



Application Binary Interface for the ARM[®] Architecture

The Base Standard

Document number: ARM IHI 0036B, current through ABI release 2.07
Date of Issue: 10th October 2008

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.

Keywords

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

THE TERMS OF YOUR ROYALTY FREE LIMITED LICENCE TO USE THIS ABI SPECIFICATION ARE GIVEN IN SECTION 1.4, **Your licence to use this specification** (ARM contract reference **LEC-ELA-00081 V2.0**). PLEASE READ THEM CAREFULLY.

BY DOWNLOADING OR OTHERWISE USING THIS SPECIFICATION, YOU AGREE TO BE BOUND BY ALL OF ITS TERMS. IF YOU DO NOT AGREE TO THIS, DO NOT DOWNLOAD OR USE THIS SPECIFICATION.

THIS ABI SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES (SEE SECTION 1.4 FOR DETAILS).

Proprietary notice

ARM, Thumb, RealView, ARM7TDMI and ARM9TDMI are registered trademarks of ARM Limited. The ARM logo is a trademark of ARM Limited. ARM9, ARM926EJ-S, ARM946E-S, ARM1136J-S, ARM1156T2F-S, ARM1176JZ-S, Cortex, and Neon are trademarks of ARM Limited. All other products or services mentioned herein may be trademarks of their respective owners.

Contents

1	ABOUT THIS DOCUMENT	3
1.1	Change control	3
1.1.1	Current status and anticipated changes	3
1.1.2	Change history	3
1.2	References	3
1.3	Terms and abbreviations	4
1.4	Your licence to use this specification	4
1.5	Acknowledgements	5
2	SCHEMATIC MAP OF THE ABI FOR THE ARM ARCHITECTURE	6
2.1	Notes about the schematic map	7
3	THE ABI FOR THE ARM ARCHITECTURE (BASE STANDARD)	8
3.1	Overview and documentation map	8
3.2	Procedure call standard for the ARM architecture	9
3.3	C++ ABI for the ARM architecture	9
3.3.1	The Generic C++ ABI	9
3.3.2	The C++ ABI supplement for the ARM architecture	10
3.3.3	The Exception handling ABI for the ARM architecture	10
3.3.4	The exception handling components specimen implementation	10
3.4	ELF for the ARM architecture	11
3.4.1	The generic ELF specification	11
3.4.2	ELF for the ARM architecture (processor supplement)	11
3.5	DWARF for the ARM architecture	12
3.5.1	DWARF 3.0	12
3.5.2	ABI DWARF usage conventions	12
3.6	Run-time ABI for the ARM architecture	12
3.7	The C library ABI for the ARM architecture	13
3.8	The base platform ABI for the ARM architecture	13
3.9	A note about ar format	13
3.10	Addenda to and errata in the ABI for the ARM Architecture	14
3.10.1	Build attributes	14
3.10.2	Thread local storage	15

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 and clarifications.

1.1.2 Change history

Issue	Date	By	Change
1.0	30 th October 2003	LS	First public release.
2.0 / A	24 th March 2005 24 th October 2007	LS	Second public release. Document renumbered (formerly GENC-003535 v2.0).
B	10 th October 2008	LS	§3.9 fixed a typo and updated the reference to ar format.

1.2 References

This document refers to the following documents.

Ref	External URL	Title
AADWARE		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		Base Platform ABI for the ARM Architecture
BSABI	<i>This document</i>	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 components, example implementations
GC++ABI	http://www.codesourcery.com/cxx-abi/abi.html	Generic C++ ABI
GDWARF	http://dwarf.freestandards.org/Dwarf3Std.php	DWARF 3.0, the generic debug format.
GABI	http://www.sco.com/developers/gabi/ ...	Generic ELF, 17 th December 2003 draft.
GLSB	http://www.linuxbase.org/spec/refspecs/ ...	gLSB v1.2 Linux Standard Base
Open BSD	http://www.openbsd.org/	Open BSD standard

Ref	External URL	Title
RTABI		Run-time ABI for the ARM Architecture

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: <ol style="list-style-type: none"> 1. The specifications to which an executable must conform in order to execute in a specific execution environment. For example, the <i>Linux ABI for the ARM Architecture</i>. 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 <i>C++ ABI for the ARM Architecture</i>, the <i>Run-time ABI for the ARM Architecture</i>, the <i>C Library ABI for the ARM Architecture</i>.
AEABI	(Embedded) ABI for the ARM architecture (<i>this</i> 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 <i>free standing</i>), 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

1.4 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.

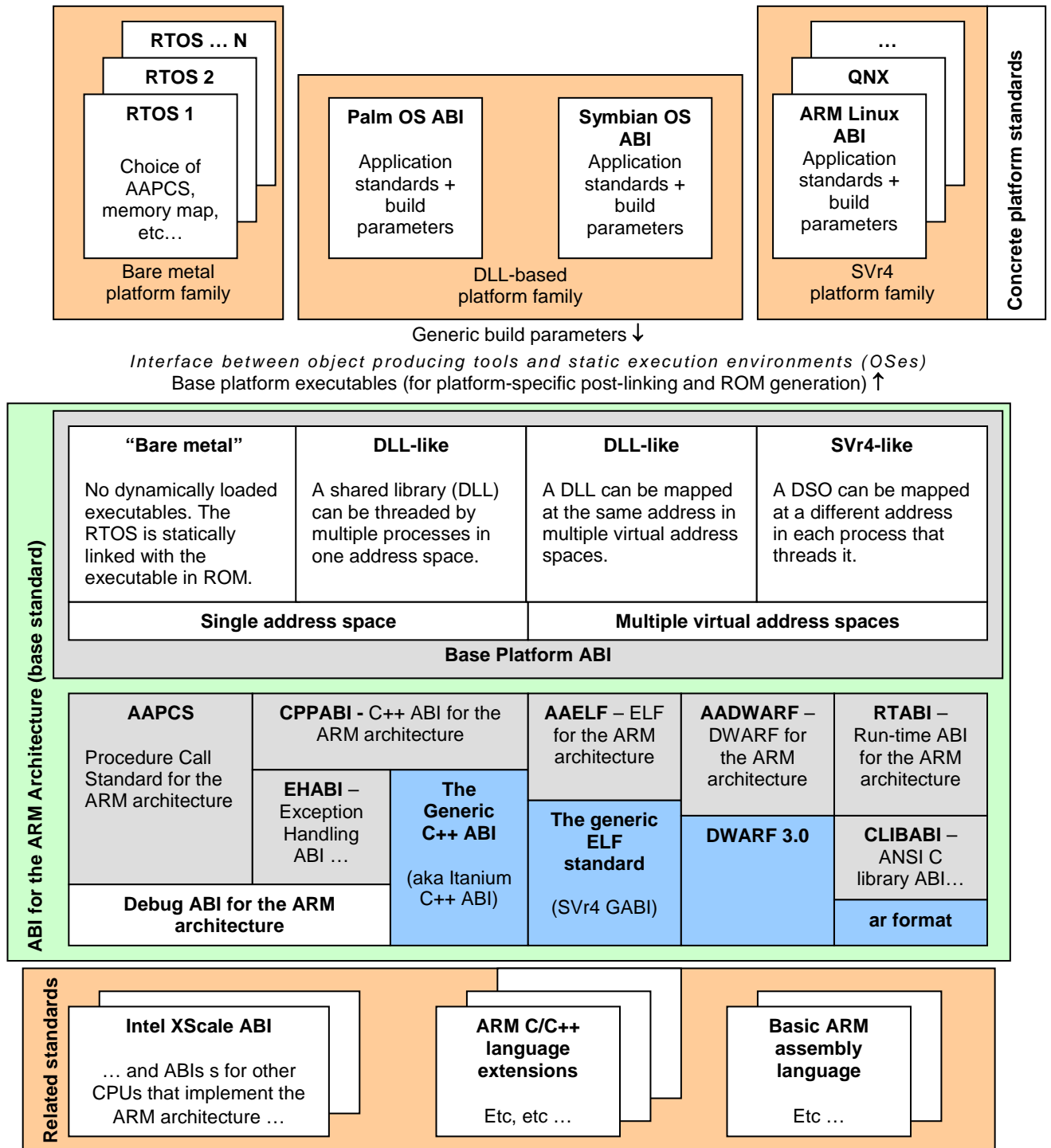
ARM Contract reference LEC-ELA-00081 V2.0 AB/LS (9 March 2005)

1.5 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.

2 SCHEMATIC MAP OF THE ABI FOR THE ARM ARCHITECTURE

Figure 1, A schematic map of the ABI for the ARM Architecture and some related standards



2.1 Notes about the schematic map

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

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.

Section 3, below, describes the base standard in detail and refers to each of its components.

3 THE ABI FOR THE ARM ARCHITECTURE (BASE STANDARD)

3.1 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.

Table 1, Documentation map of the ABI for the ARM architecture base standard

Component standard	Base standard
The <i>Procedure Call Standard for the ARM Architecture</i> [AAPCS], is summarized in §3.2.	None.
The <i>C++ ABI for the ARM Architecture</i> [CPPABI] is summarized in §3.3. It details where the C++ ABI for the ARM architecture deviates from the base standard.	The Generic C++ ABI (aka <i>C++ ABI for Itanium</i>). http://www.codesourcery.com/cxx-abi/abi.html .
The <i>Exception Handling ABI for the ARM Architecture</i> [EHABI], is summarized in §3.3.3. It describes C++-specific and language-independent exception processing.	
<i>ELF for the ARM Architecture</i> [AAELF] is summarized in §3.4. It gives processor-specific and platform-specific details not given in the generic ELF specification.	The generic ELF standard (SVr4 GABI), 17 th December 2003 draft. http://www.sco.com/developers/gabi/ ...
<i>DWARF for the ARM Architecture</i> [AADWARF] is summarized in §3.5. It describes how DWARF should be used to promote inter-operation between independent producers and consumers.	DWARF 3.0. http://dwarf.freestandards.org/Dwarf3Std.php
The <i>Run-time ABI for the ARM Architecture</i> [RTABI] is summarized in §3.6. It specifies 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 §3.9).
The <i>C Library ABI for the ARM Architecture</i> [CLIBABI], is summarized in §3.7 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 §3.9 re ar format.
The <i>Base Platform ABI for the ARM Architecture</i> [BPABI], is summarized in §3.8. It specifies 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), 17 th December 2003 draft. http://www.sco.com/developers/gabi/ ... Linux Standard Base v1.2 specification [GLSB].
<i>Addenda to, and errata in, the ABI for the ARM Architecture</i> [ADDENDA] contains late additions to this version of the ABI specification, summarized in §3.10.	None.

The ABI for the ARM architecture base standard comprises the component standards listed in Table 1. The scope and purpose of each component is explained in following subsections referred to from the table.

3.2 Procedure call standard for the ARM architecture

The Procedure Call Standard for the ARM architecture [AAPCS] 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.

3.3 C++ ABI for the ARM architecture

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

- The generic C++ ABI, summarized in §3.3.1, is the referenced base standard for this component.
- The C++ ABI supplement for the ARM architecture, summarized in §3.3.2, 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 §3.3.3, describes the language-independent and C++-specific aspects of exception handling.
- The specimen implementations of the exception handling components, summarized in §3.3.4, include:
 - A language independent unwinder.
 - A C++ semantics module.
 - ARM-specific C++ personality routines.

3.3.1 The Generic C++ ABI

The generic C++ ABI (originally developed for Itanium, [GC++ABI]) 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 §3.2).
- 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* (§3.3.3).

3.3.2 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* [CPPABI].

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.

3.3.3 The Exception handling ABI for the ARM architecture

In common with the Itanium exception handling ABI, the *Exception Handling ABI for the ARM architecture* [EHABI] 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*).

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.

3.3.4 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. It is broadly similar in scope and intent to the licence to use this specification displayed in §1.4 of this document.

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 (§3.3.1).
- **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*.

3.4 ELF for the ARM architecture

ELF for the ARM architecture comprises two components.

- The generic ELF specification, summarized in §3.4.1.
- The ELF processor supplement for the ARM architecture, summarized in §3.4.2.

3.4.1 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.

- 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.

3.4.2 ELF for the ARM architecture (processor supplement)

ELF for the ARM Architecture [AAELF] 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* (§3.8 and [BPABI]).
 - 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 §4.5 of [AAELF], 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* (§3.8 and [BPABI]) specifies how ELF is used to support the executable file organizations and execution environments depicted in Figure 1.

3.5 DWARF for the ARM architecture

DWARF for the ARM architecture comprises two components.

- The generic DWARF specification, DWARF 3.0, summarized in §3.5.1.
- The ARM DWARF usage conventions, summarized in §3.5.2.

3.5.1 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.

3.5.2 ABI DWARF usage conventions

The ABI DWARF usage conventions are described in section 3 of the document *DWARF for the ARM Architecture* [AADWARF]. This section defines:

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

3.6 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* [RTABI].

The run-time helper-function ABI specifies how relocatable files produced by one tool chain must inter-operate with the run-time library from a different tool chain 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 §3.9, below).

3.7 The C library ABI for the ARM architecture

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

The C library ABI specifies:

- A binary interface to the C89 run-time library that allows a C-library-using function built by one tool chain to use the C library implementation provided by another.
- Constraints on language library headers necessary to allow tool chain X to use its own headers, or tool chain Y's headers, when building an object that must interface to tool chain 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 §3.9, below).

3.8 The base platform ABI for the ARM architecture

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

The base platform ABI specifies:

- 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 (Figure 1).
- 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.

3.9 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 [Open BSD]. 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 [Open BSD].

Recently, we have found a *Wikipedia* article about **ar** format [[http://en.wikipedia.org/wiki/Ar_\(Unix\)](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.
- 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.

3.10 Addenda to and errata in the ABI for the ARM Architecture

Addenda to, and errata in, the ABI for the ARM Architecture [ADDENDA] 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 on page 1 of this document), there are two addenda, *Build Attributes* and *Thread Local Storage*.

As of this publication date (shown on page 1 of this document) there are no errata.

3.10.1 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.

3.10.2 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 [AAELF] (§3.4).