C++ ABI for the Arm 64-bit Architecture (AArch64)

Release 2020Q2

Document number: IHI 0059E

Arm Ltd

Jun 12, 2020

IHI0059E_2020Q2_00_en

CONTENTS

IHI0059E_2020Q2_00_en

Document number: IHI 0059E

Release version: 2020Q2

CHAPTER

ONE

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

Please check the Arm Developer site (https://developer.arm.com/architectures/system-architectures/software-standards/abi) for a later release if your copy is more than one year old.

Please report defects in this specification to arm.eabi@arm.com.

1.4 Licence

1.5 Non-Confidential Proprietary Notice

Arm Limited. Company 02557590 registered in England.
110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

1.6 Contents

Contents

CHAPTER

TWO

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

Issue	Date	By	Change
00bet3	15th December 2010	MGD	Beta release.
1.0	22nd May 2013	RE	First public release.
2018Q4	31st December 2018	OS	Typographical changes.
2019Q4	30st January 2020	TS	Add name mangling rules for half-precision Brain floating point format: *Summary of differences from and additions to the generic C++ ABI* (page 11).
2020Q2	1st July 2020	TS	Specify behaviour between BTI and unwind library.

2.3 References

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

Ref	URL or other reference	Title
AAPCS64	IHI 0055	Procedure Call Standard for the Arm 64-bit Architecture
AAELF64	IHI 0056	ELF for the Arm 64-bit Architecture

Continued on next page

Table 2.2 – continued from previous page

Ref	URL or other reference	Title
CPPABI64 (page 1)	This document	C++ ABI for the Arm 64-bit Architecture
GC++ABI	http://itanium-cxx-abi.github.io/cxx-abi/abi.html	Generic C++ ABI
Generic ELF	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[1]	https://refspecs.linuxfoundation.org/LSB_4.0.0/LSB-Core-generic/LSB-Core-generic.html	Linux Standards Base Core Specification 4.0
IA64EHABI[2]	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 Architecture* (page 1), ELF for the Arm Architecture, ...

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 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.

[1] https://refspecs.linuxfoundation.org/LSB_4.0.0/LSB-Core-generic/LSB-Core-generic.html
[2] https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html

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.

2.5 Your licence to use this specification

IMPORTANT: THIS IS A LEGAL AGREEMENT ("LICENCE") BETWEEN YOU (AN INDIVIDUAL OR SINGLE ENTITY WHO IS RECEIVING THIS DOCUMENT DIRECTLY FROM ARM LIMITED) ("LICENSEE") AND ARM LIMITED ("ARM") FOR THE SPECIFICATION DEFINED IMMEDIATELY BELOW. BY DOWNLOADING OR OTHERWISE USING IT, YOU AGREE TO BE BOUND BY ALL OF THE TERMS OF THIS LICENCE. IF YOU DO NOT AGREE TO THIS, DO NOT DOWNLOAD OR USE THIS SPECIFICATION.

"Specification" means, and is limited to, the version of the specification for the Applications Binary Interface for the Arm Architecture comprised in this document. Notwithstanding the foregoing, "Specification" shall not include (i) the implementation of other published specifications referenced in this Specification; (ii) any enabling technologies that may be necessary to make or use any product or portion thereof that complies with this Specification, but are not themselves expressly set forth in this Specification (e.g. compiler front ends, code generators, back ends, libraries or other compiler, assembler or linker technologies; validation or debug software or hardware; applications, operating system or driver software; RISC architecture; processor microarchitecture); (iii) maskworks and physical layouts of integrated circuit designs; or (iv) RTL or other high level representations of integrated circuit designs.

Use, copying or disclosure by the US Government is subject to the restrictions set out in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software – Restricted Rights at 48 C.F.R. 52.227-19, as applicable.

This Specification is owned by Arm or its licensors and is protected by copyright laws and international copyright treaties as well as other intellectual property laws and treaties. The Specification is licensed not sold.

1. Subject to the provisions of Clauses 2 and 3, Arm hereby grants to LICENSEE, under any intellectual property that is (i) owned or freely licensable by Arm without payment to unaffiliated third parties and (ii) either embodied in the Specification or Necessary to copy or implement an applications binary interface compliant with this Specification, a perpetual, non-exclusive, non-transferable, fully paid, worldwide limited licence (without the right to sublicense) to use and copy this Specification solely for the purpose of developing, having developed, manufacturing, having manufactured, offering to sell, selling, supplying or otherwise distributing products which comply with the Specification.
2. THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES EXPRESS, IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF SATISFACTORY QUALITY, MERCHANTABILITY, NONINFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE. THE SPECIFICATION MAY INCLUDE ERRORS. Arm RESERVES THE RIGHT TO INCORPORATE MODIFICATIONS TO THE SPECIFICATION IN LATER REVISIONS OF IT, AND TO MAKE IMPROVEMENTS OR CHANGES IN THE SPECIFICATION OR THE PRODUCTS OR TECHNOLOGIES DESCRIBED THEREIN AT ANY TIME.
3. This Licence shall immediately terminate and shall be unavailable to LICENSEE if LICENSEE or any party affiliated to LICENSEE asserts any patents against Arm, Arm affiliates, third parties who have a valid licence from Arm for the Specification, or any customers or distributors of any of them based upon a claim that a LICENSEE (or LICENSEE affiliate) patent is Necessary to implement the Specification. In this Licence; (i) "affiliate" means any entity controlling, controlled by or under common control with a party (in fact or in law, via voting securities, management control or otherwise) and "affiliated" shall be construed accordingly; (ii) "assert" means to allege infringement in legal or administrative proceedings, or proceedings before any other competent trade, arbitral

or international authority; (iii) “Necessary” means with respect to any claims of any patent, those claims which, without the appropriate permission of the patent owner, will be infringed when implementing the Specification because no alternative, commercially reasonable, non-infringing way of implementing the Specification is known; and (iv) English law and the jurisdiction of the English courts shall apply to all aspects of this Licence, its interpretation and enforcement. The total liability of Arm and any of its suppliers and licensors under or in relation to this Licence shall be limited to the greater of the amount actually paid by LICENSEE for the Specification or US$10.00. The limitations, exclusions and disclaimers in this Licence shall apply to the maximum extent allowed by applicable law.

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

CHAPTER

THREE

OVERVIEW

The *C++ ABI for the Arm 64-bit architecture* (page 1) comprises the following sub-components.

- The generic C++ ABI, summarized in *The Generic C++ ABI* (page 9), is the referenced base standard for this component.
- The C++ ABI supplement in *The C++ ABI Supplement* (page 11) 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* (page 10), 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* (page 15) 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* (page 16) 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 [GC++ABI] (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 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* (page 10), and the C++ exception handling ABI supplement in *EH ABI Level III: Implementation ABI for GNU Linux* (page 16).

3.2 The Exception Handling ABI for the Arm architecture

In common with [IA64EHABI[3]], this ABI specifies table-based unwinding that separates language-independent unwinding from language specific aspects. The [IA64EHABI[4]] 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 (page 15) 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 (page 16) describes the EH Level III ABI used on GNU/Linux systems.

[3] https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html
[4] https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html

CHAPTER

FOUR

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* (page 12).

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* (page 12).

GC++ABI §2.8 Initialization guard variables

This ABI only specifies the bottom bit of the guard variable. See *Guard variables* (page 12).

GC++ABI §3.1.4 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 §3.3.4 Controlling Object Construction Order

Global object construction is managed in a simplified way under this ABI. See *Controlling Object Construction Order* (page 12).

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 [Generic ELF] for both symbols must be STB_GLOBAL, not STB_WEAK as specified in [GC++ABI64]. *ELF binding of static data guard variable symbols* (page 13) 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* (page 14).

(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.

```
::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 [GC++ABI] 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 [GC++ABI] 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.
- 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 [GC++ABI] 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.

Unfortunately, [Generic ELF] 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 [Generic ELF] 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.

CHAPTER

FIVE

EH ABI LEVEL I: IMPLEMENTATION ABI FOR AARCH64

See [IA64EHABI[5]] §1.

5.1 Transferring Control to a Landing Pad

See [IA64EHABI[6]] §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.

[5] https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html
[6] https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html

CHAPTER

SIX

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[7]] §3.

6.2 Data Structures

The format of the exception tables is as specified in [LSB[8]] §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[9]] §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[10]] §3.3.

6.4 Throwing an Exception

See [IA64EHABI[11]] §3.4.

[7] https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html
[8] https://refspecs.linuxfoundation.org/LSB_4.0.0/LSB-Core-generic/LSB-Core-generic.html
[9] https://refspecs.linuxfoundation.org/LSB_4.0.0/LSB-Core-generic/LSB-Core-generic.html
[10] https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html
[11] https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html

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[12]] §3.5 for a summary.

6.5.2 The Personality Routine

The personality routine is specified in [IA64EHABI[13]] §2.5.2.

6.5.3 Exception Handlers

The behavior of exception handlers is described in [IA64EHABI[14]] §2.5.3.

[12] https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html
[13] https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html
[14] https://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html