Differences between v1 and v2 of the ABI for the ARM® Architecture

Document number:	ARM IHI 0047A
Date of Issue:	24th March 2005, reissued 30th November 2012

Abstract

This document describes the differences between versions 1 of the ABI for the ARM Architecture published in December 2003 and version 2 published in the first quarter of 2005.

Keywords

ABI for the ARM architecture, ABI base standard, embedded ABI

Proprietary notice

Contents

1 ABOUT THIS DOCUMENT

1.1 Change control

1.1.1 Current status and anticipated changes

This document has been released publicly. Anticipated changes to this document include:

- Typographical corrections.
- Clarifications.
- Compatible extensions.

1.1.2 Change history

Issue	Date	By	Change
2.0	24th March 2005	LS	First public release.
A	25th October 2007	LS	Document renumbered (formerly GENC-005701 v2.0).

1.2 References

This document refers to the following documents.

Ref	Status / External URL	Title
AADWARF		DWARF for the ARM Architecture
AAELF		ELF for the ARM Architecture
AAPCS		Procedure Call Standard for the ARM Architecture
ADDENDA		Addenda to, and errata in, the ABI for the ARM Architecture
BPABI	New in version 2.0 of the ABI.	Base Platform ABI for the ARM Architecture
BSABI		ABI for the ARM Architecture (Base Standard)
CLIBABI		C Library ABI for the ARM Architecture
CPPABI		C++ ABI for the ARM Architecture
EHABI		Exception Handling ABI for the ARM Architecture
EHEGI		Exception handling component specimen implementations
RTABI		Run-time ABI for the ARM Architecture
GC++ABI	http://mentorembedded.github.com/cxx-abi/abi.html	Itanium C++ ABI ($Revision: 1.71 $) (Although called *Itanium C++ ABI*, it is very generic).

1.3 Terms and abbreviations

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

Term	Meaning
AAPCS	Procedure Call Standard for the ARM 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*, the *Run-time ABI for the ARM Architecture*, the *C Library ABI for the ARM Architecture.*
AEABI	(Embedded) ABI for the ARM architecture (*this* ABI…)
ARM-based	… based on the ARM architecture …
core registers	The general purpose registers visible in the ARM architecture's programmer's model, typically r0-r12, SP, LR, PC, and CPSR.
EABI	An ABI suited to the needs of embedded, and deeply embedded (sometimes called *free standing*), applications.
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.
VFP	The ARM architecture's Floating Point architecture and instruction set

2 DIFFERENCES BETWEEN V2 AND V1 OF THE ABI

2.1 Overview

Version 2 of the *ABI for the ARM Architecture* comprises the ten components listed in §1.2 of this document. How the components fit together to define a coherent ABI is described in the *ABI for the ARM Architecture (Base Standard)* [BSABI].

Version 2 of the ABI is intended to extend and clarify version 1 without introducing incompatibilities. For example, it adds a tenth component – the *Base Platform ABI for the ARM Architecture* [BPABI] – to the nine components of version 1. However, in any endeavour of this scale undertaken with limited resources against a background of rapid change, it is inevitable that both defects in version 1, and changes to requirements for version 2, should generate some incompatibilities.

The following subsections of this document summarize the changes in perspective and detail between version 2 and version 1 of each component.

Version 2 also adds *Addenda to the ABI for the ARM Architecture* [ADDENDA]. This specifies late addition to the standard and is a placeholder for material that will be added during future maintenance of it.

2.2 ABI for the ARM Architecture (Base Standard)

This component gives an overview of the ABI.

- It depicts how the ABI components relate to one another and to relevant external standards.
- It briefly summarizes each component.

There are two significant changes to this component.

- Three new sections (§3.8, §3.9, and §3.10) summarize:
  - The *Base Platform ABI for the ARM Architecture* [BPABI].
  - The use of **ar** format in the ABI.
  - *Addenda to the ABI for the ARM Architecture*.

  From this overview perspective these new sections make a compatible extension to version 1.
- There is a change in perspective depicted in [BSABI] §2, Figure 1, *A schematic map of the ABI for the ARM Architecture and related standards*, described immediately below.

Change of perspective between version 2 and version 1

In version 2, the boundary between execution environments (platforms) and the *Base Platform ABI* has been adjusted to reflect current market realities better. We now depict three platform families at the executable file level.

- The bare platform family (so called bare metal).
- The DLL-based family (for example, Palm OS, Symbian OS).
- The SVr4-based family (for example, Linux, free BSD).

Within the *Base Platform ABI* there are the same procedure call standard and data addressing variants as were depicted in version 1 of the ABI, but wrapped in three platform family standards rather than four virtual platforms. We justify this change of perspective on three grounds.

- It makes no difference to the operating systems, and removes the need to define virtual ABIs related to them.

- It directly reflects the tool flow within a conforming tool chain, and defines a clear interface between a generic, cross platform, binary file format and the platform-specific post linkers that generate platform-specific binaries.
- It decouples executable file organization from procedure call standard and data addressing concerns, which is, in any case, natural for tool chains producing executable or linkable files.

Figure 2, *Base platform ABI tool flow and its relationship to concrete platforms* in §2.3, *The base platform ABI tool flow*, of [BPABI] depicts the tool flow in more detail.

2.3 Procedure Call Standard for the ARM Architecture

There are few changes between version 2 and version 1 of the procedure call standard [AAPCS]. Each one embodies a better compromise with prior art.

2.3.1 Oversize bitfields

The layout of oversize bit fields is underspecified by the C and C++ language standards. Eventually, the ABI group decided to adopt the definition used by the generic C++ ABI and the GNU C++ compiler. This constitutes a small, dark corner, incompatibility between version 2 and version 1 (see [AAPCS] §7.1.7.3, *Over-sized bit-fields*).

2.3.2 Size of enum containers

Since the publication of version 1 of the ABI it has become clear that not all users of it can agree whether enum containers should be just big enough to hold all values of the type (to minimize memory footprint) or 32 bits (to avoid disaster when an enumeration is extended across an 8-bit or 16-bit boundary).

Version 2 of this ABI requires the container policy to be specified by the ABI for the execution environment. For example, Linux and Palm OS require 32-bit enums, while the bare platform often requires containerized enums.

With careful use of guard values it is possible to write source code that will generate the same binary (using 32-bit enum containers) independent of any platform-specific specification. This strategy is recommended for interfaces to portable packages or subsystems.

2.3.3 Type of enum values

Version 2 of [AAPCS] (Table 5, *Enumerator container types* in §7.1.3, *Enumerated Types*) specifies the type of an enumerated value so that in any context in which the *usual integral promotions* of C or C++ apply, the value will promote to type int whenever type int is capable of representing the value.

In version 1, some values promoted to unsigned int even when type int could represent the value.

This is a small, dark corner change to an area of compiler behavior that is often not well specified.

2.4 C++ ABI for the ARM Architecture

With the exception of the change to over sized bit field layout noted in §2.3, above, there are no deliberately incompatible changes between version 1 and version 2 of the *C++ ABI for the ARM Architecture*.

Otherwise, the following clarifications are significant and might reveal incompatibilities between implementations.

- Version 2 clarifies the definition of *key function* and adds an example of how the *C++ ABI for the ARM Architecture* deviates from the generic C++ ABI (no change of intended behavior from version 1).
- Version 2 makes clear that each platform must specify whether RTTI must be identical by address (as required by the generic C++ ABI [GC++ABI]) or merely by value (the minimum required by the C++ language

standard). The former is appropriate to SVr4-based platforms (such as Linux), the latter to platforms that cannot resolve vague linkage at dynamic link time (such as Symbian OS).

- Version 2 makes clear that C1 and C2 constructors and D1 and D2 destructors must return *this* (to support tail calling from C3 and to remove the need for D0 to save *this* across calls to D1/D2).
- In version 2, the specimen implementation of __aeabi_vec_delete now behaves correctly if the destructor throws an exception. __aeabi_vec_delete3 has been extended to cope with the deallocator throwing too.
- Version 2 requires top level static constructor array elements to be relocated by R_ARM_TARGET1 rather than R_ARM_RELABS32. This is a change of perspective, not of relocation code (the numerical value is the same).
- Version 2 fixes a defect in version 1 with respect to registering static object destructions. To do this in a way that works for all platforms (including those that statically allocate space for the list of object to destroy), destructions must be registered using __aeabi_atexit, not __cxa_atexit. Implementations that follow version 1 are likely to be defective.
- In version 2, a new section (§3.2.5) describes the control of visibility of entities (import and export control) between DLLs. Only §3.2.5.5, *Inter-DLL visibility rules for C++ ABI-defined symbols*, is mandatory.

2.5 Exception Handling ABI for the ARM Architecture

Between version 1 and version2 of the exception handling ABI there is one significant change to the top level organization of exception handling tables, and a number of changes of detail. These relate most significantly to fully supporting C++ exception handling semantics.

There can be no binary compatibility between implementations conforming to version 1 (if, indeed, such conformance was ever possible) and those conforming to version 2 (for which there are implementations).

2.5.1 Top level organizational change

The version 1 specification tried to allow exception index tables and exception handling tables to be allocated to their own, independently loaded, read only, executable segments. Thus an executable file could have from 1-3 independently loaded, read only, executable segments. This generated two difficulties.

- It made inter-segment references difficult, requiring new, unfamiliar relocation modes (and more segment base addresses to be juggled by run-time systems), or many more dynamic relocations (undesirable, and often impossible for read only segments).
- No operating system we know of supports executable files with multiple read only segments.

Version 2 of the ABI allows exception handling tables and exception index tables to be read only, position independent, and free of dynamic relocations, but does not support their separation into independently loaded segments.

This change would introduce significant incompatibility between implementations conforming to the version 1 and version 2 specifications. On the other hand, we believe that a full version 1 implementation was never possible (the required relocation modes were never defined), and that we made the change when implementers were first struggling with the specification.

A number of changes of detail follow from this change.

2.5.2 Changes of detail

This section lists version 2 details that were different in version 1.

- In a generic exception handling table entry (§4.4.2, §6.2) the pointer to the table handling function is now encoded as a place-relative offset (R_ARM_PREL31). In version 1 it was a segment relative offset.

- A new section (§4.4.2, *Relocations*) makes explicit the obligation on producers of relocatable file producers to emit exception handling table fragments that can be read only, position independent, and free of dynamic relocations. This section specifies the relocations to be used.
- A revised section (§5, *Index table entries*) specifies the relocation of index tables. Version 2 index tables are incompatible with version 1 tables (a 31-bit place-relative relocation replaces a 32-bit segment relative one).
- In the ARM-defined compact model (§6.3) a 31-bit place-relative relocation replaces a 32-bit segment relative one (as immediately above).

The above changes follow from the top level organizational change.

The following changes are not consequences of the top level organizational change.

- Version 2 clarifies use of the barrier cache data record by personality routines (in §7.2, *Language-independent unwinding types and functions*).
- Version 2 withdraws support for the obsolete FPA floating point unit (removed from §7.5.1, *Control types*).
- Version 2 better supports restoring VFP registers, adding restore with FSTMD to the previously defined use of FSTMX (which is partly implementation defined and, hence, less suitable for portable use).
- Version 2 makes minor changes to the use of a complete C++ exception object (§8.2). A field used in version 1 is now required to be 0.
- Version 2 significantly changes the type signature of __cxa_type_match (§8.4.2, *Personality routine helper functions*). The result type changes from bool to an enumerated type, and it gains an additional parameter. This incompatible change is required to correctly support C++ exception processing semantics. Version 2 also clarifies the behavioral description of __cxa_type_match.
- Version 2 specifies some small but significant changes to the encoding of exception-handling table entries (§9.2) for catch descriptors. To correctly implement C++ exception handling semantics, a descriptor must distinguish catching T& from catching T. Coincidentally, the change of top level organization frees up 1 bit in the previous encoding (where a 32-bit segment relative offset to the RTTI for T becomes a 31-bit place relative offset to the RTTI, leaving 1 bit to distinguish T& from T).
- Version 2 clarifies and extends the set of supported frame unwinding instructions (§9.3, *Frame unwinding instructions*), as well as withdrawing those previously associated with the FPA floating point unit.

2.5.3 Exception handling component specimen implementations

The exception handling component specimen implementations have tracked the changed in specification between version 1 and version 2 of the exception handling ABI.

A number of defects found during testing have also been corrected.

2.6 ELF for the ARM Architecture

At version 1, *ELF for the ARM Architecture* was a partially completed draft. Version 2 [AAELF] completes the specification. There are many new sections. We believe there are no significant incompatible *changes* to those sections that were mandatory (not optional) in version 1.

Version 2 takes the December 2003 draft of the underlying ELF specification as its base standard rather than the functionally identical April 2001 draft (the later draft corrects a typographical defect).

Version 2 adds material to the previously empty §3, *PLATFORM STANDARDS*. At issue 0.61 it adds requirements related to the *Base Platform ABI for the ARM Architecture* (§2.10 and [BPABI]) as follows.

- Symbol versioning (in §3.1.1).
- Symbol pre-emption in DLLs (in §3.1.2).
- PLT [code] sequences and usage models (in §3.1.3).

There are also the following detailed changes in version 2.

- In §4.2.1, *ELF Identification*, version 2 adds an obligation to set EI_OSABI to a conforming value.
- Version 2 adds a second ARM-specific section type (in Table 4 2, *Processor specific section types4*, in §3.2, *Section Types*), and a matching special section name in Table 4-3.

Version 2 completely revises the enumeration of processor-specific relocation types (in §4.6). Some obsolete values have been withdrawn, and several values defined in version 1 are *deprecated*. These should no longer be generated by producers, though consumers must continue to process them if they need to handle legacy relocatable files. Many new relocation types have been introduced, including some related to the new Thumb2 instruction set.

There will most likely be further detailed changes before version 2 is published.

2.7 DWARF for the ARM Architecture

There are no deliberately incompatible changes between version 1 and version 2.

- Version 2 adds the obligation (on producers, §2.1.1, *Support for stack unwinding*) to always generate frame unwinding descriptions. This is to support unwinding *through* packages. Code that does not conform can be considered to be blocking unwinding deliberately.
- Version 2 defines an idealized debugging illusion (§2.1.2, *The debugging illusion (not mandatory)*). There is no obligation to conform, but we present it as a simple model to which producers might aspire when asked to give priority to the quality of debugging experience.
- Version 2 corrects 2 small typographical defects in Table 1, *Mapping from DWARF register number to ARM architecture register number* (in §3.1, *DWARF register names*).
- Version 2 adds a section (§3.5, *Common information entries*) that specifies a required default interpretation of register numbers missing from CIE entries. These arise naturally and unavoidably under the ARM architecture (for example when generating portable relocatable code to run on an unknown target), so a defined default is necessary. In turn this generates clear obligations on both consumers and producers.

2.8 Run-time ABI for the ARM Architecture

There are no deliberately incompatible changes between version 1 and version 2.

- In version 2, use of floating point terminology has been tightened (§4.1.1, *The floating point model*).
- Version 2 clarifies the Notes on Table 3, above, and Table 5, below (in §4.1.2, *The floating-point helper functions*), making the required status flag settings unambiguous.
- Version 2 adds two new 32-bit integer division functions (__aeabi_idiv and __aeabi_uidiv) that return only the quotient. This is in deference to the Cortex M3 division instructions that behave similarly (the body of each of these functions can be replaced by a single instruction when executing on that CPU).
- Version 2 supports a third option for handling division by zero, namely that of returning a fixed value (such as zero). This is in deference to execution environment that require this behavior.
- There are some minor corrections to the descriptions of C++ helper functions defined by the generic C++ ABI (in §4.4.3.1, *Helper functions defined by the generic C++ ABI*). In two cases this ABI requires the helpers to return *this*.
- Version 2 gives a more detailed description of static object finalization (§4.4.5, *Static object finalization*) and of the difference between calling __aeabi_atexit and __cxa_atexit to register destructions. We referred to this change in the penultimate bullet in §2.4, above.

2.9 C Library ABI for the ARM Architecture

2.9.1 Changes of perspective

There are two small, but important, changes of perspective between version 1 and version 2 of the *C Library ABI for the ARM Architecture* [CLIBABI] that drive many of the remaining detailed differences.

- Version 2 abandons the direct, but fiddly, implementation of *link time constants* as absolute symbols in favor of a slightly less efficient, but more accessible, implementation using *extern const int*… (§4.2.3.1, *Compile time constants*). This allows link time constants to be defined and maintained in C rather than only in assembly language.
- Version 2 introduces a compile time (pre-processor) test of conformance that can be applied header by header to the C library, and specifies conformance requirements for each header (§5.1.1, *Detecting whether a header file honors an AEABI portability request*). This allows library implementations to opt into conformance header by header, and lets an application check at compile time whether its conformance requirements can be met.

Version 1 was released at issue 0.1 DRAFT status, so, formally, this breaks no promise of binary compatibility.

2.9.2 Detailed changes

Version 2 makes the following detailed changes to the version 1 specification.

- Version 2 corrects a defect in the definition of the *assert* macro and introduces an obligation to signal conformance (as described in §5.1.1 of the *C Library ABI for the ARM Architecture* and referred to from the second bullet of §2.9.1, above).
- Version 2 defines 3 ways for a *ctype* implementation to conform to the ABI, and adds the §5.1.1 requirement for conformance to be detectable at compile time.
- Substantive changes to the *errno*, *limits*, *locale*, *stdlib*, *time*, and *wctype* ABIs follow entirely from the changes in perspective described above in §2.9.1.
- Version 2 adds to the *math* ABI the §5.1.1 requirement for conformance to be detectable at compile time. For producers that implement some of the C99 standard in this area the result is to require what C99 requires, together with the ability to detect conformance at compile time.
- Version 2 simplifies the *setjmp* ABI by reducing what an application can count on portably. We believe this is the right compromise.
- Most of the substantive changes to the *signal* ABI follow from the changes in perspective described above in §2.9.1. Version 2 also fixes some typographical defects in version 1, adds a specification of sig_atomic_t, and clarifies the alternative ways for conforming implementations to define the standard signal handlers.
- Most of the substantive changes to the *stdio* ABI follow from the changes in perspective described above in §2.9.1. Version 2 revises the definition of fpos_t by adding the mbstate_t member required (in effect) by the C99 library standard.
- Version 2 of the *wchar* ABI relaxes the alignment requirement on an mbstate_t structure. There was never a good argument for treating this as a single 8-byte value. There are also the §5.1.1 consequences described above in §2.9.1.

Version 2 also adds header by header summaries of requirements, link time constants, and additional ABI-defined functions.

2.10 Base Platform ABI for the ARM Architecture

This is an entirely new component in version 2 of the ABI. There are, therefore, no incompatibilities with version 1.

For a summary of the BPABI see §3.8, *The base platform ABI for the ARM architecture*, of [BSABI]. For an overview see §2.1, *The role of this standard in the ABI for the ARM Architecture*, of [BPABI].

2.11 Addenda to the ABI for the ARM Architecture

As of the date of publication recorded on the first page of this document, *Addenda to, and errata in, the ABI for the ARM Architecture* describes two late additions to v2.0 (this version) of the ABI, and records no errata.

The two addenda concern *Build Attributes* and *Thread Local Storage*.

- Build attributes record long-lived facts about how a relocatable file was built. They allow a static linker to determine incompatibilities between relocatable files and to choose the best variant of a library function when multiple variants are available.
- The ABI issues associated with Thread Local Storage are sketched and the concrete models adopted by *Linux for ARM* are described. Linux-specific TLS relocations are described in [AAELF]