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

2022Q3

Date of Issue: 20th October 2022

1 Preamble

1.1 Abstract

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

1.2 Keywords

DWARF, DWARF 3.0, use of DWARF format

1.3 Latest release and defects report

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

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

1.4 Licence

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 counterclaim 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.

1.5 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.

1.6 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.

1.7 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.

1.8 Copyright

Copyright (c) 2010, 2013, 2018, 2020-2022, 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.

Content relating to SVE should be considered as having **Beta** support level. This includes:

- DWARF register names marked as **Beta** in DWARF register names
- Recommended expression of the vector types (Vector types)

All other content in this document is at the **Release** 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
00bet3	16th December 2010	Beta release.
1.0	22nd May 2013	First public release.
2018Q4	31st December 2018	Add SVE and pointer authentication support.
2019Q4	30th January 2020	Minor layout changes.
2020Q2	1st June 2020	Add requirements for unwinding MTE tagged stack. Describe DWARF representation of SVE vector types.

Issue	Date	Change
2020Q4	21st December 2020	• document released on Github • new Licence: CC-BY-SA-4.0 • new sections on Contributions, Trademark notice, and Copyright • AArch64 DWARF pointer signing operations table columns switched • Add Thread ID register numbers.
2022Q1	1st April 2022	• Release of Pointer authentication. • In Call frame instructions, document a limitation of DW_CFA_AARCH64_negate_ra_state.
2022Q3	20th October 2022	• Added Changes in vector length at **Alpha** quality.

2.2 References

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

Ref	URL or other external reference	Title
AADWARF64	Source for this document	DWARF for the Arm 64-bit Architecture (AArch64). (*This document*)
GDWARF	http://dwarfstd.org/Dwarf3Std.php	DWARF 3.0, the generic debug table format.

2.3 Terms and abbreviations

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

A32

The instruction set named Arm 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.

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 Addenda32, AAPCS64, ...

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.

MTE

Memory Tagging Extension.

PAC

Pointer Authentication Code.

PAUTH

Pointer Authentication Extension.

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.

SVE

Scalable Vector Extension.

T32

The instruction set named Thumb in the Armv7 architecture; T32 uses 16-bit and 32-bit instructions.

3 Overview

The ABI for the Arm 64-bit architecture specifies the use of DWARF 3.0 format debugging data. For details of the base standard see GDWARF.

The ABI for the Arm 64-bit architecture gives additional rules for how DWARF 3.0 should be used, and how it is extended in ways specific to the Arm 64-bit architecture. The following topics are covered in detail:

- The enumeration of DWARF register numbers for using in `.debug_frame` and `.debug_info` sections (DWARF register names).
- The definition of *Canonical Frame Address* (CFA) used by this ABI (Canonical frame address).
- The definition of *Common Information Entries* (CIE) used by this ABI (Common information entries).
- The definition of *Call Frame Instructions* (CFI) used by this ABI (Call frame instructions).
- The definition of DWARF Expression Operations used by this ABI (dwarf expression operations).

4 Arm-specific DWARF definitions

4.1 DWARF register names

GDWARF, §2.6.1, Register Name Operators, suggests that the mapping from a DWARF register name to a target register number should be defined by the ABI for the target architecture. DWARF register names are encoded as unsigned LEB128 integers.

Mapping from DWARF register numbers to Arm 64-bit architecture registers

DWARF register number	AArch64 register name	Description
0–30	X0–X30	64-bit general registers (Note 1)
31	SP	64-bit stack pointer
32	PC	64-bit program counter (Note 9)
33	ELR_mode	The current mode exception link register
34	RA_SIGN_STATE	Return address signed state pseudo-register (Note 8)
35	TPIDRRO_ELO	EL0 Read-Only Software Thread ID register
36	TPIDR_ELO	EL0 Read/Write Software Thread ID register
37	TPIDR_EL1	EL1 Software Thread ID register
38	TPIDR_EL2	EL2 Software Thread ID register
39	TPIDR_EL3	EL3 Software Thread ID register
40-45	Reserved	-
46	VG (**Beta**)	64-bit SVE vector granule pseudo-register (Note 2, Note 3)
47	FFR (**Beta**)	VG × 8-bit SVE first fault register (Note 4)
48-63	P0-P15 (**Beta**)	VG × 8-bit SVE predicate registers (Note 4)
64-95	V0-V31	128-bit FP/Advanced SIMD registers (Note 5, Note 7)
96-127	Z0-Z31 (**Beta**)	VG × 64-bit SVE vector registers (Note 6, Note 7)

Note

1. The size of a general register is to be taken from context. For instance in a `.debug_info` section if the `DW_AT_location` attribute of a variable is `DW_OP_reg0` then the number of significant bits in the register is determined by the variable's `DW_AT_type` attribute. If no context is available (for example in `.debug_frame` or `.eh_frame` sections) then the register number refers to a 64-bit register.

2. The value of the SVE vector granule pseudo-register is an even integer in the range 2 to 32. The value of the register is the available size in bits of the SVE vector registers in the current call frame divided by 64.

3. The SVE vector granule pseudo-register enables the construction of DWARF expressions that require the use of the current vector length, such as the location of saved SVE predicate and vector registers on the stack using the DWARF stack frame operator `DW_CFA_expression`.

4. The available size of a SVE predicate register and the first fault register is VG × 8-bits.

5. In a similar manner to the general register file the size of an FP/Advanced SIMD register is taken from some external context to the register number. If no context is available then only the least significant 64 bits of the register are referenced. In particular this means that the most significant part of a SIMD register is unrecoverable by frame unwinding.

6. The available size of the SVE vector registers is VG × 64-bits.

7. The architecture defines that the FP/Advanced SIMD registers (V registers) overlap with the SVE vector registers (Z registers). A given V register is mapped to the low 128-bits of the corresponding Z 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 V registers and Z 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 either a V register or a Z register a consumer must take into account the aliasing between the V and Z registers.

8. The RA_SIGN_STATE pseudo-register records whether the return address has been signed with a PAC. This information can be used when unwinding. It is an unsigned integer with the same size as a general register. Only bit[0] is meaningful and is initialized to zero. A value of 0 indicates the return address has not been signed. A value of 1 indicates the return address has been signed.

9. Normally, the program counter is restored from the return address, however having both LR and PC columns is useful for describing asynchronously created stack frames. A DWARF expression may use this register to restore the context in case of a signal context.

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 (sp) at the call site in the previous frame.

4.3 Common information entries

The DWARF virtual unwinding model is based, conceptually, on a tabular structure with one column for each target register (GDWARF, §6.4.1, Structure of Call Frame Information). A `.debug_frame` Common Information Entry (CIE) specifies the initial values (on entry to an associated function) of each register.

The variability of execution environments conforming to the Arm architecture creates a problem for this model. A producer cannot reliably enumerate all the registers in the target. For example, an integer-only function might be included in one executable file for use in execution environments with floating-point and another for use in environments without. In effect, it must be acceptable for a producer not to initialize, in a CIE, registers it does not know about. In turn this generates an obligation on consuming debuggers to default missing initial values.

This generates the following obligations on producers and consumers of CIEs:

1. Consumers must default the CIE initial value of any target register not mentioned explicitly in the CIE.

   - Callee-saved registers (and registers intentionally unused by the program, for example as a consequence of the procedure call standard) should be initialized as if by `DW_CFA_same_value`, other registers as if by `DW_CFA_undefined`.

     A debugger can use built-in knowledge of the procedure call standard or can deduce which registers are callee-saved by scanning all CIEs.

   - The VG pseudo-register should be initialized as if by `DW_CFA_same_value`.
   - The `RA_SIGN_STATE` pseudo-register should be initialized as described in DWARF register names Note 8.

2. To allow consumers to reliably default the initial values of missing entries by scanning a program's CIEs, without recourse to built-in knowledge, producers must identify registers not preserved by callees, as follows:

   - If a function uses any register from a particular hardware register class (e.g. Arm core registers), its associated CIE must initialize all the registers of that class that are not callee-saved to `DW_CFA_undefined`.
   - If a function uses a callee-saved register R, its associated CIE must initialize R using one of the defined value methods (not `DW_CFA_undefined`).

   (As an optimization, a producer need not initialize registers it can prove cannot be used by any associated functions and their descendants. Although these are not callee-saved, they are not callee-used either.)

This ABI defines two CIE augmentation characters that may appear as part of a CIE augmentation string.

1. The character 'B' indicates that associated frames are using the B key for return address signing.
2. The character 'G' indicates that associated frames may modify MTE tags on the stack space they use.

Note

1. The mark on a frame recording that it may have set MTE tags other than the stack background is information which can be used when unwinding.

4.4 Call frame instructions

This ABI defines one vendor call frame instruction DW_CFA_AARCH64_negate_ra_state.

AArch64 vendor CFA operations

Instruction	High 2 bits	Low 6 bits	Operand 1	Operand 2
DW_CFA_AARCH64_negate_ra_state	0	0x2D	-	-

The DW_CFA_AARCH64_negate_ra_state operation negates bit[0] of the RA_SIGN_STATE pseudo-register. It does not take any operands. The DW_CFA_AARCH64_negate_ra_state must not be mixed with other DWARF Register Rule Instructions (GDWARF, §6.4.2.3) on the RA_SIGN_STATE pseudo-register in one Common Information Entry (CIE) and Frame Descriptor Entry (FDE) program sequence.

4.5 DWARF expression operations

This ABI defines one vendor DWARF expression operation DW_OP_AARCH64_operation.

AArch64 vendor DWARF expression operations

Operation	Code
DW_OP_AARCH64_operation	0xea

The DW_OP_AARCH64_operation takes one mandatory operand encoded as an unsigned LEB128. Bits[6:0] of this value specify an AArch64 DWARF Expression sub-operation. The remaining operands and the action performed are as specified by the sub-operation. The DW_OP_AARCH64_operation allows this ABI to define operations specific to the Arm 64-bit architecture outside the encoding space of DWARF expression operations.

AArch64 DWARF expression sub-operations

Sub-operation	Code
DW_SUB_OP_AARCH64_sign	0x00

The DW_SUB_OP_AARCH64_sign sub-operation takes a single operand encoded as an unsigned LEB128 operand. This value specifies a pointer key signing operation given in the AArch64 DWARF pointer signing operations table. The top two stack stack entries are popped, the first is treated as an 8-byte address value to be signed and the second is treated as an 8-byte salt. The key signing operation is performed on the address value using the salt, and the result is pushed to the stack.

AArch64 DWARF pointer signing operations

Operation	Code
Sign Instruction address with Key A	0x0
Sign Instruction address with Key B	0x1
Sign data address with Key A	0x2
Sign data address with Key B	0x3
Sign address with Generic key	0x4

4.6 Changes in vector length (Alpha)

In principle, the value of VG could change during the execution of a program. There are two main mechanisms by which this could happen:

- The program might explicitly change the SVE vector length. It could do this:
  - directly, by modifying the appropriate system registers (if it has sufficient permission)
  - indirectly, such as by using an operating system call
- The program might enter or leave SME streaming mode, such as by using the SMSTART and SMSTOP instructions. This would have the effect of changing VG on systems whose streaming vector length is different from their non-streaming vector length.

The following requirements apply to functions that might change VG in this way:

- The function's executable code must save the old value of VG to some location L before the operation that might change VG.
- The contents of L must be recoverable by an unwinder. One simple way of meeting this requirement is to make L be a location in the function's stack frame.
- The function's Frame Description Entry must describe the save of VG to L.
- The function's Frame Description Entry must describe whichever operation restores the old vector length as restoring VG from L.

4.7 Vector types (Beta)

The recommended way of describing an Advanced SIMD or SVE vector type is to use an array type (`DW_TAG_array_type`) that has the GNU vector type attribute (`DW_AT_GNU_vector`, code `0x2107`). The array index for these vectors has a lower bound of zero. For variable-length SVE vectors, the upper bound (`DW_AT_upper_bound`) or element count (`DW_AT_count`) is an expression based on the VG pseudo-register. For Advanced SIMD vectors and fixed-length SVE vectors, the upper bound or element count is constant.

For example, the recommended representation of the SVE type `svfloat32_t` is:

```
DW_TAG_array_type
  DW_AT_name("...")
  DW_AT_GNU_vector
  DW_AT_type(reference to float)
  DW_TAG_subrange_type
    DW_AT_upper_bound(expression=
      DW_OP_bregx(46, 0)
      DW_OP_lit2
      DW_OP_mul
      DW_OP_lit1
      DW_OP_minus)
```

if using `DW_AT_upper_bound` and:

```
DW_TAG_array_type
  DW_AT_name("...")
  DW_AT_GNU_vector
  DW_AT_type(reference to float)
  DW_TAG_subrange_type
    DW_AT_count(expression=
      DW_OP_bregx(46, 0)
```

```
        DW_OP_lit2
        DW_OP_mul)
```

if using `DW_AT_count`. Note that the zero lower bound is implicit for C and C++.