in Capability pointer: DWARF description.

Capability pointer: C++ source

```
char * __capability cap;
```

Capability pointer: DWARF description

```
1$: DW_TAG_base_type
        DW_AT_name("char")
        ...
2$: DW_TAG_pointer_type
        DW_AT_type(reference to 1$)
        DW_AT_address_class(DW_ADDR_capability)
        DW_AT_byte_size(16)
```

Capability reference and rvalue reference types can be represented in the same way using `DW_TAG_reference_type` and `DW_TAG_rvalue_reference_type`, respectively.

5.1.2 Types `__intcap_t` and `__uintcap_t`

The `__intcap_t` and `__uintcap_t` types in example fragment Intcap types: C++ source can be described in DWARF as illustrated in Intcap types: DWARF description.

Intcap types: C++ source

```
__intcap_t intcap;
__uintcap_t uintcap;
```

Intcap types: DWARF description

```
1$: DW_TAG_base_type
        DW_AT_name("__intcap_t")
        DW_AT_encoding(DW_ATE_CHERI_signed_intcap)
        DW_AT_byte_size(16)
2$: DW_TAG_base_type
        DW_AT_name("__uintcap_t")
        DW_AT_encoding(DW_ATE_CHERI_unsigned_intcap)
        DW_AT_byte_size(16)
```

5.1.3 Optimized description in DWARF Version 5

Additional attributes needed to describe capability types can have recognizable impact on the size of debugging information in programs that extensively use capability pointers and references.

DWARF Version 5 introduced the new operand form `DW_FORM_implicit_const` (GDWARF5, §7.5.3, Abbreviations Tables) which allows to extract common attributes of capability types in section `.debug_abbrev` instead of repeating them in `.debug_info`.

Two capability pointer types in example fragment Capability pointers: C++ source can be described in DWARF Version 5 as illustrated in Capability pointers: DWARF Version 5 description.

Capability pointers: C++ source

```
char * __capability ccap;
int * __capability icap;
```

Capability pointers: DWARF Version 5 description

```
! *** Section .debug_abbrev content
a$h: 1
     DW_TAG_compile_unit
     ...
     0
     2
     DW_TAG_base_type
     DW_CHILDREN_no
     DW_AT_name                DW_FORM_string
     DW_AT_encoding            DW_FORM_data1
     DW_AT_byte_size           DW_FORM_data1
     0
     3
     DW_TAG_pointer_type
     DW_CHILDREN_no
     DW_AT_type                DW_FORM_ref4
     DW_AT_address_class       DW_FORM_implicit_const
                               DW_ADDR_capability
     DW_AT_byte_size           DW_FORM_implicit_const
                               16
     0

! *** Section .debug_info content
! Compilation unit header
     <length>                  ; unit_length
     5                         ; version
     DW_UT_compile             ; unit_type
     8                         ; address_size
     offset of a$h             ; debug_abbrev_offset
! Entry for the compile unit
     1
     ...
! Entry for "char"
i$1: 2
     "char"
     DW_ATE_unsigned_char
     1
! Entry for "int"
i$2: 2
     "int"
```

```
      DW_ATE_signed
      4
! Entry for "char * __capability"
      3
      offset of i$1
! Entry for "int * __capability"
      3
      offset of i$2
```

5.2 CFI assembler syntax for pure capability functions

It is recommended for a toolchain vendor to introduce assembler syntax that allows the description of call frame information for subroutines that conform to AAPCS64-cap as easily as for the ones conforming to AAPCS64.

For instance, listing GNU assembler syntax for pure capability functions shows the extension that was introduced to the GNU assembler syntax:

GNU assembler syntax for pure capability functions

```
.cfi_startproc purecap
```

Using the `purecap` parameter causes `.cfi_startproc` to do the following:

- Initial CFI instructions consist of one instruction `DW_CFA_def_cfa CSP, 0`.
- Return address register is set to CLR.
- The 'C' character is included in the CIE augmentation string.