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

Document number: ARM IHI 0059B, current through AArch64 ABI release 1.0
Date of Issue: 22nd May 2013

Abstract

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

Keywords

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

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

Please check the *ARM Information Center* (http://infocenter.arm.com/) for a later release if your copy is more than 3 months old (navigate to the *Software Development Tools* section, *Application Binary Interface for the ARM Architecture* subsection).

Please report defects in this specification to *arm* dot *eabi* at *arm* dot *com*.

Licence

Proprietary notice

Contents

1 ABOUT THIS DOCUMENT

1.1 Change Control

1.1.1 Current Status and Anticipated Changes

This document's status is released. Clarifications, extensions and minor changes should be expected.

1.1.2 Change History

Issue	Date	By	Change
00bet3	15th December 2010	MGD	Beta release.
1.0	22nd May 2013	RE	First public release.

1.2 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
CPPABI64	This document	C++ ABI for the ARM 64-bit Architecture
GC++ABI	http://mentorembedded.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	http://refspecs.freestandards.org/LSB_4.0.0/LSB-Core-generic/LSB-Core-generic/book1.html	Linux Standards Base Core Specification 4.0
IA64EHABI	http://www.codesourcery.com/public/cxx-abi/abi-eh.html	Itanium C++ ABI: Exception Handling

1.3 Terms and Abbreviations

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

Term	Meaning
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.

Term	Meaning
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,* …
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.
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.

1.4 Your Licence to Use This Specification

2 OVERVIEW

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

- The generic C++ ABI, summarized in §2.1, is the referenced base standard for this component.
- The *C++ ABI supplement* in §3 details ARM-specific additions to and deviations from the generic standard.
- The generic C++ exception handling ABI summarized in §2.2, describes the language-independent and C++-specific aspects of exception handling.
- The C++ exception handling ABI supplement for GNU/Linux in §4 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 environment.

2.1 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 §2.2, and the C++ exception handling ABI supplement in §4.

2.2 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.

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

Section 4 describes the EH Level III ABI used on GNU/Linux systems.

3 THE C++ ABI SUPPLEMENT

3.1 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 §3.2.

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 §3.2.1.

GC++ABI §2.8 Initialization guard variables

This ABI only specifies the bottom bit of the guard variable. See §3.2.2.

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 §3.2.3.

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 §5.2.2 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]. §3.2.4 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 §3.2.5.

(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.

3.2 Differences in detail

3.2.1 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.

3.2.2 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.

3.2.3 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.

3.2.4 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.

3.2.5 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.

4 EH ABI LEVEL III: IMPLEMENTATION ABI FOR GNU LINUX

4.1 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.

4.2 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.

4.3 Standard Runtime Initialization

See [IA64EHABI] §3.3.

4.4 Throwing an Exception

See [IA64EHABI] §3.4.

4.5 Catching an Exception

4.5.1 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.

4.5.2 The Personality Routine

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

4.5.3 Exception Handlers

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