C++ Application Binary Interface Standard for the Arm® 64-bit Architecture

2021Q1

Date of Issue: 12th April 2021

arm

1 Preamble

1.1 Abstract

This document describes the C++ Application Binary Interface for the Arm 64-bit architecture.

1.2 Keywords

C++, Application Binary Interface, ABI, AArch64, C++ ABI, generic C++ ABI

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.

Copyright © 2010, 2013, 2018, 2020, 2021, Arm Limited and its affiliates. All rights reserved.

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, 2021, Arm Limited and its affiliates. All rights reserved.

Copyright © 2010, 2013, 2018, 2020, 2021, Arm Limited and its affiliates. All rights reserved.

Contents

1 Preamble **2**

1.1 Abstract 2

1.2 Keywords 2

1.3 Latest release and defects report 2

1.4 Licence 3

1.5 About the license 3

1.6 Contributions 3

1.7 Trademark notice 3

1.8 Copyright 3

2 About this document **5**

2.1 Change control 5

2.1.1 Current status and anticipated changes 5

2.2 Change History 5

2.3 References 5

2.4 Terms and Abbreviations 6

3 Overview **8**

3.1 The Generic C++ ABI 8

3.2 The Exception Handling ABI for the Arm architecture 8

4 The C++ ABI Supplement **9**

4.1 Summary of differences from and additions to the generic C++ ABI 9

4.2 Differences in detail 10

4.2.1 Representation of pointer to member function 10

4.2.2 Guard variables 10

4.2.3 Controlling Object Construction Order 10

4.2.4 ELF binding of static data guard variable symbols 11

4.2.5 Unwind Table Location 12

5 EH ABI Level I: Implementation ABI for AArch64 **13**

5.1 Transferring Control to a Landing Pad 13

6 EH ABI Level III: Implementation ABI for GNU Linux **14**

6.1 Introduction 14

6.2 Data Structures 14

6.3 Standard Runtime Initialization 14

6.4 Throwing an Exception 14

6.5 Catching an Exception 14

6.5.1 Overview of Catch Processing 14

6.5.2 The Personality Routine 14

6.5.3 Exception Handlers 14

Copyright © 2010, 2013, 2018, 2020, 2021, Arm Limited and its affiliates. All rights reserved.

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.

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

2.2 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	15th December 2010	Beta release.
1.0	22nd May 2013	First public release.
2018Q4	31st December 2018	Typographical changes.
2019Q4	30th January 2020	Add name mangling rules for half-precision Brain floating point format: Summary of differences from and additions to the generic C++ ABI.
2020Q2	1st July 2020	Specify behaviour between BTI and unwind library.
2021Q1	12th April 2021	• document released on Github • new Licence: CC-BY-SA-4.0 • new sections on Contributions, Trademark notice, and Copyright

2.3 References

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

Copyright © 2010, 2013, 2018, 2020, 2021, Arm Limited and its affiliates. All rights reserved.

Ref	URL or other reference	Title
AAPCS64		Procedure Call Standard for the Arm 64-bit Architecture
AAELF64		ELF for the Arm 64-bit Architecture
CPPABI64	*This document*	C++ ABI for the Arm 64-bit Architecture
GCPPABI	https://itanium-cxx-abi.github.io/cxx-abi/abi.html	Itanium C++ ABI ($Revision: 1.71 $) (Although called *Itanium C++ ABI*, it is very generic).
GELF	http://www.sco.com/developers/gabi/	Generic ELF, 17th December 2003 draft.
ISO C++	ISO/IEC 14882:2003 (14882:1988 with *Technical Corrigendum*)	International Standard ISO/IEC 14882:2003 – Programming languages C++
LSB	https://refspecs.linuxfoundation.org/LSB_4.0.0/LSB-Core-generic/LSB-Core-generic.html	Linux Standards Base Core Specification 4.0
IA64EHABI	https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html	Itanium C++ ABI: Exception Handling

2.4 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 C++ ABI for the Arm 64-bit Architecture [CPPABI64], or ELF for the Arm Architecture [AAELF64].

Arm-based

... based on the Arm architecture ...

Branch Target Identification

Security technique ensuring a degree of control flow integrity by marking valid targets of indirect branches.

Floating point

Copyright © 2010, 2013, 2018, 2020, 2021, Arm Limited and its affiliates. All rights reserved.

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.

T32

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

More specific terminology is defined when it is first used.

Copyright © 2010, 2013, 2018, 2020, 2021, Arm Limited and its affiliates. All rights reserved.

3 Overview

The C++ ABI for the Arm 64-bit architecture (CPPABI64) comprises the following sub-components.

- The generic C++ ABI, summarized in The Generic C++ ABI, is the referenced base standard for this component.
- The C++ ABI supplement in The C++ ABI Supplement details Arm-specific additions to and deviations from the generic standard.
- The generic C++ exception handling ABI summarized in The Exception Handling ABI for the Arm architecture, describes the language-independent and C++-specific aspects of exception handling.
- The C++ exception handling supplement for AArch64 in EH ABI Level I: Implementation ABI for AArch64 details Arm-specific adddtions to and deviations from the generic standard.
- The C++ exception handling ABI supplement for GNU/Linux in EH ABI Level III: Implementation ABI for GNU Linux details Arm-specific additions to and deviations from the generic standard for GNU/Linux systems.

The generic C++ ABI is implicitly an SVr4-based standard, and takes an SVr4 position on symbol visibility and vague linkage. This document does not cover extensions for DLL-based environments.

3.1 The Generic C++ ABI

The generic C++ ABI [GCPPABI] (originally developed for SVr4 on Itanium) specifies:

- The layout of C++ non-POD class types in terms of the layout of POD types (specified for this ABI by the Procedure Call Standard for the Arm 64-bit Architecture [AAPCS64]).
- How class types requiring copy construction are passed as parameters and results.
- The content of run-time type information (RTTI).
- Necessary APIs for object construction and destruction.
- How names with linkage are mangled (name mangling).

The generic C++ ABI refers to a separate Itanium-specific specification of exception handling. When the generic C++ ABI is used as a component of this ABI, corresponding reference must be made to the generic C++ exception handling ABI as summarised in The Exception Handling ABI for the Arm architecture, and the C++ exception handling ABI supplement in EH ABI Level III: Implementation ABI for GNU Linux.

3.2 The Exception Handling ABI for the Arm architecture

In common with [IA64EHABI], this ABI specifies table-based unwinding that separates language-independent unwinding from language specific aspects. The [IA64EHABI] specification describes three levels of ABI:

Level I. A Base API, interfaces common to all languages and implementations.

Level II. The C++ ABI, interfaces for interoperability of C++ implementations.

Level III. The Implementation ABI specific to a particular runtime implementation.

EH ABI Level I: Implementation ABI for AArch64 describes EH Level I ABI used on AArch64 systems.

The AArch64 C++ EH ABI uses Level II of the Itanium exception handling ABI as specified.

EH ABI Level III: Implementation ABI for GNU Linux describes the EH Level III ABI used on GNU/Linux systems.

Copyright © 2010, 2013, 2018, 2020, 2021, Arm Limited and its affiliates. All rights reserved.

4 The C++ ABI Supplement

4.1 Summary of differences from and additions to the generic C++ ABI

This section summarizes the differences between the C++ ABI for the Arm 64-bit architecture and the generic C++ ABI. Section numbers in captions refer to the generic C++ ABI specification. Larger differences are detailed in subsections of Differences in detail.

GC++ABI §2.2 POD Data Types

The GC++ABI defines the way in which empty class types are laid out. For the purposes of parameter passing in [AAPCS64], a parameter whose type is an empty class shall be treated as if its type were an aggregate with a single member of type unsigned byte.

> ***Note***
>
> Of course, the single member has undefined content.

GC++ABI §2.3 Member Pointers

The pointer to member function representation differs from that used by Itanium. See Representation of pointer to member function.

GC++ABI §2.8 Initialization guard variables

This ABI only specifies the bottom bit of the guard variable. See Guard variables.

GC++ABI §3.1.3 Return Values

When a return value has a non-trivial copy constructor or destructor, the address of the caller-allocated temporary is passed in the Indirect Result Location Register (r8) rather than as an implicit first parameter before the `this` parameter and user parameters.

GC++ABI §3.3.5 Controlling Object Construction Order

Global object construction is managed in a simplified way under this ABI. See Controlling Object Construction Order.

GC++ABI §3.4 Demangler

This ABI provides the demangler interface as specified in the generic C++ ABI. The ABI does not specify the format of the demangled string.

GC++ABI §5.1.5 Builtin Types

The `__bf16` is mangled as `u6__bf16`.

GC++ABI §5.2.2 Static Data

If a static datum and its guard variable are emitted in the same COMDAT group, the ELF binding [GELF] for both symbols must be `STB_GLOBAL`, not `STB_WEAK` as specified in [GCPPABI]. ELF binding of static data guard variable symbols justifies this requirement.

GC++ABI §5.3 Unwind Table Location

The exception unwind table shall be located by use of program header entries of type `PT_AARCH64_UNWIND`. See Unwind Table Location.

(No section in the generic C++ ABI) A library nothrow new function must not examine its 2nd argument

Library versions of the following functions must not examine their second argument.

Copyright © 2010, 2013, 2018, 2020, 2021, Arm Limited and its affiliates. All rights reserved.

```
::operator new(std::size_t, const std::nothrow_t&)
::operator new[](std::size_t, const std::nothrow_t&)
```

(The second argument conveys no useful information other than through its presence or absence, which is manifest in the mangling of the name of the function. This ABI therefore allows code generators to use a potentially invalid second argument – for example, whatever value happens to be in R1 – at a point of call).

(No section in the generic C++ ABI, but would be §2.2) POD data types

Pointers to `extern "C++"` functions and pointers to `extern "C"` functions are interchangeable if the function types are otherwise identical.

In order to be used by the library helper functions described below, implementations of constructor and destructor functions (complete, sub-object, deleting, and allocating) must have a type compatible with:

```
extern "C" void (*)(void* /* , other argument types if any */);
```

(No section in the generic C++ ABI) Namespace and mangling for the `va_list` type

The type `__va_list` is in namespace std. The type name of `va_list` therefore mangles to `St9__va_list`.

4.2 Differences in detail

4.2.1 Representation of pointer to member function

The generic C++ ABI [GCPPABI] specifies that a pointer to member function is a pair of words `<ptr, adj>`. The least significant bit of `ptr` discriminates between (0) the address of a non-virtual member function and (1) the offset in the class's virtual table of the address of a virtual function.

This encoding cannot work for the AArch64 instruction set where the architecture reserves all bits of code addresses.

This ABI specifies that `adj` contains twice the `this` adjustment, plus 1 if the member function is virtual. The least significant bit of `adj` then makes exactly the same discrimination as the least significant bit of `ptr` does for Itanium.

A pointer to member function is NULL when `ptr = 0` and the least significant bit of `adj` is zero.

4.2.2 Guard variables

The generic C++ ABI [GCPPABI] specifies the bottom byte of a static variable guard variable shall be 0 when the variable is not initialized, and 1 when it is. All other bytes are platform defined.

This ABI instead only specifies the value bit 0 of the static guard variable; all other bits are platform defined. Bit 0 shall be 0 when the variable is not initialized and 1 when it is.

4.2.3 Controlling Object Construction Order

The generic ABI specifies a `#pragma` and `.priority_init` section type to allow the user to specify object construction order. This scheme is not in wide use and so this ABI uses a different scheme which has several pre-existing implementations.

The compiler is responsible for sequencing the construction of top-level static objects defined in a translation unit in accordance with the requirements of the C++ standard. The run-time environment (helper-function library) sequences the initialization of one translation unit after another. The global constructor vector provides the interface between these agents as follows:

- Each translation unit provides a fragment of the constructor vector in an ELF section called `.init_array` of type `SHT_INIT_ARRAY` (=0xE) and section flags `SHF_ALLOC + SHF_WRITE`.

Copyright © 2010, 2013, 2018, 2020, 2021, Arm Limited and its affiliates. All rights reserved.

- Each element of the vector contains the address of a function of type `extern "C" void (* const)(void)` that, when called, performs part or all of the global object construction for the translation unit. Producers must treat `.init_array` sections as if they were read-only.
- The appropriate entry for an element referring to, say, `__sti_file` that constructs the global static objects in filecpp, is 0 relocated by `R_AARCH64_ABS64 (__sti_file)`.
- Object construction order may be controlled by appending an unsigned integer in the range 0-65535 (formatted as if by `printf("%05d", priority)`) to the name of the section. The linker must lay these sections out in ascending lexicographical order.
- Sections without a priority number appended are assumed to have a lower priority than those sections with a priority number. The linker should lay out sections without a priority number after those sections with.
- The priority values 0 to 100 inclusive are reserved to the implementation.
- Run-time support code iterates through the global constructor vector in increasing address order calling each identified initialization function in order.

4.2.4 ELF binding of static data guard variable symbols

The generic C++ standard [GCPPABI] states at the end of §5.2.2:

> Local static data objects generally have associated guard variables used to ensure that they are initialized only once (see 3.3.2). If the object is emitted using a COMDAT group, the guard variable must be too. It is suggested that it be emitted in the same COMDAT group as the associated data object, but it may be emitted in its own COMDAT group, identified by its name. In either case, it must be weak.

In effect the generic standard permits a producer to generate one of two alternative structures. Either:

```
COMDAT Group (Variable Name) {
    Defines Variable Name       // ELF binding STB_GLOBAL, mangled name
    Defines Guard Variable Name // ELF binding STB_WEAK, mangled name ...
}                               // (... this ABI requires STB_GLOBAL binding)
```

Or:

```
COMDAT Group (Variable Name) {
    Defines Variable Name       // ELF binding STB_GLOBAL, mangled name
}
+
COMDAT Group (Guard Variable Name) {
    Defines Guard Variable Name // ELF binding STB_WEAK, mangled name
}
```

A link step involving multiple groups of the first kind causes no difficulties. A linker must retain only one copy of the group and there will be one definition of Variable Name and one weak definition of Guard Variable Name.

A link step involving pairs of groups of the second kind also causes no difficulties. A linker must retain one copy of each group so there will be one definition of Variable Name and one weak definition of Guard Variable Name.

A link step involving a group of the first kind and a pair of groups of the second kind generates two sub-cases.

- If the linker discards the group that defines two symbols there is no problem.
- If the linker retains the group that defines both Variable Name and Guard Variable Name it must nonetheless retain the group called Guard Variable Name. There are now two definitions of Guard Variable Name with ELF binding STB_WEAK.

In this second case there is no problem provided the linker picks one of the definitions.

Copyright © 2010, 2013, 2018, 2020, 2021, Arm Limited and its affiliates. All rights reserved.

Unfortunately, [GELF] does not specify how linkers must process multiple weak definitions when there is no non-weak definition to override them. If a linker faults duplicate weak definitions there will be a functional failure.

This ABI requires the ELF binding of Guard Variable Name in the first structure to be `STB_GLOBAL`.

The rules codified in [GELF] then make all three linking scenarios well defined and it becomes possible to link the output of compilers such as armcc that choose the first structure with the output of those such as gcc that choose the second without relying on linker behavior that the generic ELF standard leaves unspecified.

4.2.5 Unwind Table Location

Exception tables are located in sections with the name `.eh_frame` and `.eh_frame_hdr`. Linkers shall put the `.eh_frame_hdr` section in a single text segment, with a `PT_AARCH64_UNWIND` program table entry identifying the unwind table header location.

Copyright © 2010, 2013, 2018, 2020, 2021, Arm Limited and its affiliates. All rights reserved.

5 EH ABI Level I: Implementation ABI for AArch64

See [IA64EHABI] §1.

5.1 Transferring Control to a Landing Pad

See [IA64EHABI] §1.6.3.

The unwind library may elect to transfer control to landing pads via a jump instruction that requires the jump target to be identified as a valid destination for indirect jumps.

> ***Note***
>
> The intent is that the landing pads should start with a `bti j` instruction (or equivalent), when they might operate in an environment with Branch Target Identification enabled.

Copyright © 2010, 2013, 2018, 2020, 2021, Arm Limited and its affiliates. All rights reserved.

6 EH ABI Level III: Implementation ABI for GNU Linux

6.1 Introduction

This section describes the Exception Handling Implementation ABI for GNU Linux systems.

It specifies:

- The format of the unwind tables
- Standard Runtime Initialization features
- Throwing an Exception
- Catching an Exception

This section follows the layout of [IA64EHABI] §3.

6.2 Data Structures

The format of the exception tables is as specified in [LSB] §II.11.6 (Exception Frames).

The codes used to describe the encoding of pointers used in the exception frame tables, are the values described in [LSB] §II.11.5.1 (DWARF Exception Header Encoding).

Note that in particular that the layout of the Language Specific Data Area (LSDA) is not specified by this ABI. The structure and layout of a LSDA is specific to a particular implementation of a personality routine.

6.3 Standard Runtime Initialization

See [IA64EHABI] §3.3.

6.4 Throwing an Exception

See [IA64EHABI] §3.4.

6.5 Catching an Exception

6.5.1 Overview of Catch Processing

Stack unwinding itself is begun by calling `__Unwind_RaiseException()`, and performed by the unwind library. See [IA64EHABI] §3.5 for a summary.

6.5.2 The Personality Routine

The personality routine is specified in [IA64EHABI] §2.5.2.

6.5.3 Exception Handlers

The behavior of exception handlers is described in [IA64EHABI] §2.5.3.

Copyright © 2010, 2013, 2018, 2020, 2021, Arm Limited and its affiliates. All rights reserved.