Application Binary Interface for the Arm® Architecture – The Base Standard

2025Q1

Date of Issue: 07th April 2025

arm

1 Preamble

1.1 Abstract

This document describes the structure of the Application Binary Interface (ABI) for the Arm architecture, and links to the documents that define the base standard for the ABI for the Arm Architecture. The base standard governs inter-operation between independently generated binary files and sets standards common to Arm-based execution environments.

1.2 Keywords

ABI for the Arm architecture, ABI base standard, embedded 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 © 2003, 2005, 2007, 2008 2018, 2020-2025, 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) 2003, 2005, 2007, 2008 2018, 2020-2025, Arm Limited and its affiliates. All rights reserved.

Copyright © 2003, 2005, 2007, 2008 2018, 2020-2025, 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

2.1.2 Change history 5

2.2 References 6

2.3 Terms and abbreviations 6

2.4 Acknowledgements 7

3 Schematic map of the ABI for the Arm Architecture **8**

3.1 Notes about the schematic map 8

4 The ABI for the Arm Architecture (base standard) **10**

4.1 Overview and documentation map 10

4.2 Procedure call standard for the Arm architecture 11

4.3 C++ ABI for the Arm architecture 11

4.3.1 The Generic C++ ABI 12

4.3.2 The C++ ABI supplement for the Arm architecture 12

4.3.3 The Exception handling ABI for the Arm architecture 12

4.3.4 The exception handling components specimen implementation 13

4.4 ELF for the Arm architecture 13

4.4.1 The generic ELF specification 13

4.4.2 ELF for the Arm architecture (processor supplement) 14

4.5 DWARF for the Arm architecture 14

4.5.1 DWARF 3.0 14

4.5.2 ABI DWARF usage conventions 14

4.6 Run-time ABI for the Arm architecture 15

4.7 The C library ABI for the Arm architecture 15

4.8 The base platform ABI for the Arm architecture 15

4.9 A note about `ar` format 16

4.10 Addenda to and errata in the ABI for the Arm Architecture 17

4.10.1 Build attributes 17

4.10.2 Thread local storage 17

Copyright © 2003, 2005, 2007, 2008 2018, 2020-2025, 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.

2.1.2 Change history

If there is no entry in the change history table for a release, there are no changes to the content of the document for that release.

Issue	Date	Change
1.0	30th October 2003	First public release.
2.0	24th March 2005	Second public release.
A	24th October 2007	Document renumbered (formerly GEN-003535 v2.0).
B	10th October 2008	A note about ar format fixed a typo and updated the reference to ar format.
2018 Q4	21st December 2018	Minor typographical fixes, updated links.
2019 Q4	30th January 2020	Minor layout changes.
2021 Q1	12th April 2021	• document released on Github • new Licence: CC-BY-SA-4.0 • new sections on Contributions, Trademark notice, and Copyright

Copyright © 2003, 2005, 2007, 2008 2018, 2020-2025, Arm Limited and its affiliates. All rights reserved.

2.2 References

This document refers to the following documents.

Ref	External URL	Title
AADWARF32		DWARF for the Arm Architecture
AAELF32		ELF for the Arm Architecture
AAPCS32		Procedure Call Standard for the Arm Architecture
ADDENDA32		Addenda to, and errata in, the ABI for the Arm Architecture
BPABI32		Base Platform ABI for the Arm Architecture
BSABI32	This standard	ABI for the Arm Architecture (Base Standard)
CLIBABI32		C Library ABI for the Arm Architecture
CPPABI32		C++ ABI for the Arm Architecture
EHABI32		Exception Handling ABI for the Arm Architecture
EHEGI		Exception handling, components, example implementations
RTABI32		Run-time ABI for the Arm Architecture
GCPPABI	https:/itanium-cxx-abi.github.io/cxx-abi/abi.html	Generic C++ ABI
GDWARF	http://dwarfstd.org/Dwarf3Std.php	DWARF 3.0, the generic debug table format.
GABI	http://www.sco.com/developers/gabi/	Generic ELF, 17th December 2003 draft.
GLSB	https://refspecs.linuxfoundation.org/lsb.shtml	gLSB v1.2 Linux Standard Base
OpenBSD	https://www.openbsd.org/	Open BSD Standard

2.3 Terms and abbreviations

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

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*.

Copyright © 2003, 2005, 2007, 2008 2018, 2020-2025, Arm Limited and its affiliates. All rights reserved.

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 [CPPABI32], the [RTABI32], the [CLIBABI32].

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 toolchain-specific means by which a standard requirement is met.

VFP

The Arm architecture's Floating Point architecture and instruction set. In this ABI, this abbreviation includes all floating point variants regardless of whether or not vector (V) mode is supported.

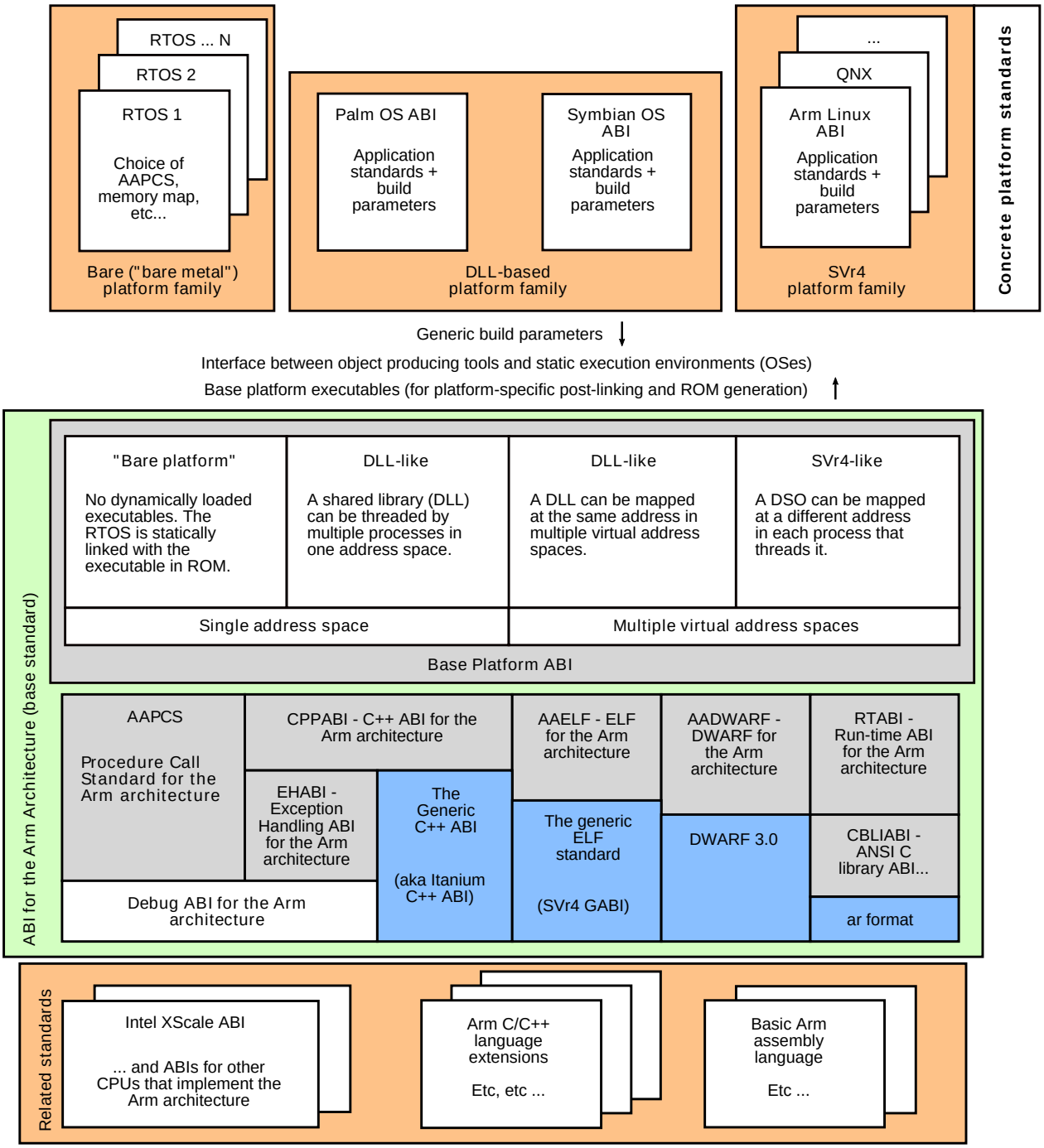
2.4 Acknowledgements

This specification has been developed with the active support of the following organizations. In alphabetical order: Arm, CodeSourcery, Intel, Metrowerks, Montavista, Nexus Electronics, PalmSource, Symbian, Texas Instruments, and Wind River.

Copyright © 2003, 2005, 2007, 2008 2018, 2020-2025, Arm Limited and its affiliates. All rights reserved.

3 Schematic map of the ABI for the Arm Architecture

A schematic map of the ABI for the Arm Architecture and some related standards



3.1 Notes about the schematic map

Pale gray boxes depict the most important components of the base standard for the *ABI for the Arm Architecture*.

Copyright © 2003, 2005, 2007, 2008 2018, 2020-2025, Arm Limited and its affiliates. All rights reserved.

Pastel blue (or darker gray on a gray-scale printed copy) boxes depict the most important external standards we refer to. We do not show them all – for example, we also refer to the ANSI standards for programming languages C and C++ and to the IEEE 754 standard for floating-point arithmetic.

The tan (also darker gray on a gray-scale printed copy) annotation boxes label groups of related standards that might be developed in the future, and a pastel green box (pale gray on a gray-scale printed copy) encloses all components (direct and referenced) of the *ABI for the Arm Architecture* (base standard).

The size of each box is unrelated to the size or significance of the component depicted.

Sections depicted with white boxes on a tan background are beyond the scope of *this* base standard. In each case the section involves either or both of the following.

- A third party on whom there is no obligation to contribute.
- Future intentions to which there is no current commitment.

The sections depicted with white boxes on a tan background show the position of *this base standard* in a larger context. They depict some of the ways in which those affected by this ABI standard might like to grow it, and how the base standard would relate to other plausible pieces of a larger jigsaw of Arm architecture-related standards. In no case shall this depiction be interpreted as an intention or commitment by Arm or any third party to create the component standard depicted.

The ABI for the Arm Architecture (base standard), below, describes the base standard in detail and refers to each of its components.

Copyright © 2003, 2005, 2007, 2008 2018, 2020-2025, Arm Limited and its affiliates. All rights reserved.

4 The ABI for the Arm Architecture (base standard)

4.1 Overview and documentation map

The *ABI for the Arm Architecture* is a collection of standards, some open and some specific to the Arm architecture, that regulate the inter-operation of binary files and development tools in a spectrum of Arm-based execution environments from bare metal to major operating systems such as Arm Linux. We expect that ABIs for specific execution environments will build on, and extend, the slices of this ABI that apply to them.

Standardizing the inter-operation of binary files requires standardizing certain aspects of code generation itself, so this base standard is aimed principally at the authors and vendors of C and C++ compilers, linkers, and run-time libraries. In general, there can be no complying executable files until there are complying relocatable files.

Documentation map of the ABI for the Arm architecture base standard

Component standard	Base standard
The *Procedure Call Standard for the the Arm Architecture* [AAPCS32] is summarized in Procedure call standard for the Arm architecture.	None.
The *C++ ABI for the Arm Architecture* [CPPABI32] is summarized in C++ ABI for the Arm architecture. It details where the C++ ABI for the ABI deviates from the base standard.	The Generic C++ ABI (*aka C++ ABI for Itanium*). https:/itanium-cxx-abi.github.io/cxx-abi/abi.html
The *Exception Handling ABI for the Arm Architecture* [EHABI32] is summarized in The Exception handling ABI for the Arm architecture. It describes C++-specific and language-independent exception processing.	
ELF for the Arm Architecture [AAELF32] is summarized in ELF for the Arm architecture. It gives processor-specific and platform-specific details not given in the generic ELF specification.	The generic ELF standard (SVr4 GABI), 17th December 2003 draft. http://www.sco.com/developers/gabi/
DWARF for the Arm Architecture [AADWARF32] is summarized in DWARF for the Arm architecture. It describes how DWARF should be used to promote inter-operation between independent producers and consumers.	DWARF 3.0. http://dwarfstd.org/Dwarf3Std.php
The *Run-time ABI for the Arm Architecture* [RTABI32] is summarized in Run-time ABI for the Arm architecture. It specified a helper-function ABI to support C, C++, and arithmetic (floating-point, integer division, and non-trivial long long arithmetic).	The unix `ar` format is the base standard for libraries of relocatable ELF files (see A note about ar format).
The *C Library ABI for the Arm Architecture* [CLIBABI32] is summarized in The C library ABI for the Arm architecture. It describes an ANSI C library ABI that can easily be supported by existing libraries.	ISO/IEC 9899:1990 Programming languages - C, with some reference to ISO/IEC 9899:1999. See also A note about ar format re `ar` format

Copyright © 2003, 2005, 2007, 2008 2018, 2020-2025, Arm Limited and its affiliates. All rights reserved.

Component standard	Base standard
The *Base Platform ABI for the Arm Architecture* [BPABI32] is summarized in The base platform ABI for the Arm architecture. It specified executable and shared object files suited to the execution environments supported by this ABI, and the static linker functionality required to create them.	The generic ELF standard (SVr4 GABI), 17th December 2003 draft. http://www.sco.com/developers/gabi/ Linux Standard Base v1.2 specification [GLSB]
Addenda to, and errata in, the ABI for the Arm Architecture [ADDENDA32] contains late additions to this version of the ABI specification, summarized in Addenda to and errata in the ABI for the Arm Architecture.	None.

The ABI for the Arm architecture base standard comprises the component standards listed in Documentation map of the ABI for the Arm architecture base standard. The scope and purpose of each component is explained in following subsections referred to from the table.

4.2 Procedure call standard for the Arm architecture

The Procedure Call Standard for the Arm architecture [AAPCS32] specifies:

- The size, alignment, and layout of C and C++ **Plain Old Data** (POD) types including
    - Primitive data types.
    - Structures.
    - Enumerated types.
    - Bit field types.
- Primitive types specific to C++ (references and pointers to members).
- How to pass control and data between publicly visible functions. A function is publicly visible if its callers are translated separately from it, and some callers might have no knowledge of how it was translated, other than that it conforms to the AAPCS.

  (When the public visibility of F is made explicit – for example by using a `#pragma` or annotation such as `__export` or `__declspec(dllexport)` – we also describe F as *exported*).
- Use of the run-time stack, and the stack invariants that must be preserved.

4.3 C++ ABI for the Arm architecture

The C++ ABI for the Arm architecture comprises four sub-components.

- The generic C++ ABI, summarized in The Generic C++ ABI, is the referenced base standard for this component.
- The C++ ABI supplement for the Arm architecture, summarized in The C++ ABI supplement for the Arm architecture, details Arm-specific deviations from the generic standard and records Arm-specific interpretations of it.
- The separately documented *Exception Handling ABI for the Arm Architecture*, summarized in The Exception handling ABI for the Arm architecture, describes the language-independent and C++-specific aspects of exception handling.
- The specimen implementations of the exception handling components, summarized in The exception handling components specimen implementation, include:
    - A language independent unwinder.
    - A C++ semantics module.

Copyright © 2003, 2005, 2007, 2008 2018, 2020-2025, Arm Limited and its affiliates. All rights reserved.

- Arm-specific C++ personality routines.

4.3.1 The Generic C++ ABI

The generic C++ ABI (originally developed for Itanium, [GCPPABI]) 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*, summarized in Procedure call standard for the Arm architecture).
- 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 represented as ELF symbols (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 *Exception Handling ABI for the Arm Architecture* (The Exception handling ABI for the Arm architecture).

4.3.2 The C++ ABI supplement for the Arm architecture

The Arm C++ ABI supplement is a major section in the document *C++ ABI for the Arm Architecture* [CPPABI32].

The Arm C++ ABI supplement describes where the C++ ABI for the Arm architecture necessarily diverges from the generic C++ ABI, because Itanium-specifics that cannot work (efficiently) for the Arm architecture show through an otherwise generic specification. For example, the generic encoding of a pointer to member function uses the least significant bit of a word to distinguish a code address from a v-table offset. The Arm architecture uses the same bit to distinguish Arm-code from Thumb-code, so the Arm ABI must deviate.

4.3.3 The Exception handling ABI for the Arm architecture

In common with the Itanium exception handling ABI, the *Exception Handling ABI for the Arm architecture* [EHABI32] specifies table-based stack unwinding that separates language-independent unwinding from language specific concerns. The Arm specification describes:

- The **base class** understood by the language-independent exception handling system, and its representation in object files. The language-independent exception handler only uses fields from this base class.
- A **derived class** used by Arm tools that efficiently encodes stack-unwinding instructions and compactly represents the data tables needed for handling C++ exceptions.
- The interface between the language-independent exception handling system and the **personality routines** specific to a particular implementation for a particular language. Personality routines interpret the language specific, derived class tables. Conceptually (though not literally, for reasons of implementation convenience and run-time efficiency), personality routines are member functions of the derived class.
- The interfaces between the (C++) language exception handling semantics module and
  - The language independent exception handling system.
  - The personality routines.
  - The (C++) application code (effectively the interface underlying throw).

Copyright © 2003, 2005, 2007, 2008 2018, 2020-2025, Arm Limited and its affiliates. All rights reserved.

The *Exception Handling ABI for the Arm Architecture* contains a significant amount of commentary to aid and support independent implementation of:

- Personality routines.
- The language-specific exception handling semantics module.
- Language independent exception handling.

This commentary does not provide, and is not intended to provide, a complete guide to independent implementation, but it does give a rationale for the interfaces to, and among, these components.

4.3.4 The exception handling components specimen implementation

Licence to use the exception handling components specimen implementation

The licence to use the specimen implementation of the exception handling components is included in the zip file containing them (as the file `LICENCE.txt`) and referred to from each source file.

Contents of the exception handling components example implementation

The exception handling components example implementation [EHEGI] comprises the following files.

- `cppsemantics.cpp` is a module that implements the semantics of C++ exception handling. It uses the language-independent unwinder (`unwinder.c`), and is used by the Arm-specific personality routines (`unwind_pr.[ch]`).
- `cxxabi.h` describes the generic C++ ABI (The Generic C++ ABI).
- `LICENCE.txt` contains your licence to use, copy, modify, and sublicense the specimen implementation.
- `unwind_env.h` is a header that describes the build and execution environments of the exception handling components. This header must be edited if the exception handling components are to be built with non-Arm compilers. This header #includes `cxxabi.h`.
- `unwind_pr.c` implements the three Arm-specific personality routines described in the *Exception Handling ABI for the Arm Architecture*.
- `unwinder.c` is an implementation of the language-independent unwinder.
- `unwinder.h` describes the interface to the language-independent unwinder, as described in the *Exception Handling ABI for the Arm Architecture*.

4.4 ELF for the Arm architecture

ELF for the Arm architecture comprises two components.

- The generic ELF specification, summarized in The generic ELF specification.
- The ELF processor supplement for the Arm architecture, summarized in ELF for the Arm architecture (processor supplement).

4.4.1 The generic ELF specification

The generic *Executable and Linking Format* specification was originally developed for Unix System V by AT&T. The latest version and the most recent stable drafts are published by *The SCO Group* at [GABI]. They specify:

- The format and meaning of statically linkable object files.
- The format and meaning of executable and shared-object files.

In each case, a supplement specifies processor-specific and platform-specific aspects.

Copyright © 2003, 2005, 2007, 2008 2018, 2020-2025, Arm Limited and its affiliates. All rights reserved.

- The enumeration of relocation directives is specific to a processor. Often, this is the only processor-specific facet of statically linkable (relocatable) ELF files.
- For executable files a platform-specific supplement specifies the interface to loading and dynamic linking.

4.4.2 ELF for the Arm architecture (processor supplement)

ELF for the Arm Architecture [AAELF32] describes the following:

- The representation in ELF and generation of cross-platform executable file information required by the *Base Platform ABI for the Arm Architecture* (The base platform ABI for the Arm architecture and [BPABI32]).
  - Symbol versioning information.
  - Symbol pre-emption information.
  - Procedure linkage table (PLT) entries, also known to users of the Arm architecture as intra-call veneers.
- The enumeration of static and dynamic relocation directives.
- Processor-specific flags and conventions (for example, the **mapping symbols** described in section 'Mapping symbols' of [AAELF32], used to accommodate the use of the Arm and Thumb instruction sets in the same code section).
- Two kinds of big-endian executable file (corresponding to the two flavors of big-endian code defined by Arm architecture v6 – in a BE8 big-endian executable file, code is nonetheless encoded little-endian).
- Miscellaneous Arm-specific executable and shared-object flags and section types used by the *ABI for the Arm Architecture*.

The Base Platform ABI for the Arm Architecture (The base platform ABI for the Arm architecture and [BPABI32]) specifies how ELF is used to support the executable file organizations and execution environments depicted in Schematic map of the ABI for the Arm Architecture.

4.5 DWARF for the Arm architecture

DWARF for the Arm architecture comprises two components.

- The generic DWARF specification, DWARF 3.0, summarized in DWARF 3.0.
- The Arm DWARF usage conventions, summarized in ABI DWARF usage conventions.

4.5.1 DWARF 3.0

DWARF 3.0 [GDWARF] makes precise many ambiguous and ill-defined aspects of the DWARF 2.0 specification, and extends that specification with:

- Additional constructs for describing optimized code and stack unwinding.
- Additional constructs for describing C++, Java, and Fortran 90.

4.5.2 ABI DWARF usage conventions

The ABI DWARF usage conventions are described in section 'Arm-specific DWARF definitions' of the document *DWARF for the Arm Architecture* [AADWARF32]. This section defines:

- An Arm-specific allocation of DWARF register numbers (in .debug_frame unwind descriptions).

Copyright © 2003, 2005, 2007, 2008 2018, 2020-2025, Arm Limited and its affiliates. All rights reserved.

- How Arm-state and Thumb-state are encoded in DWARF line number tables.
- How to describe data known to be in the other byte order (Arm architecture v6 access to other-endian data).
- The Canonical Frame Address (CFA).
- The default interpretation of debug frame Common Information Entries (CIEs).

4.6 Run-time ABI for the Arm architecture

The run-time helper-function ABI is described in the document *Run-time ABI for the Arm Architecture* [RTABI32].

The run-time helper-function ABI specifies how relocatable files produced by one toolchain must inter-operate with the run-time library from a different toolchain or execution environment. It gives a simple model of what a producer may assume of its output's eventual static linking and execution environments. It defines the following.

- A minimum model of floating-point arithmetic, based on the IEEE 754 floating-point arithmetic standard:
  - To which producers of relocatable files must conform.
  - Which producers of relocatable files can assume of the eventual execution environment.

  (The model sets a minimum standard. Implementers may implement the full IEEE 754 specification).
- The type signatures, meaning, and allowable names of the helper functions that all conforming static linking environments must support. The set of helper functions is divided into those required by C and assembly language, and those required only by C++.
- The provision, as part of the relocatable object itself or in separately delivered libraries, of all other helper functions used by a translation unit.

Libraries of relocatable ELF files must be formatted as Unix-style `ar` format linkable libraries (see A note about ar format, below).

4.7 The C library ABI for the Arm architecture

The C library ABI is described in the document *C Library ABI for the Arm Architecture* [CLIBABI32].

The C library ABI specifies:

- A binary interface to the C89 run-time library that allows a C-library-using function built by one toolchain to use the C library implementation provided by another.
- Constraints on language library headers necessary to allow toolchain X to use its own headers, or toolchain Y's headers, when building an object that must interface to toolchain Y's run-time library.

Compliance with this specification is a header-by-header **quality of implementation** issue. Compliance is not required in order to claim compliance to this base standard ABI for the Arm architecture.

Libraries of relocatable ELF files must be formatted as Unix-style `ar` format linkable libraries (see A note about ar format, below).

4.8 The base platform ABI for the Arm architecture

The base platform ABI is described in the document *Base Platform ABI for the Arm Architecture* [BPABI32].

The base platform ABI specifies:

Copyright © 2003, 2005, 2007, 2008 2018, 2020-2025, Arm Limited and its affiliates. All rights reserved.

- The content and format of ELF-based executable files suitable for post-processing to platform-specific binary formats appropriate to the families of execution environment supported by this ABI (Schematic map of the ABI for the Arm Architecture).
- The division of responsibility between static linkers generating fully symbolic executable ELF files and post-linkers generating less symbolic, platform-specific executable files.
- The static linking functionality needed to generate a generic executable file – the functionality needed to encompass the platform families supported by this ABI.

In most cases, some platform-specific post-processing is required to produce a platform executable file, but the complexity of the post processor is limited:

- For the SVr4 platform family, the required post-processing is a tiny increment on the static linking needed to generate a BPABI executable file. We expect most static linkers will offer an option to directly generate an executable file for Linux.
- For the DLL-based platform families platform-specific post-linking is significant, but little more complicated than an off-line version of SVr4 dynamic linking followed by a file format conversion.
- The bare metal platform family may demand additional static linking functionality to manage separate load and execution addresses and multiple image segments. Extracting such segments from an ELF executable file to drive ROM generating tools is trivial in comparison with the above tasks.

We expect post linking to be used primary in support of DLL-based platforms and specialized execution environments that feature dynamically loaded executable files.

4.9 A note about ar format

This ABI specifies that libraries of relocatable ELF files must be formatted as Unix-style ar format linkable libraries. This section specifies the ar variant used by Arm tools.

Unfortunately, ar format is not well standardized, and good public references to the format are hard to find. The ar command is deprecated from the Linux base standard [GLSB] which states that it is "... expected to disappear from a future version of the LSB".

A good general introduction to ar format, including a brief history and a warning about the incompatibility of its variants, is given in the *Manuals* section of [OpenBSD]. Search there for ar in section 5 – *File Formats*. However, please be aware of the following concerning the name field in archive headers:

- Different ar variants manage long file names (> 14 characters), and file names containing spaces, differently.
- *RealView* tools from Arm do *not* use the BSD file name conventions described at [OpenBSD].

Recently, we have found a *Wikipedia* article about ar format [http://en.wikipedia.org/wiki/Ar_(Unix)]. The *GNU variant* it describes is similar to the *RealView* variant summarized immediately below with this difference.

- As of early October 2008, this *Wikipedia* article claims that the 32-bit binary integers in the symbol table member (called '/') are encoded **big endian**.
- Arm targeted GNU tools and *RealView* tools always encode binary data using the byte order of the target system – little endian for little endian targets and big endian for big endian targets.

ar format conventions used by RealView tools and Arm-targeted GNU tools

File names recorded in archive member headers are terminated with a '/'. This allows short (≤ 14 characters) names to contain spaces.

The symbol table member (always present if an archive contains relocatable files) has the header name '/'. The symbol table member contains, in order:

- A 32-bit count of the number of symbols in the table. The byte order is that of the target system.

Copyright © 2003, 2005, 2007, 2008 2018, 2020-2025, Arm Limited and its affiliates. All rights reserved.

- For each symbol, the 32-bit offset within the archive of the header of the member defining it. The byte order is that of the target system.
- The NUL-terminated name of each symbol, listed in the same order as the offsets.

There is always a file names member with the header name '//'. It contains the names of all the files in the archive. Each name is terminated by '/' followed by '\n' (so the member contains only printable text).

If the file name of an archive member is longer than 14 characters, its header name is '/' followed by the decimal offset of its name in the file names member. Otherwise the header name is the file name of the member.

Ordinary members follow the symbol table member and the file names member.

4.10 Addenda to and errata in the ABI for the Arm Architecture

Addenda to, and errata in, the ABI for the Arm Architecture [ADDENDA32] contains late additions to version 2.0 (*this* version) and will contain any significant additions made during future maintenance of v2.0.

As of the publication of v2.0 of the *ABI for the Arm Architecture* (date shown at the top of this document), there are two addenda, in sections 'Build Attributes' and 'Thread Local Storage' of [ADDENDA32].

As of this publication date (shown at the top of this document) there are no errata.

4.10.1 Build attributes

Build attributes record:

- The use of architectural features and ABI variants by the code and data in a relocatable file.
- To a limited extent, the intentions of the builder of the file.

Attributes allow linkers to determine whether separately built relocatable files are inter-operable or incompatible, and to select the variant of a required library member that best matches the intentions of their builders.

4.10.2 Thread local storage

Thread Local Storage (TLS) is a class of **own data** (static storage) that – like the stack – is instanced once for each thread of execution.

This addendum defines the thread local storage (TLS) model for Linux for the Arm architecture. It covers:

- An introduction to the ABI issues raised by thread local storage.
- An introduction to addressing thread local variables.
- How Linux for the Arm architecture addresses thread local variables:
    - How thread local variables must be addressed from dynamically loadable DSOs.
    - How thread local variables may be addressed more efficiently from applications and DSOs loaded only when a process is created.

The Linux-specific TLS relocations are described in [AAELF32] (ELF for the Arm architecture).

Copyright © 2003, 2005, 2007, 2008 2018, 2020-2025, Arm Limited and its affiliates. All rights reserved.