Addenda to, and Errata in, the ABI for the ARM® Architecture

Document number: ARM IHI 0045A, current through ABI release 2.06
Date of Issue: 13th November 2007

Abstract

This document describes late additions (addenda) to the ABI for the ARM Architecture version 2.0, and errors (errata) discovered in it after publication.

Keywords

Addenda to the ABI for the ARM Architecture, errata in the ABI for the ARM Architecture

Latest release of this specification

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).

Licence

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 and clarifications to any component of the *ABI for the ARM Architecture* [BSABI].
- Compatible extensions to any component of the *ABI for the ARM Architecture* [BSABI].

1.1.2 Change history

Issue	Date	By	Change
v1.0 r2.0	24th March 2005	LS	First public release.
v1.01 r2.01	4th July 2005	LS	Added new §2.2.6, (attribute) *Extensibility and compatibility*. Noted component errata and omissions (§5.2, §5.5, and §5.6).
v1.02	7th October 2005	LS	Added WMMX v2 architecture, TAG_CPU_unaligned_access (§2.3); changed R_ARM_PC24 to R_ARM_CALL (§3.3.1); added list of reserved name-space prefixes (§4); noted errata and omissions (§5.1, §5.3, §5.5, §5.7, and §5.8)
v1.03 r2.02	13th October 2005	LS	Minor typographical fixes.
v1.04 r2.03	6th January 2006	LS	Noted errata and omissions (§5.2.1, §5.2.3, and §5.6.1).
v1.05 r2.04	8th May 2006	LS	Added missing Tag_VFP_arch value for VFPv3 (§2.3.5). Noted errata and omissions (§5.1.2, §5.2.3, §5.3.1, §5.5.2, and §5.6.1).
v1.06 r2.05	18th January 2007	LS	Major clarification of, and some compatible extension to, §2, *Build Attributes*.
v1.07 r2.06	23rd October 2007	LS	Added: CPU_arch values for v6S-M and v6-M; and VFP_arch value for VFPv3-D16; added Tag_nodefaults, Tag_ABI_FP_16bit_format, and Tag_VFP_HP_extension. Rewrote §2, *Build Attributes*. Noted errata and omissions.
A	25th October 2007	LS	Document renumbered (formerly GENC-005895 v1.07).
A	13th November 2007	LS	Minor corrections to §5, *ERRATA AND MINOR ADDENDA*.

1.2 References

This document refers to the following documents.

Ref	Status / External URL	Title
AAELF		ELF for the ARM Architecture
AAPCS		Procedure Call Standard for the ARM Architecture

Ref	Status / External URL	Title
BSABI		ABI for the ARM Architecture (Base Standard).
EHABI		Exception Handling ABI for the ARM Architecture
RTABI		Run-time ABI for the ARM Architecture
ARM ARM	ARM DDI 0100E, ISBN 0 201 737191 (Also from http://infocenter.arm.com/help/index.jsp as the *ARMv5 Architecture Reference Manual*)	The ARM Architecture Reference Manual, 2nd edition, edited by David Seal, published by Addison-Wesley.
	ARM DDI 0406 (Subject to licence terms; please apply to ARM)	ARM Architecture Reference Manual ARM v7-A and ARM v7-R edition
GDWARF	http://dwarf.freestandards.org/Dwarf3Std.php	DWARF 3.0, the generic debug table format.
GELF	http://www.sco.com/developers/gabi/ …	Generic ELF, 17th December 2003 draft.

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 Vector 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 IMMEDITATELY 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)

2 ADDENDUM: BUILD ATTRIBUTES

2.1 Introduction to build attributes

2.1.1 About build attributes and compatibility

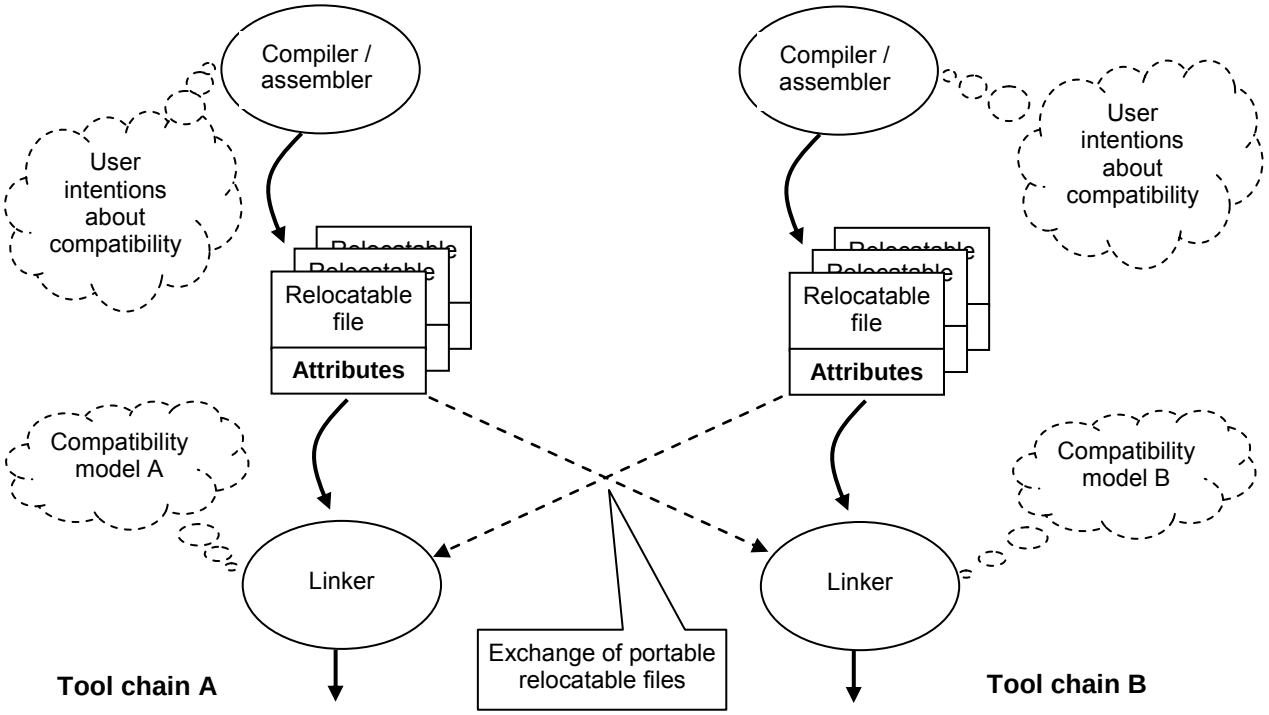
Build attributes record data that a linker needs to reason mechanically about the compatibility, or incompatibility, of a set of relocatable files. Other tools that consume relocatable files may find the data useful.

Build attributes are designed to have long-term invariant meaning. They record choices to which there is long term public commitment through the *ARM Architecture Reference Manual* [ARM ARM], the *ABI for the ARM Architecture* (of which this document is a component), vendor data sheets, and similar long lived publications.

Build attributes approximate the *intentions* the user of a compiler or assembler has for the compatibility of the relocatable file produced by the compiler or assembler (§2.1.1.1).

Figure 2-1, below, depicts the software development flows in which build attributes are important.

Figure 2-1, software development flows supported by build attributes



In this depiction there are two principal uses of build attributes.

- Within a tool chain, build attributes generate rich opportunities for a linker to diagnose incompatibility, enforce compatibility, and select library members intelligently according to its compatibility model.
- Between tool chains, build attributes describe the intended compatibility of a relocatable file and the entities it defines in terms independent of either tool chain, promoting safe exchange between tool chains of portable code in binary form.

2.1.1.1 Attribute values are based on user intentions

We base attribute values on user intentions to avoid the values being an unpredictable (effectively random) function of a compiler's code generation algorithms and to support using attributes with assembly language without overburdening programmers. Where attributes support exchanging portable relocatable files among tool chains, predictability is worth more than precision.

Capturing a user's compile-time *intentions* about compatibility is also important at link time. For example:

- A user might permit a compiler to use both the ARM ISA and the Thumb ISA.

  (The user intends the code to be usable on any processor that has both the ARM ISA and the Thumb ISA).

- The compiler might *choose* to use only the Thumb ISA in a specific build of a source file.

  Nonetheless, the relocatable file should be tagged as having been *permitted* to use the ARM ISA so that a linker can later link it with ARM-state library code and generate ARM-state intra-call veneers if that gives benefit to the executable file.

On the other hand, if the user intends code to be executed by both ARM7TDMI and Cortex-M3, the compiler must be constrained to generate only Thumb v1 instructions and the relocatable file should be tagged as *not permitted* to use the ARM ISA.

2.1.1.2 Capturing user intentions about compatibility

This standard does not specify *how* a tool should capture and approximate the intentions of its users.

As far as possible, ABI-defined compatibility tags (§2.3) model the long-term compatibility commitments implicit in architectural specifications, product data sheets, and the *ABI for the ARM Architecture*.

In general, tools have invocation options – command-line options and GUI configuration options – that present choices similar to those revealed in such documentation and *modeled* by ABI-defined compatibility tags.

The challenge for a tool that generates relocatable files is to select the set of build attributes – giving a value to each compatibility tag – that best approximates the user's intentions implicit in its invocation options.

This part of the problem of managing compatibility does not have a perfect solution. A user's intentions are imperfectly approximated by invocation options that are then sometimes imperfectly mapped to build attributes.

2.1.1.3 No standard compatibility model

This specification standardizes the meaning of build attributes, not the compatibility models within which they will be interpreted.

For the majority of build attributes there is only one reasonable interpretation of compatibility among their values, and it is an obvious one.

For a minority – mostly associated with procedure-call compatibility between functions – this is not the case and it is reasonable for different tool chains to take different positions according to the markets they serve.

Thus it is entirely reasonable that a relocatable file produced by tool chain A and accepted by tool chain B's linker might be rejected by tool chain C's linker when targeting exactly the same environment as tool chain B.

Our hope and intention for attributes is that they might prevent C's linker from accepting the output of A and silently generating a non functioning executable file or failing in a mysterious and difficult to explain manner.

2.1.2 The kinds of compatibility modeled by build attributes

Build attributes primarily model two kinds of compatibility.

- The compatibility of binary code with target hardware conforming to a revision of the ARM Architecture.
- The procedure-call compatibility between functions conforming to variants of this ABI.

The intuitive notion of compatibility can be given a mathematically precise definition using sets of demands placed on an execution environment.

For example, a program could be defined to be compatible with a processor if (and only if) the set of instructions the program might try to execute is a subset of the set of instructions implemented by the processor.

Target-related attributes (§2.3.5) describe the hardware-related demands a relocatable file will place on an execution environment through being included in an executable file for that environment.

For example, target-related attributes record whether use of the Thumb ISA is permitted, and at what architectural revision use is permitted. A pair of values for these attributes describes the set of Thumb instructions that code is permitted to execute and that the target processor must implement.

Procedure call-related attributes (§2.3.6) describe features of the ABI contract that the ABI allows to vary, such as whether floating point parameters are passed in floating point registers, the size of wchar_t, whether enumerated values are containerized according to their size, and so on.

We can also understand procedure call-related compatibility in terms of sets of demands placed on an execution environment, but the modeling is more difficult. In this case the *environment* is less obvious, more abstract, and elements of it can depend on an operating system or the tool chain itself.

Mathematically, A *compatible with* B can be understood as: {demands made by A} $\subseteq$ {demands made by B}.

Making this concrete sometimes requires combining information from several tags. Writing {T16@v4T} to denote the set of 16-bit Thumb instructions a processor must execute to conform to architecture version v4T, we can understand T16@v4T *compatible with* T16@v5TE as {T16@v4T} $\subset$ {T16@v5TE}.

2.1.3 The scope of build attributes

Unless #pragma or other mechanisms specific to a tool chain are used, it is usual for all parts of a relocatable file to be built the same way with the same compatibility intentions. So, usually, build attributes are given file scope and apply to all entities defined in the file to which they can apply. For example:

- File-scope attributes that model the compatibility of binary instructions with processors naturally apply to each instruction in every code-containing section in the file.

  (They obviously do *not* apply to sections that contain no code, nor to the data – such as literal pools – embedded in code sections).
- Attributes that describe the procedure call-compatibility of functions naturally apply to every function symbol defined in the code sections contained within the file.

Build attributes can be given to individual entities defined in a relocatable file.

- To an individual ELF section.

  These attributes also apply to every (relevant) symbol defined in the section.
- To an individual function or data object identified by an ELF symbol definition.

For the public build attributes defined by this ABI (§2.3) a compiler for C/C++ can only need to use section and symbol attributes in the presence of #pragmas, language extensions, or other mechanisms that affect the compatibility of individual entities within a binary file.

Linkers that support relocatable file merging – termed *partial linking* by *RealView* linkers – will, in general, need to transfer the file scope attributes of an input file to the individual sections that file contributes to the output file.

This standard places no requirements on the presence or absence of build attributes in executable files.

2.1.4 Build attributes and conformance to the ABI

In revision 2.05 and later revisions of this ABI specification, the presence of a build attributes section containing an "aeabi" subsection containing a *conformance tag* (§2.3.7.4) denotes an *explicit* claim by the producer that the relocatable file conforms to:

- The version of the ABI described by tag's string parameter.
- The variant of that version described by the public ("aeabi") build attributes.

A claim to conform to ABI version "0" denotes that no unconditional claim to conform is being made. Generic compatibility tags (§2.3.7.2) may then describe limited or conditional claims to conform.

In revisions of this specification before r2.05 any claim to conform to this ABI was implicit, and the version to which conformance was (possibly) claimed was implicitly "2".

2.1.5 Combining attribute values

Suppose E1 and E2 are entities (for example, relocatable files) with attribute values a1 and a2 for an attribute tag T. This section discusses how to generate the correct value of T for the entity formed by combining E1 and E2 (for example, the executable file formed by linking E1 with E2)

In each case, the values of a tag can be partially ordered according to the sets of demands they represent. We shall write a1 ≤ a2 if an entity tagged with <T, a1> makes no more demands on its environment than an entity tagged with <T, a2>.

Writing {T:a1} to denote the set of demands made by an entity tagged with <T, a1>, we can define a1 ≤ a2 if {T:a1} ⊆ {T:a2}. (A set of demands might be the set of instructions a processor must execute, for example).

Informally we say that *a1 is compatible with a2* or *a1 is more compatible than a2* when a1 ≤ a2.

Using `Tag_CPU_arch` (§2.3.5.2) as an example, v4T ≤ v5TE, because the set of instructions conforming to architecture v4T is a subset of the set conforming to architecture v5TE. Stated more precisely, for both the ARM ISA and the 16-bit Thumb ISA, it is the case that {*ISA*@v4T} ⊆ {*ISA*@v5TE}.

This partial order often differs from the arithmetic order of the enumerated values though in many cases it

- Is identical (as with `Tag_THUMB_ISA_use` in §2.3.5.2).
- Is reversed (as with `Tag_ABI_align8_preserved` in §2.3.6.2).
- Represents mutually incompatible choices with which only the identical choice, or no use at all, is compatible (as with `Tag_ABI_PCS_wchar_t` in §2.3.6.2).

Note that the appropriate partial order to use can evolve over time as the underlying specifications evolve.

2.1.5.1 Combining two values of the same tag

We shall write a1 <> a2 if a1 and a2 are *unordered* in the partial order of demands/compatibility and a1 + a2 to denote the combination of a1 and a2.There are the following combination rules.

- If a1 ≤ a2, a1 + a2 = a2. Similarly, if a2 ≤ a1, a1 + a2 = a1. ('+' behaves like the *maximum* function).
- If a1 <> a2 there are two mutually exclusive sub-cases.

- There is a least a3 such that a1 ≤ a3 and a2 ≤ a3. Then a1 + a2 = a3.
  Example: `Tag_CPU_arch` when a1 = v6KZ, a2 = v6T2, and a3 = v7.
- There is no such a3, so a1 + a2 denotes the attempted combination of incompatible values.
  Example: `Tag_ABI_PCS_wchar_t` when a1 = 2 and a2 = 4.

In this second sub-case it is a matter of notational taste whether a1 + a2 is defined to have a value such as *error* or *Top*, or defined to have no value. Either way, in practice we expect an attempted combination to fail in a way specific to a tool chain's compatibility model (for example by provoking a link-time diagnostic).

2.2 Representing build attributes in ELF files

2.2.1 Encoding

Encoding build attributes needs more than a few bits, so we encode them in an vendor-specific section of type SHT_ARM_ATTRIBUTES and name `.ARM.attributes` (for further details see [AAELF]).

An attribute is encoded in a <*tag*, *value*> pair.

Both tags and numerical values are encoded using unsigned LEB128 encoding (ULEB128), DWARF-3 style (for details see [GDWARF]).

The public tags and values specified by this version of the ABI encode identically as ULEB128 numbers and single byte numbers (all values are in the range 0-127).

String values are encoded using NUL-terminated byte strings (NTBS).

2.2.2 Structure of an ELF attributes section

An attributes section contains a sequence of subsections. Each one is either

- Defined by this ABI and public to all tools that process the file.
- Private to a tool vendor's tools.

The type of each subsection is given by a short textual (NTBS) name.

This ABI requires a vendor to register a short vendor name to use as a prefix to the names of private helper functions (for details see [RTABI] or [AAELF]). The same vendor name identifies attribute subsections private to that vendor's tools.

A public attributes subsection is named *aeabi.* Names beginning *Anon* and *anon* are reserved to unregistered private use.

2.2.3 Formal syntax of an ELF attributes section

The syntactic structure of an attributes section is:

```
<format-version: 'A'>
[ <uint32: subsection-length> NTBS: vendor-name
  <bytes: vendor-data>
]*
```

Format-version describes the format of the following data. It is a single byte. This is version 'A' (0x41). This field exists to allow future changes in format. We do not intend that there will be many versions.

Subsection-length is a 4-byte integer in the byte order of the ELF file. It encodes the length of the subsection, including the length field itself, the vendor name string and its terminating NUL byte, and the following vendor data. It gives the offset from the start of this subsection to the start of the next one.

Vendor-name is a NUL-terminated byte string (NTBS) like a C-language literal string.

No requirements are placed on the format of private vendor data. The format of a public attributes subsection ("aeabi" pseudo-vendor data) is described in §2.2.4.

We expect a *dot-ARM-dot-attributes* section in a relocatable file will most typically contain one vendor subsection from the "aeabi" pseudo-vendor and possibly one from the generating tool chain (e.g. "ARM", "gnu", "WRS", etc).

Formally, there are no constraints on the order or number of vendor subsections. A consumer can collect the public ("aeabi") attributes in a single pass over the section, then all of its private data in a second pass.

2.2.4 Formal syntax of a public ("aeabi") attributes subsection

The syntactic structure of a public attributes subsection is:

```
[   Tag_File    (=1) <uint32: byte-size> <attribute>*
  | Tag_Section (=2) <uint32: byte-size> <section number>* 0 <attribute>*
  | Tag_symbol  (=3) <unit32: byte-size> <symbol number>*  0 <attribute>*
]+
```

A public subsection contains any number of sub-subsections. Each records attributes relating to:

- The whole relocatable file.
  These sub-subsections contain just a list of attributes. They are identified by a leading `Tag_File` (=1) byte.
- A set of sections within the relocatable file.
  These sub-subsections contain a list of section numbers followed by a list of attributes. They are identified by a leading `Tag_Section` (=2) byte.
- A set of (defined) symbols in the relocatable file.
  These sub-subsections contain a list of symbol numbers followed by a list of attributes. They are identified by a leading `Tag_Symbol` (=3) byte.

In each case, *byte-size* is a 4-byte unsigned integer in the byte order of the ELF file. *Byte-size* includes the initial tag byte, the size field itself, and the sub-subsection content. That is, it is the byte offset from the start of this sub-subsection to the start of the next sub-subsection.

Both section indexes and defined symbol indexes are non-zero, so a NUL byte ends a string and a list of indexes without ambiguity.

There are no constraints on the order or number of sub-subsections in a public attributes subsection (but see remarks in §2.3.7.4 concerning `Tag_conformance` and §2.3.7.5 concerning `Tag_nodefaults`).

A consumer that needs the data in scope nesting order can obtain the file attributes, the section-related attributes, and the symbol-related attributes, by making three passes over the subsection.

2.2.5 Conformance constraints

This ABI requires the following of files that *claim to conform* (§2.1.4) to the ABI.

- Attributes that record data about the compatibility of this file with other files must be recorded in a public "aeabi" subsection.
- Attributes meaningful only to the producer must be recorded in a private vendor subsection. These must not affect compatibility between relocatable files.

When a producer does not *explicitly* claim compatibility to the ABI, it may nonetheless publicly describe the effect on compatibility of its private attributes by using generic compatibility tags (§2.3.7.2). These must record a safe approximation. The producer can record precise information that only its own tool chain comprehends in a private vendor subsection.

- We intend that another tool chain should not mistakenly link incompatible files. The price of safety is that a tool chain might sometimes diagnose incompatibility between files that could be safely linked, because their compatibility has been approximated.
- We do not expect that a tool chain should be able to comprehend the private data of a different tool chain (other than through private agreement among tool chains).

2.2.6 Coding extensibility and compatibility

As a specification like this evolves, legacy binary files and the tools that process them get out of step. This will cause difficulties when a tool must consume a public attributes subsection containing tags it does not comprehend (that is, when the tool follows an earlier version of the specification than the binary file).

The attributes defined in §2.3, below, carry a mix of information that consumers can safely ignore and information about incompatibility that must not be ignored. In the absence of further conventions, the only safe course for a tool to take on encountering an unknown tag would be to stop processing.

To support more graceful behavior in the face of an evolving set of public tags, we adopt these conventions.

- Tags 0-63 convey information that a consuming tool *must* comprehend. This includes all the tags (1-32) defined by the first release (v1.0) of this addendum. A tool encountering an unknown tag in this range should stop processing or take similar defensive action (Q-o-I).
- Tags 64-127 convey information a consumer can ignore *safely* (though maybe with degraded functionality).
- For $N \geq 128$, tag N has the same properties as tag N modulo 128.

To allow an ignored tag and its parameter value to be skipped easily, we adopt this convention.

- For N > 32, even numbered tags have a ULEB128 parameter and odd numbered ones have a null-terminated byte string (NTBS) parameter.
- A consumer must comprehend tags 1-32 individually.

2.3 Public ("aeabi") attribute tags

2.3.1 About public tags

A consumer must recognize the tags described in this section (§2.3) in vendor subsections named "aeabi".

Other vendor sections may re-use these tags, define their own tags, or use a completely different encoding of their private attribute data. In each case the meaning of the data is private to the defining vendor.

While public tags and their numerical parameters are specified as ULEB128-encoded, the values defined by this version, and all earlier versions, of the ABI encode identically as ULEB128 numbers and single byte numbers.

2.3.2 Default values for public tags

Unless `Tag_nodefaults` (§2.3.7.5) is used, the effect of omitting a public tag in file scope is identical to including it with a value of 0 or “”, depending on its parameter type.

A producer must consider what value it intends to give to each public tag. In the absence of `Tag_nodefaults` a consumer should conclude that a producer *intended* to give the value 0 to each omitted public tag.

2.3.3 Inheritance of default tag values

Unless `Tag_nodefaults` (§2.3.7.5) is used, an attribute acquires a default value of zero by omission of its tag only in file scope (defined by `Tag_File`, §2.2.4).

If `Tag_nodefaults` is used in file scope, an attribute acquires a default value of zero in section scope by omission of its tag in section scope. No section-specific or symbol-specific values are inherited from file scope.

A value given explicitly in section scope overrides a value given, or defaulted, in file scope.

If `Tag_nodefaults` is used in section scope, the section has no attributes of any kind, default or otherwise, and it is erroneous to give any other tag an explicit value in the section.

A value given explicitly in symbol scope overrides a value given, or defaulted, in section scope or file scope.

In any scope it is erroneous to give two different values to the same attribute, though the same value may be given more than once.

A producer should generate explicit per-section and per-symbol attribute data only when the data cannot be inferred through inheritance. Being explicit is verbose, and per-section and per-symbol build attributes should only be used to capture exceptions to per-file attribution.

2.3.4 How this specification describes public attributes

In the following sections we describe each attribute in a uniform style, as follows.

```
Tag_tag_name (=tag value), parameter type (=uleb128 or NTBS)
 value  Comment
 value  Comment
 ...
```

Block commentary about the tag and its possible values.

Tag value gives the numerical representation of the tag. It is a small integer less than 128.

Parameter type gives the type of the parameter that immediately follows the tag in the attributes section. It is either a ULEB128-encoded integer or a NUL-terminated byte string (NTBS).

Following lines enumerate the currently defined parameter values, giving a short comment about each one.

A block of explanatory text follows in some cases.

2.3.5 Target-related attributes

2.3.5.1 About target-related attributes

Target-related attributes describe the demands a user *permitted* a producer to place on the target system (§2.1.1.2).

These attributes summarize the target features and facilities needed to execute the instructions contained in the code sections of this relocatable file. A file may make fewer demands than its attributes describe, but not more.

2.3.5.2 The target-related attributes

```
Tag_CPU_raw_name, (=4), NTBS
```

The value "" denotes that the raw name is identical to the CPU name (described immediately below). It denotes that the user built for a generic implementation (such as ARM946E-S) rather than a vendor-specific part (such as ML692000). It is always safe to use "" as a dummy value for this tag, or to omit this tag.


```
Tag_CPU_name, (=5), NTBS
```

The string value is defined by ARM or the *architecture licensee* responsible for designing the part. It is the official product name, with no extension and no abbreviation.

An ARM-defined architecture name may be used instead of a CPU name, and denotes that the user had more generic intentions. ARM-defined names of CPUs and architectures recognized by RVCT 3.1 are listed in §2.4.

Using string-valued CPU names avoids central coordination and supports peer to peer agreement – for example, between an architecture licensee and a tools vendor.

The following two tags describe the processor architecture version and architecture profile for which the user intended the producer to generate code.

```
Tag_CPU_arch, (=6), uleb128
 0  Pre-v4
 1  ARM v4     // e.g. SA110
 2  ARM v4T    // e.g. ARM7TDMI
 3  ARM v5T    // e.g. ARM9TDMI
 4  ARM v5TE   // e.g. ARM946E-S
 5  ARM v5TEJ  // e.g. ARM926EJ-S
 6  ARM v6     // e.g. ARM1136J-S
 7  ARM v6KZ   // e.g. ARM1176JZ-S
 8  ARM v6T2   // e.g. ARM1156T2F-S
 9  ARM v6K    // e.g. ARM1136J-S
10  ARM v7     // e.g. Cortex A8
11  ARM v6-M   // e.g. Cortex M1
12  ARM v6S-M  // v6-M with the System extensions

Tag_CPU_arch_profile (=7), uleb128
 0  Architecture profile is not applicable (e.g. pre v7, or cross-profile code)
'A' (0x41) The application profile (e.g. for Cortex A8)
'R' (0x52) The real-time profile (e.g. for Cortex R4)
'M' (0x4D) The microcontroller profile (e.g. for Cortex M3)
'S' (0x53) Application or real-time profile (i.e. the 'classic' programmer's model)
```

`Tag_CPU_arch_profile` states that the attributed entity requires the noted architecture profile.

The value 0 states that there is no requirement for any specific architecture profile. The value 'S' denotes that the attributed entity requires the classic programmer's model rather than the microcontroller programmer's model.

Like architectures, instruction sets are long term stable concepts relevant to compatibility. The following tags track the permitted use of instruction sets. In each case, omitting the tag is identical to including it with a value of 0. The architecture revision (`Tag_CPU_arch`) implies the permitted subset of instructions from the permitted ISA.

```
Tag_ARM_ISA_use, (=8), uleb128
 0  The user did not permit this entity to use ARM instructions
 1  The user intended that this entity could use ARM instructions

Tag_THUMB_ISA_use, (=9), uleb128
 0  The user did not permit this entity to use Thumb instructions
 1  The user permitted this entity to use 16-bit Thumb instructions [**]
 2  32-bit Thumb instructions were permitted (implies 16-bit instructions permitted)
```

[**] In the absence of other 32-bit Thumb instructions, BL is considered to be two 16-bit instructions.

```
Tag_VFP_arch, (=10), uleb128
 0  The user did not permit this entity to use FP hardware
 1  The user permitted this entity to use VFPv1 instructions
 2  VFPv2 instructions were permitted (implies VFPv1 instructions were permitted)
 3  VFPv3 instructions were permitted (implies VFPv2 instructions were permitted)
 4  VFPv3 instructions were permitted, but only citing registers D0-D15, S0-S31

Tag_WMMX_arch, (=11), uleb128
 0  The user did not permit this entity to use WMMX
 1  The user permitted this entity to use WMMX v1
 2  The user permitted this entity to use WMMX v2

Tag_Advanced_SIMD_arch, (=12), uleb128
 0  The user did not permit this entity to use the Advanced SIMD Architecture (Neon)
 1  Use of the Advanced SIMD Architecture (Neon) was permitted

Tag_VFP_HP_extension (=36), uleb128
 0  The user did not permit this entity to use the VFPv3/Advanced SIMD optional
    half-precision extension
 1  Use of the VFPv3/Advanced SIMD optional half-precision extension was permitted
```

The following tag describes the unaligned data accesses the user permitted the producer to make.

```
Tag_CPU_unaligned_access, (=34), uleb128
 0  The user did not intend this entity to make unaligned data accesses
 1  The user intended that this entity might make v6-style unaligned data accesses
```

2.3.6 Procedure call-related attributes

2.3.6.1 About procedure call-related attributes

Procedure call-related attributes describe compatibility with the ABI. They summarize the features and facilities that must be agreed in an interface contract between functions defined in this relocatable file and elsewhere. We call the functions subject to such interface contracts *public functions*.

(**Aside**: In C/C++ terminology, all public functions have extern linkage but not all extern functions are public. In ELF terminology, all public symbols are global, but not all global symbols are public. **End aside**).

For a public function defined in this relocatable file, attributes describe what the user intended the producer to guarantee about the function.

For a public function used by code in this relocatable file, attributes describe what the user intended the producer to assume about the function.

2.3.6.2 The procedure call-related attributes

```
Tag_PCS_config, (=13), uleb128
 0  No standard configuration used, or no information recorded
 1  Bare platform configuration
 2  Linux application configuration
 3  Linux DSO configuration
 4  Palm OS 2004 configuration
 5  Reserved to future Palm OS configuration
 6  Symbian OS 2004 configuration
 7  Reserved to future Symbian OS configuration
```

`Tag_PCS_config` summarizes the user intention behind the procedure-call standard *configuration* used. Its value must be consistent with the values given to the tags below, and must not be used as a macro in place of them.

The following four tags summarize how the user intended the attributed entity to address static data.

```
Tag_ABI_PCS_R9_use, (=14), uleb128
 0  R9 used as V6 (just another callee-saved register, implied by omitting the tag)
 1  R9 used as SB, a global Static Base register
 2  R9 used as TLS pointer – (This use is not supported by version 2.0 of the ABI)
 3  R9 not used at all by code associated with the attributed entity
```

R9 has a role in some variants of the PCS. `Tag_ABI_PCS_R9_use` describes the user's chosen PCS variant.

```
Tag_ABI_PCS_RW_data, (=15), uleb128
 0  RW static data was permitted to be addressed absolutely
 1  RW static data was only permitted to be addressed PC-relative
 2  RW static data was only permitted to be addressed SB-relative
 3  The user did not permit this entity to use RW static data

Tag_ABI_PCS_RO_data, (=16), uleb128
 0  RO static data was permitted to be addressed absolutely
 1  RO static data was only permitted to be addressed PC-relative
 2  The user did not permit this entity to use RO static data

Tag_ABI_PCS_GOT_use, (=17), uleb128
 0  The user did not permit this entity to import static data
 1  The user permitted this entity to address imported data directly
 2  The user permitted this entity to address imported data indirectly (e.g. via a GOT)
```

Compatibility among shared objects and their clients is affected by whether imported data are addressed directly or indirectly. Linux imported data must be addressed indirectly (via the Global Object Table, or GOT). Symbian OS (2004) imported data must be addressed directly.

The following two tags describe the permitted sizes of a wide character and an enumerated data item.

```
Tag_ABI_PCS_wchar_t, (=18), uleb128
 0  The user prohibited the use of wchar_t when building this entity
 2  The user intended the size of wchar_t to be 2
 4  The user intended the size of wchar_t to be 4

Tag_ABI_enum_size, (=26), uleb128
 0  The user prohibited the use of enums when building this entity
 1  Enum values occupy the smallest container big enough to hold all their values
 2  The user intended Enum containers to be 32-bit
 3  The user intended that every enumeration visible across an ABI-complying interface
    contains a value needing 32 bits to encode it; other enums can be containerized
```

The following pair of tags summarizes the 8-byte alignment contract.

```
Tag_ABI_align8_needed, (=24), uleb128
 0  The user did not permit code to depend the alignment of 8-byte data
 1  Code was permitted to depend on the 8-byte alignment of 8-byte data items
 2  Code was permitted to depend on the 4-byte alignment of 8-byte data items

Tag_ABI_align8_preserved, (=25), uleb128
 0  The user did not require code to preserve 8-bye alignment of 8-byte data objects
 1  Code was required to preserve 8-byte alignment of 8-byte data objects
 2  Code was required to preserve 8-byte alignment of 8-byte data objects and to
    ensure (SP MOD 8) = 0 at all instruction boundaries (not just at function calls)
```

The following five tags summarize the requirements code associated with this attributed entity was permitted to place on floating-point arithmetic.

```
Tag_ABI_FP_rounding, (=19), uleb128
 0  The user intended this code to use the IEEE 754 round to nearest rounding mode
 1  The user permitted this code to choose the IEEE 754 rounding mode at run time

Tag_ABI_FP_denormal, (=20), uleb128
 0  The user built this code knowing that denormal numbers might be flushed to (+) zero
 1  The user permitted this code to depend on IEEE 754 denormal numbers
 2  The user permitted this code to depend on the sign of a flushed-to-zero number being
    preserved in the sign of 0

Tag_ABI_FP_exceptions, (=21), uleb128
 0  The user intended that this code should not check for inexact results
 1  The user this code to check the IEEE 754 inexact exception

Tag_ABI_FP_user_exceptions, (=22), uleb128
 0  The user intended that this code should not enable or use IEEE user exceptions
 1  The user permitted this code to enables and use IEEE 754 user exceptions

Tag_ABI_FP_number_model, (=23), uleb128
 0  The user intended that this code should not use floating point numbers
 1  The user permitted this code to use IEEE 754 format normal numbers only
 2  The user permitted numbers, infinities, and one quiet NaN (see [RTABI])
 3  The user permitted this code to use all the IEEE 754-defined FP encodings
```

FP model hierarchies are difficult to specify. In practice, there is a large lattice of potentially useful models, depending on whether FP arithmetic is done by software or by hardware, and on the properties of that hardware. The tags above allow requirements to be specified using independent features. For example, code following the Java numerical model should record `Tag_ABI_FP_denormal = 1` and `Tag_ABI_FP_number_model = 2`, while graphics code concerned with speed above all other considerations might record `Tag_ABI_FP_number_model = 1` and `Tag_ABI_FP_optimization_goals = 2` (see below).

The following tag summarizes use of 16-bit floating point numbers by the attributed entities.

```
Tag_ABI_FP_16bit_format (=38), uleb128
 0  The user intended that this entity should not use 16-bit floating point numbers
 1  Use of IEEE 754 (draft, November 2006) format 16-bit FP numbers was permitted
 2  Use of VFPv3/Advanced SIMD "alternative format" 16-bit FP numbers was permitted
```

Options 1 and 2 are mutually incompatible.

The next three tags record permitted use of VFP and WMMX co-processors. Note that:

- Under the *base variant* of the procedure call standard [AAPCS], FP parameters and results are passed the *soft FP* way, in core registers or on the stack. WMMX parameters and results are passed the same way.
- The *VFP variant* of [AAPCS] uses VFP registers D0-D7 (s0-s15) to pass parameters and results.
- The Intel WMMX convention is to use wR0-wR9 to pass parameters and results.

```
Tag_ABI_HardFP_use, (=27), uleb128
 0  The user intended that VFP use should be implied by Tag_VFP_arch
 1  The user permitted this entity to use only SP VFP instructions
 2  The user permitted this entity to use only DP VFP instructions
 3  The user permitted this entity to use both SP and DP VFP instructions
    (Note: This is effectively an explicit version of the default encoded by 0)

Tag_ABI_VFP_args, (=28), uleb128
 0  The user intended FP parameter/result passing to conform to AAPCS, base variant
 1  The user intended FP parameter/result passing to conform to AAPCS, VFP variant
 2  The user intended FP parameter/result passing to conform to tool chain-specific
    conventions
```

```
Tag_ABI_WMMX_args, (=29), uleb128
 0  The user intended WMMX parameter/result passing conform to the AAPCS, base variant
 1  The user intended WMMX parameter/result passing conform to Intel's WMMX conventions
 2  The user intended WMMX parameter/result passing conforms to tool chain-specific
    conventions
```

2.3.7 Miscellaneous attributes

2.3.7.1 Optimization attributes

The final pair of ABI-related tags record optimization goals. These are not required for reasoning about incompatibility, but assist with selecting appropriate variants of library members.

```
Tag_ABI_optimization_goals, (=30), uleb128
 0  No particular optimization goals, or no information recorded
 1  Optimized for speed, but small size and good debug illusion preserved
 2  Optimized aggressively for speed, small size and debug illusion sacrificed
 3  Optimized for small size, but speed and debugging illusion preserved
 4  Optimized aggressively for small size, speed and debug illusion sacrificed
 5  Optimized for good debugging, but speed and small size preserved
 6  Optimized for best debugging illusion, speed and small size sacrificed
```

With `Tag_ABI_optimization_goals` we capture one of three potentially conflicting intentions – high performance, small size, and easy debugging – at one of two levels.

At the first level the goal is unambiguous, but pursuit of it is constrained. The conflicting goals still matter, but less than the primary goal.

At the second level, the conflicting goals are insignificant in comparison to the primary goal. It is difficult to capture optimization intentions precisely, but to a significant degree what matters to a tool chain is the user's goal (speed, small size, or debug-ability), and whether or not the user is willing to sacrifice all other considerations to achieving that goal.

```
Tag_ABI_FP_optimization_goals, (=31), uleb8
 0  No particular FP optimization goals, or no information recorded
 1  Optimized for speed, but small size and good accuracy preserved
 2  Optimized aggressively for speed, small size and accuracy sacrificed
 3  Optimized for small size, but speed and accuracy preserved
 4  Optimized aggressively for small size, speed and accuracy sacrificed
 5  Optimized for accuracy, but speed and small size preserved
 6  Optimized for best accuracy, speed and small size sacrificed
```

With `Tag_ABI_FP_optimization_goals` we also capture one of three potentially conflicting goals at one of the same two levels as `Tag_ABI_optimization_goals` captures.

Some accuracy sacrificed is intended to allow, for example, re-association of expressions in generated code. In library code it is intended to permit the assumption that value ranges will be *reasonable*. In particular, binary to decimal and decimal to binary conversion may meet only the minimum standards specified by IEEE 754, and range reduction for trigonometric functions should be assumed to be naive.

2.3.7.2 Generic compatibility tag

The generic compatibility tag describes the limited compatibility this file might offer when no strong claim to conform to the ABI (§2.1.4) has been made using `Tag_conformance` (§2.3.7.4).

```
Tag_compatibility (=32), uleb128: flag, NTBS: vendor-name
```

An omitted tag implies flag = 0, vendor-name = "". An explicit flag value is not 0 and can be considered to be the first byte(s) of the vendor name for the purpose of skipping the entry. The default value of 0 describes the implicit claim made by files generated prior to v1.06 of this specification.

The defined flag values and their meanings are as follows.

```
0   The tagged entity has no tool chain-specific requirements
    (and no vendor tag hides an ABI incompatibility)
1   This entity can conform to the ABI if processed by the named tool chain
    The ABI variant to which it conforms is described solely by public “aeabi” tags
>1  The tagged entity does not conform to the ABI but it can be processed by other
    tools under a private arrangement described by flag and vendor-name
```

A *flag* value >1 identifies an arrangement, beyond the scope of the ABI, defined by the named vendor. A tool chain that recognizes the arrangement might successfully process this file. Note that a producer must use the name of the vendor defining the arrangement, not the name of the producing tool chain.

(Versions of this specification through v1.05 stated:

```
>1 The tagged entity is compatible only with identically tagged entities,
   and entities not tagged by this tool chain
```

The underlined part of that definition was a mistake that makes the definition useless. With the underlined part removed, the old definition is effectively compatible with, but more restrictive than, the new one).

2.3.7.3 Secondary compatibility tag

`Tag_CPU_arch` conceals a difficulty in relation to the 16-bit Thumb ISA: there is no way to tag an entity as compatible with both ARM7TDMI (architecture v4T) and Cortex M1 (architecture v6-M variant).The set of instructions compatible with both targets excludes the 16-bit Thumb SVC (formerly SWI) instruction. In effect we need a pseudo architecture *v4T-no-SVC* to describe such code, but no such architecture exists so it cannot simply be added to the `Tag_CPU_arch` enumeration.

We have chosen to deal with this problem describing such code as *v4T also compatible with v6-M* (or as *v6-M also compatible with v4T*) using the following *safely ignorable* (§2.2.6) tag.

```
Tag_also_compatible_with (=65), NTBS: data
```

The data string must be further interpreted as a ULEB128-encoded tag followed by a value of that tag. The value bytes of the NUL-terminated data string may be:

- A ULEB128-encoded number followed by a NUL byte.
- A NUL-terminated byte string.

The following byte sequence records the intention to also be compatible with architecture version v6-M.

```
Tag_also_compatible_with (=65) Tag_CPU_arch (=6) v6-M (=11) 0
```

At this release of the ABI (r2.06) the only defined use of `Tag_also_compatible_with` is to express *v4T also compatible with v6-M* and *v6-M also compatible with v4T*. All other uses are RESERVED to the ABI. Future releases of the ABI may relax this constraint.

2.3.7.4 Conformance tag

The conformance tag describes the version of the ABI to which conformity is claimed by an entity.

```
Tag_conformance (=67), string: ABI-version
```

This version of the ABI is “2.06”. The minor version (dot-06) is for information and does not affect the claim. Version “0” denotes no claim to conform and is the default if the tag is omitted.

To simplify recognition by consumers in the common case of claiming conformity for the whole file, this tag should be emitted *first* in a *file-scope* sub-subsection of the *first* public subsection of the attributes section.

In this case, the *dot-ARM-dot-attributes* section would begin “A~~~~aeabi\0\1~~~~C2.06”, where ‘~’ denotes an unknown byte value.

2.3.7.5 No defaults tag

The occurrence of a *no defaults* tag in a file-scope sub-subsection indicates that the value of a tag with no file-scope definition is UNDEFINED rather than 0.

```
Tag_nodefaults (=64), uleb128: ignored (write as 0)
```

A consuming tool may take IMPLEMENTATION DEFINED action if any tag still has an UNDEFINED value after a *dot-ARM-dot-attributes* section has been fully processed.

(The IMPLEMENTATION DEFINED behavior for consumers that do not recognize this tag is that UNDEFINED values default to 0).

To make processing easy for consumers, this tag should be emitted before any other tag other than the conformance tag (§2.3.7.4).

We expect the *no defaults* tag is to be used only by linkers that merge relocatable files.

2.4 ARM CPU names recognized by RVCT 3.1 armcc

The RVCT 3.1 armcc recognizes the following ARM processor products.

ARM7EJ-S	ARM9TDMI	ARM1020E	MPCore
ARM7TM	ARM920T	ARM1022E	MPCoreNoVFP
ARM7TDM	ARM922T	ARM1026EJ-S	Cortex-A8
ARM7TDMI	ARM940T	ARM1136J-S	Cortex-A8NoNeon
ARM710T	ARM9E-S	ARM1136JF-S	Cortex-R4
ARM720T	ARM9EJ-S	ARM1136J-S-rev1	Cortex-M3-rev0
ARM740T	ARM926EJ-S	ARM1136JF-S-rev1	Cortex-M3
ARM7TM-S	ARM946E-S	ARM1156T2-S	Cortex-M1
ARM7TDMI-S	ARM966E-S	ARM1156T2F-S	Cortex-M1.os_extension
ARM810	ARM968E-S	ARM1176JZ-S	Cortex-M1.no_os_extension
		ARM1176JZF-S	

(Many of these names are trademarks of ARM Limited. For details see http://www.arm.com/legal.html).

The following pseudo processor names are also recognized as architecture version names.

4	5T	6	7
4T	5TE	6K	7-A
	5TEJ	6T2	7-R
		6Z	7-M
		6-M	
		6S-M	

2.5 Attributes summary and history

Table 1lists the defined attribute tag names, their numerical values, and their parameter types. For each tag the table lists the ABI version in which it was introduced, and the versions in which its parameter values or meaning were amended.

A program claiming conformance to revision r2.*x* of the ABI must comprehend tags with values less than 64 (§2.2.6) defined in revisions r2.*x* and earlier, and must comprehend all parameter values defined for those tags by those revisions. Table 2 summarizes amendments to the specification of build attributes ABI revision by ABI revision.

Table 1, Summary and history of individual attributes

Tag	Value	Parameter type	ABI revision	Amendments
Tag_File	1	uint32	r2.0	
Tag_Section	2	uint32	r2.0	
Tag_Symbol	3	uint32	r2.0	
Tag_CPU_raw_name	4	NTBS	r2.0	
Tag_CPU_name	5	NTBS	r2.0	
Tag_CPU_arch	6	uleb128	r2.0	**r2.06**:Added enum values for v6-M, v6S-M
Tag_CPU_arch_profile	7	uleb128	r2.0	**r2.05**: Added 'S' denoting 'R' or 'M' (don't care)
Tag_ARM_ISA_use	8	uleb128	r2.0	
Tag_THUMB_ISA_use	9	uleb128	r2.0	
Tag_VFP_arch	10	uleb128	r2.0	**r2.04**: Added enumerated value for VFPv3 **r2.06**: Added enumerated value for VFPv3 restricted to D0-D15
Tag_WMMX_arch	11	uleb128	r2.0	**r2.02**: Added an enumerated value for wireless MMX v2.
Tag_Advanced_SIMD_arch	12	uleb128	r2.0	
Tag_PCS_config	13	uleb128	r2.0	
Tag_ABI_PCS_R9_use	14	uleb128	r2.0	
Tag_ABI_PCS_RW_data	15	uleb128	r2.0	
Tag_ABI_PCS_RO_data	16	uleb128	r2.0	
Tag_ABI_PCS_GOT_use	17	uleb128	r2.0	
Tag_ABI_PCS_wchar_t	18	uleb128	r2.0	
Tag_ABI_FP_rounding	19	uleb128	r2.0	
Tag_ABI_FP_denormal	20	uleb128	r2.0	
Tag_ABI_FP_exceptions	21	uleb128	r2.0	
Tag_ABI_FP_user_exceptions	22	uleb128	r2.0	
Tag_ABI_FP_number_model	23	uleb128	r2.0	
Tag_ABI_align8_needed	24	uleb128	r2.0	
Tag_ABI_align8_preserved	25	uleb128	r2.0	
Tag_ABI_enum_size	26	uleb128	r2.0	
Tag_ABI_HardFP_use	27	uleb128	r2.0	
Tag_ABI_VFP_args	28	uleb128	r2.0	

Tag	Value	Parameter type	ABI revision	Amendments
Tag_ABI_WMMX_args	29	uleb128	r2.0	
Tag_ABI_optimization_goals	30	uleb128	r2.0	
Tag_ABI_FP_optimization_goals	31	uleb128	r2.0	
Tag_compatibility	32	NTBS	r2.0	**r2.05**: Revised the previously ineffective definition. The new definition is compatible with the old one if a nonsensical clause in the old one is ignored.
Tag_CPU_unaligned_access	34	uleb128	r2.02	
Tag_VFP_HP_extension	36	uleb128	r2.06	
Tag_ABI_FP_16bit_format	38	uleb128	r2.06	
Tag_nodefaults	64	uleb128	r2.06	
Tag_also_compatible_with	65	NTBS	r2.05	**r2.06**: Restricted usage as noted in §2.3.7.3.
Tag_conformance	67	NTBS	r2.05	

Table 2, History of attributes in ABI revisions and ABI-Addenda document versions

ABI Revision	Doc vsn	Date	Amendments
r2.0	v1.0	March 2005	Initial release of the build attributes specification.
r2.01	v1.01	5th July 2005	Added extensibility and compatibility rules (now §2.2.6).
r2.02	v1.03	13th October 2005	Added wMMX v2 to Tag_WMMX_arch enumeration. Added Tag_CPU_unaligned_access.
r2.03	v1.04	6th January 2006	No changes to the specification of build attributes.
r2.04	v1.05	8th May 2006	Added VFPv3 to Tag_VFP_arch enumeration.
r2.05	v1.06	25th January 2007	Added introductory section 2.1, *Purpose and principles of operation*. Clarified that all current uleb128 values are also plain byte values. Clarified inheritance of default values through nested scopes. Clarified the definitionof *conformance*. Added 'S' to the Tag_CPU_arch_profile enumeration. Corrected the previously useless definition of Tag_compatibility. Added Tag_also_compatible_with and Tag_conformance. Added section 2.3.6 about combining attribute values. Many minor editorial improvements.
r2.06	A	October 2007	Major re-write of material with emphasis on clear presentation. Added v6-M and v6S-M to Tag_VFP_arch enumeration. Added VFPv3 restricted to D0-D15 to Tag_VFP_arch enumeration. Added Tag_VFP_HP_extension, Tag_ABI_FP_16bit_format, and Tag_nodefaults. Restricted the use of Tag_also_compatible_with in the face of persistent difficulties with its formal definition.

3 ADDENDUM: THREAD LOCAL STORAGE

3.1 Introduction to 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. It fits into the abstract storage hierarchy as follows.

- (**Most global**) Program-own data (static and extern variables, instanced once per program/process).
- Thread local storage (variables instanced once per thread, shared between all accessing function activations).
- (**Most local**) Automatic data (stack variables, instanced once per function activation, per thread).

Thread local storage generates a number of issues at a number of levels, not all of which affect an ABI.

- How to denote TLS in source programs.

  Gcc uses __tls T t…; Microsoft use __declspec(thread) T t…; this is Q-o-I.
- How to represent the initializing images of TLS in object files, and how to define symbols in TLS.

  The rules for ELF are well established (see SHF_TLS, STT_TLS in [GELF])
- How a loader or run-time system creates instances of TLS per-thread at execution time.

  This is part of ABI for the platform or execution environment.
- How to relocate, statically and dynamically, with respect to symbols defined in TLS (for details of relocations relevant to ARM Linux see [AAELF]).
- How code must address variables allocated in TLS (the subject of the notes below).

It is the last two bullet points that are the subject of this ABI.

3.2 Introduction to TLS addressing

In the most general form supported by Linux, Windows, and similar platforms, a program is constructed dynamically from an application and a number of shared libraries (called, respectively, DSOs or DLLs). Each component (application or shared library) can be mapped into multiple processes. Additionally, a DSO or DLL can be loaded dynamically by a program, rather than being part of the initial process image constructed when the program is first loaded.

For the purpose of addressing TLS, both Linux and Windows identify the components of an application using indexes. On both platforms, indexes are allocated dynamically when a process is created, or when a DSO or DLL is loaded dynamically. The details of how indexes are allocated are specific to a platform and differ significantly between Linux and Windows.

A component can have a different TLS index in two different processes, so its thread index must be part of its program-own state (or be queried dynamically). The run-time system is responsible for maintaining a per-thread vector of pointers to allocated TLS regions indexed by these component indexes. Under both Linux and the Win32 API, access to the vector is hidden behind run-time functions.

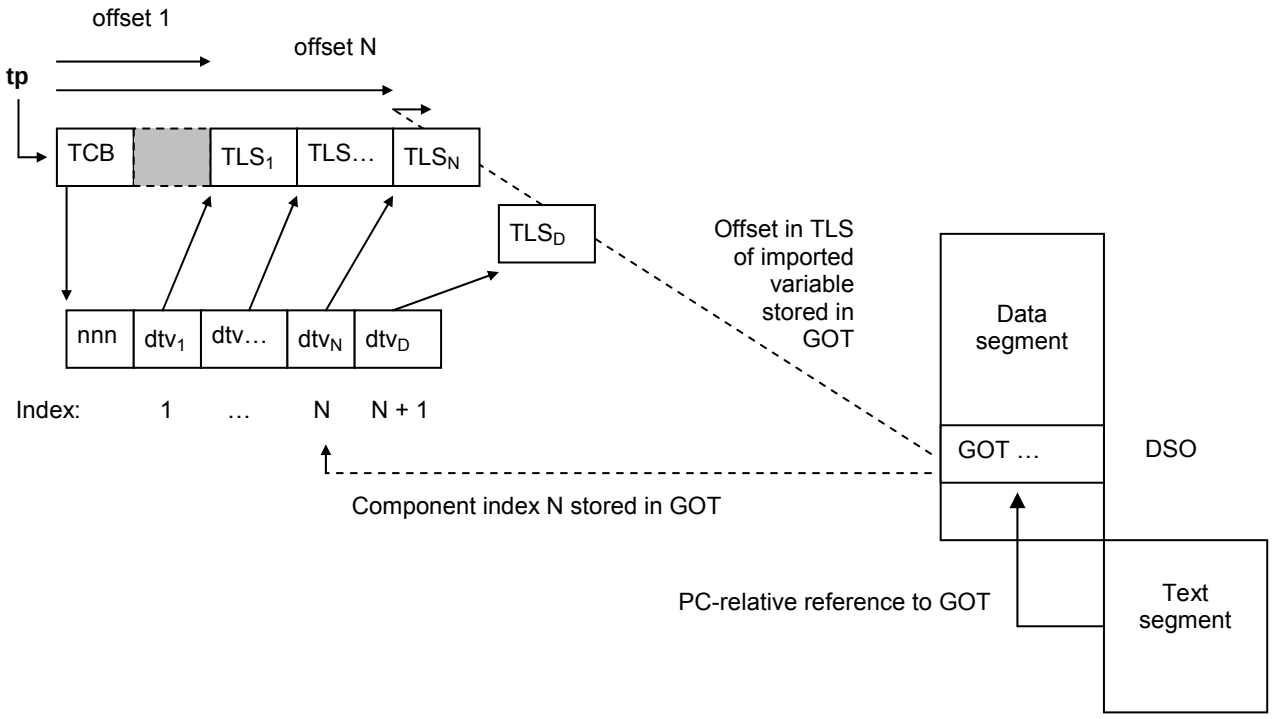
There is a system resource (such as a dedicated register) called the *thread pointer* that, typically, points to a control block for the currently executing thread which, in turn, points to the TLS vector for that thread.

At this stage, the details of TLS addressing become quite platform specific. In the next subsection we describe the concepts appropriate to Linux for ARM, which is all that this revision of the ABI supports.

3.3 Linux for ARM TLS addressing

Figure 3-1 depicts the fundamental components of the TLS addressing architecture used by Linux for ARM.

Figure 3-1, Linux for ARM TLS addressing architecture



In the most general case, the location of an imported thread local datum – or an exported datum that might be pre-empted – is represented by a pair of GOT entries that give:

- The index in the dynamic thread vector of the pointer to the TLS block containing the datum (the application itself has index 1 and index 0 is reserved to the run-time system).
- The offset of the datum in the pointed-to TLS block.

In the most general case the expression to address a thread local symbol S is:

(**tp**[0])[GOT_S[0]] + GOT_S[1]

3.3.1 Linux for ARM general dynamic model

In the *general dynamic model*, the addressing expression is packaged by a call to __tls_get_addr that takes a single parameter, the address of GOT_S. The code sequence and required relocations are shown in Table 3, below.

A space-optimized version of the general dynamic model that calls ___tls_get_addr (triple-underscore) is shown in Table 4, below. The parameter passed to ___tls_get_addr is the offset of GOT_S from lr.

The Linux for ARM TLS model has two *local dynamic* variants. These are used to address component-local thread-local variables more efficiently, but it can still be used in a dynamically loaded DSO. (A variable is local if its symbol S has STB_LOCAL binding or non-STV_DEFAULT visibility). In these variants the offset of variable in the component's TLS segment in known at link time and only the component index must be loaded from the GOT.

Specimen code sequences and relocations are given in Table 5, below. This model is advantageous whenever a function accesses more than one variable (the address of the TLS block can be a common sub-expression).

Table 3, General dynamic model (time optimized)

Location	General dynamic model code sequence	Relocation	Symbol

.text	ldr r0, .Lt0
	.L1 add r0, pc, r0
	bl __tls_get_addr
	
	ldr rS, [r0] ; Load S...
		R_ARM_CALL	__tls_get_addr

literal pool	.Lt0 .word S(tlsgd) + [. – .L1 – 8]	R_ARM_TLS_GD32	S
GOT[S]	.word .word	R_ARM_TLS_DTPMOD32 R_ARM_TLS_DTPOFF32	S S

Table 4, General dynamic model (space optimized)

Location	General dynamic model (space optimized)	Relocation	Symbol

.text	ldr r0, .Lt0
	.L1 bl __tls_get_addr
	
	ldr rS, [r0] ; Load S...
		R_ARM_CALL	__tls_get_addr

literal pool	.Lt0 .word S(tlsgd) + [. – .L1 – 4]	R_ARM_TLS_GD32	S
GOT[S]	.word .word	R_ARM_TLS_DTPMOD32 R_ARM_TLS_DTPOFF32	S S

Table 5, Local dynamic models

Location	Local dynamic model	Relocation	Symbol

.text	ldr r0, .Lt0
	.L1 add r0, pc, r0
	bl __tls_get_addr
	
	... r0 points to my TLS, a CSE ...
	
	ldr r1, .Lt1
	ldr rX, [r0, r1] ; long offset to X
	
	... ; but a short offset to Y
	
	ldr rY, [r0, #Y(tlsldo)]
		R_ARM_CALL
		
		R_ARM_TLS_LDO12
			__tls_get_addr
			
			Y

literal pool	.Lt0 .word X(tlsldm) + [. – .L1 – 8] .Lt1 .word Y(tlsldo)	R_ARM_TLS_LDM32 R_ARM_TLS_LDO32	X Y
GOT[X]	.word .word 0	R_ARM_TLS_DTPMOD32	X

R_ARM_TLS_LDM32 sets the first element of the GOT pair to the symbol's dynamic TLS vector index, as does R_ARM_TLS_GD32, but sets the second element to 0 (as shown in Table 5, above).

TLS-related relocations for Linux for ARM are described further in [AAELF].

3.3.2 Linux for ARM static (initial exec) model

If DSOs need not be loaded dynamically, more efficient addressing modes can be constructed if the run-time system allocates TLS at process creation time. As depicted in Figure 3-1, the offset from the thread pointer **tp** to a component's TLS is then known at process creation time (dynamic link time), so a component can address its thread local variables relative to **tp**, without indexing the dynamic thread vector.

Below, **$tp** denotes a general purpose register containing the result of reading **tp**.

Table 6, below, shows the general model. It works for DSOs and the root application, in ARM and Thumb state.

Table 7 below shows an optimized model for DSOs If the compiler uses a GOT pointer (denoted by $gp) and small PIC (12-bit PC-relative) addressing. This model works only in ARM state.

Finally, Table 8 shows the code to access an application's local thread-local variables. Because an application's TLS is allocated first in the initial TLS vector for the process, the offsets of its variables from **tp** are known at *static* link time and do not need to be read from the GOT. The 12-bit model works only in ARM state.

Table 6, General initial exec model

Location	Initial exec model code sequence	Relocation	Symbol

.text	`ldr r0, .Lt0`
	`.L1 ldr r0, [pc, r0]`
	`...`
	`ldr rS, [$tp, r0] ; Load S...`		

literal pool	`.Lt0 .word S(tpoff) + [. – .L1 – 8]`	R_ARM_TLS_IE32	S
GOT[S]	.word	R_ARM_TLS_TPOFF32	S

Table 7, Initial exec model, DSO with GOT pointer and small FPIC addressing (DSO only)

Location	Initial exec model code sequence	Relocation	Symbol

.text	`ldr r0, [$gp, #S(gottpoff)]`
	`...`
	`ldr rS, [$tp, r0] ; Load S...`
		R_ARM_TLS_IE12GP	S

GOT[S]	.word	R_ARM_TLS_TPOFF32	S

Table 8, Initial exec model, access to an application's local thread-local variables

Location	Initial exec model code sequence	Relocation	Symbol

.text	`ldr r0, .Lt0`
	`...`
	`ldr rX, [$tp, r0] ; Load X`
	`...`
	`ldr rY, [$tp, #Y(tpoff)] ; Load Y`
		R_ARM_TLS_LE12	Y

literal pool	.Lt0 .word S(tpoff)	R_ARM_TLS_LE32	X

TLS-related relocations for Linux for ARM are described further in [AAELF].

4 RESERVED NAMES

For external symbols defined/required by the C++ ABI there are already agreed names – either the standard mangling of C++ names or names like __cxa_acquire_guard.

The *ABI for the ARM Architecture* also needs a space of global symbol names private to each compiler vendor – external names guaranteed not to clash with other vendors – and a C++ name space private to each vendor.

Many of these names have C or assembly language linkage, so we propose to reserve names of the form *__vendor-name_name*, for example:

```
__ARM_foo       __gnu_foobar     __cxa_foobaz
```

In each case, namespace *__vendor-name*[v*n*] is also reserved in C++, forr example:

```
namespace __ARM {...}           namespace __ARMv2 {...}
namespace __gnuv1 {...}         namespace __aeabi {...}
```

We also reserve the corresponding upper case vendor name with a single leading underscore to use by the vendor for C macro names, for example:

```
#if _AEABI_... != 0
#if _ARM_... == 2
```

Prefix names themselves must not contain underscore ('_') or dollar ('$'). The following prefixes are registered.

Table 9, Namespace prefixes registered under the ABI for the ARM Architecture

Name	Vendor
aeabi	Reserved to the ABI for the ARM Architecture (EABI pseudo-vendor)
AnonXyz anonXyz	Reserved to private experiments by the *Xyz* vendor. Guaranteed not to clash with any registered vendor name.
ARM	ARM Ltd (Note: the company, not the processor).
cxa	C++ ABI pseudo-vendor
GHS	Green Hills Systems
gnu	GNU compilers and tools (Free Software Foundation)
iar	IAR Systems
intel	Intel Corporation
ixs	Intel Xscale
PSI	PalmSource Inc.
TASKING	Altium Ltd.
TI	TI Inc.
tls	Reserved for use in thread-local storage routines.
WRS	Wind River Systems.

5 ERRATA AND MINOR ADDENDA

This section details errors found in the *ABI for the ARM Architecture* after publication of version 2.0 and minor additions made since then.

5.1 DWARF for the ARM Architecture

5.1.1 Clarifications

(v2.01, r2.02) The ABI-v2.0 DWARF register numbering scheme for VFP registers (S0-S31 → 64-95) has been declared *obsolescent*. It will become *obsolete* in the next major release of the *ABI for the ARM Architecture*.

5.1.2 Errors fixed

(v2.02, r2.04) §3.3, *Describing other endian data*, suggested that DW_AT_endianness, coded as 0x5b, might be approved by the DWARF3.0 standardization committee. In fact, 0x5b was used for another purpose and DW_AT_endianity, coded as 0x65, has been approved. The parametric values of this attribute have also changed their names and encodings.

5.1.3 Additions and omissions fixed

(v2.01, r2.02) §3.1.1, *VFP-v3 and Neon register descriptions*, specifies how to describe the VFP-v3/Neon SIMD register file. There is a new range of DWARF register numbers allocated to D0-D31 and new schemes for describing Neon Q registers and VFP S registers. The new numbering should also be used for VFP-v2.

5.2 ELF for the ARM Architecture

5.2.1 Clarifications

(v1.05, r2.06) §4.1.1, *Registered Vendor Names*: Inserted the complete table of registered vendor names, now shared among AAELF, CLIBABI, and RTABI.

(v1.02, r2.03) §4.2, *ELF Header*: Corrected the wording of the description of e_entry.

(v1.03, r2.04) §4.2, *ELF Header*: Clarified that bit[0] of [e_entry] controls the instruction set selection on entry.

(v1.02, r2.03) §4.5.4, *Symbol names*: Clarified the necessary restrictions on local symbol removal in relocatable files.

(v1.04, r2.05) §4.6.1.2, *Relocation types*: Clarified that 'Pa' is the adjusted address of the place being relocated, with the Thumb-bit stripped (defined as P & 0xFFFFFFFE), for Thumb state LDR- and ADR-type relocations.

(v1.04, r2.05) §4.6.1.4, *Static ARM relocations*: Clarified that R_ARM_LDR_PC_G0 applies equally to LDRB, STRB.

(v1.04, r2.05) §4.6.1.6, *Static Thumb32 relocations*: Added this section explicitly tabulating the relocations specific to 32-bit Thumb instructions.

(v1.05, r2.06) §4.1.1, *Registered Vendor Names*: Inserted the complete table of registered vendor names, now shared among AAELF, CLIBABI, and RTABI.

5.2.2 Errors fixed

(v1.01, r2.01) §4.3.2, *Section Types*, Table 4-4, *Processor specific section types*: The definition of SHT_ARM_ATTRIBUTES was given the value 0x70000002, already allocated to SHT_ARM_PREEMPTMAP (used by RVCT 2.2 and others). The correct value of SHT_ARM_ATTRIBUTES is 0x70000003.

(v1.04, r2.05) §4.6.1.2, *Relocation types*: Corrected the relocation formulae for the R_ARM_ALU_{PC|SB}_G_N_NC relocations so that they uniformly include the obligation to set the T bit when this is required.

(v1.05, r2.06) §4.6.1.2, *Relocation types*: Corrected the definition of Pa (introduced in r2.05) which had the wrong mask.

(v1.05, r2.06) §4.6.1.9, *Relocations for thread-local storage*: Corrected the text following Table 4-15, *Static TLS relocations*, which misspelled R_ARM_TLS_LE32 (relocation #108) and R_ARM_TLS_LE12 (relocation #110).

5.2.3 Additions and omissions fixed

(v1.01, r2.01) §4.3.2, *Section Types*, Table 4-4, *Processor specific section types*: The definition of SHT_ARM_PREEMPTMAP was omitted. The correct value is 0x70000002.

(v1.03, r2.04) §4.3.3.1, *Merging of objects in sections with SHF_MERGE*: Rules governing SHF_MERGE optimizations are needed to support inter-operation between tool chains (omitted from release 1.02 and earlier).

(v1.01, r2.01) §4.3.4, *Special sections*, Table 4-5, *ARM special sections*: The definition and explanation of `.ARM.preemptmap` were omitted.

(v1.03, r2.04) §4.6.1.1, *Addends and PC-bias compensation*: Release 1.03 makes explicit the rules describing the required initial addends for REL-type relocations.

(v1.04, r2.05) §4.6.1.2, *Relocation types*: Added additional relocations to support a new, experimental Linux TLS addressing model described in §4.6.1.9, *Relocations for thread-local storage*.

(v1.02, r2.03) §4.6.1.9, *Dynamic relocations*: Added a definition of R_ARM_RELATIVE when S = 0; described the new, experimental, Linux TLS addressing model.

(v1.02, r2.03) §5.2.1, *Platform architecture compatibility information*: Added a specification of architecture compatibility information for executable files.

5.3 Procedure Call Standard for the ARM Architecture

5.3.1 Clarifications

(v2.05, r2.04) Added **§5.1,** *Machine Registers*, clarifying the roles of core registers and co-processor registers in the AAPCS.

(v2.03, r2.02) §5.5, *Parameter Passing*, clarified that a callee may overwrite an incoming parameter area on the stack.

(v2.03, r2.02) §5.1.1.1, *VFP register usage conventions (VFP v2 and v3)*, described how VFP-v3 d16-d31 are used.

(v2.04, r2.04) §5.3.1.1, *Use of IP by the linker*, clarified when linking may insert intra-call veneers that may corrupt r12 and the condition flags.

(v2.01, r2.01) §7.1.3, *Enumerated Types*, following Table 5, *Enumerator container types*: Clarified that if a platform chooses that all container types should be word sized, the type of the container is `int` unless the upper bound of the enumerator exceeds 2147483647.

5.3.2 Errors fixed

(v2.03, r2.02) §7.1.7, *Bit-fields*, retracted the requirement that the type of a plain bit-field be *unsigned* by default.

5.3.3 Additions and omissions fixed

(v2.05, r2.05) §5.1.1.1, *Handling values larger than 32 bits*, added in support of the *Advanced SIMD Architecture*.

(v2.05, r2.05) §5.1.2.1, *VFP register usage conventions*, extended in support of the *Advanced SIMD Architecture*.

(v2.05, r2.05) §5.4 *Result Return*, extended in support of the *Advanced SIMD Architecture*.

(v2.05, r2.05) §6.1.1, *Mapping between registers and memory format*, added in support of the *Advanced SIMD Architecture*.

(v2.05, r2.05) §6.1.2.1, *Result return* and **§6.1.2.2,** *Parameter passing*, extended in support of the *Advanced SIMD Architecture*.

(v2.05, r2.05) *APPENDIX A SUPPORT FOR ADVANCED SIMD EXTENSIONS*, added in support of the *Advanced SIMD Architecture*.

(v2.06, r2.06) §4.1, *Fundamental Data Types*: Added half-precision floating point to the data type table.

(v2.06, r2.06) Added **§4.1,** *Half-precision Floating Point*.

(v2.06, r2.06) §5.4, *Result Return,* **§6.1.2.1,** *Result return*: Extended the rules to half-precision floating point.

(v2.06, r2.06) §5.5, *Parameter Passing,* **§6.1.2.1,** *Parameter passing*: Extended the rules to half-precision floating point.

(v2.06, r2.06) Added **§6.4.4**, *Half-precision Format Compatibility*.

(v2.06, r2.06) §7.1.1, *Arithmetic Types*: Added __f16 (as an ARM extension) to Table 3.

(v2.06, r2.06) §A.2, *Advanced SIMD data types*: Added float16x4_t to Table7.

5.4 Base Platform ABI for the ARM Architecture

No changes since version 2.0.

5.5 C Library ABI for the ARM Architecture

5.5.1 Clarifications

(v2.01, r2.01) General: When a library function is added to a header (e.g. following additions to the C standard), any inline (e.g. macro-implemented) version should be hidden *when* _AEABI_PORTABILITY_LEVEL != 0.

(v2.01, r2.01) §5.3.1.1, *Encoding of ctype table entries and macros (_AEABI_PORTABILITY_LEVEL != 0)*: Names of macros (__A, __X, etc) are for illustration only and are not mandated by the specification.

(v2.04, r2.06) §3.4, *Private names for private and AEABI-specific helper functions*: Inserted the complete table of registered vendor names, now shared among AAELF, CLIBABI, and RTABI.

5.5.2 Errors fixed

(v2.01, r2.01) §5.9, *locale.h*, **Table 6**, *struct __aeabi_lconv*: `struct lconv` should be `struct __aeabi_lconv`.

(v2.01, r2.01) §5.11, *setjmp.h*, **Table 10**, *setjmp.h definitions*: The text "((__aeabi_JMP_BUF_SIZE))" at the end of the table is left over from a previous version and should be deleted.

(v2.02, r2.02) §5.3.1.1, *Encoding of ctype table entries and macros (_AEABI_PORTABILITY_LEVEL != 0)*: The C99 isblank() function cannot be implemented as a macro within this framework because __B is required to exclude tab when used by the C89 function isprint() and to include it when used by isblank(). To fix this, we define isblank() out of line, but allow compilers to inline the obvious implementation that is excluded from being a macro because it evaluates its parameter twice.

(v2.03, r2.04) §5.12, *signal.h*: Corrected misinformation suggesting that it might be possible to access 8-byte types using LDRD/STRD and LDM/STM.

5.5.3 Additions and omissions fixed

(v2.01, r2.01) §4.1, *C Library overview*, **Table 1**, *C library headers*: The C99 header stdbool.h is missing. There are no portability implications of providing it. A new **§5.14**, describes stdbool.h, and *Table 19*, *Summary of conformance requirements when _AEABI_PORTABILITY_LEVEL != 0* in **§6** has a new entry for stdbool.h.

(v2.01, r2.01) §5.3.1, *ctype.h when _AEABI_PORTABILITY_LEVEL != 0 and isxxxxx inline*: The same character translations and locale bindings should be used for the implementations of the **to***xxxx* macros and functions as are used for the **is***xxxx* macros and functions.

5.6 C++ ABI for the ARM Architecture

5.6.1 Clarifications

(v2.02, r2.03) §3.2.5.5, *Inter-DLL visibility rules for C++ ABI-defined symbols*: Clarify that entities defined in unnamed namespaces must not be exported (because unnamed namespaces do not have globally defined names).

(v2.03, r2.04) In **§3.2.2.3**, *Library helper functions* [definition of __aeabi_atexit at the end of the section], **§3.2.2.5**, *Code example for __aeabi_atexit*, **§3.2.4.2**, *Static object destruction*: Clarified the use of __aeabi_atexit for registering atexit functions.

(v2.04, r2.06) In **§1.2,** *References,* Updated the base standard for C++ to ISO/IEC 14882:2003.

5.6.2 Errors fixed

(v2.01, r2.01) §3.2.4, *Static object construction and destruction*: The global constructor vector section (.init_array) of ELF type SHT_INITARRAY, was erroneously specified to be read-only. The generic ELF specification requires these sections to have the SHF_WRITE flag set. The *C++ ABI for the ARM Architecture* requires producers to generate these sections *as if* they were read-only. (Some execution environments require .init_array sections to be read-only and linkers targeting these environments may drop the SHF_WRITE flag. Doing so must not cause run-time failures).

5.6.3 Additions and omissions fixed

(v2.04, r2.06) In **§3.1**, *Summary of differences from and additions to the generic C++ ABI*, specified the name mangling (GC++ABI §5.1.5) for the 16-bit FP type added to [AAPCS].

(v2.04, r2.06) In **§3.2.6**, *ELF binding of static data guard variable symbols*, added an ARM-specific rule for the ELF binding of guard variable symbols.

5.7 Exception Handling ABI for the ARM Architecture

5.7.1 Clarifications

(v2.04, r2.05) In paragraph **5** in **§7.4**, *Phase 2 unwinding*, clarified that an unwinder must in general restore *all* the machine registers listed in the VRS.

(v2.02, r2.02) In **§7.7**, *Implications for implementations*, and **§8.4.1**, *Compiler helper functions*, we clarify that _Unwind_Complete may overwrite UCB fields specific to the exception propagation that has just completed, and make clear the consequences of this. In **§8.4.1** we are less prescriptive about which of__cxa_allocate_exception and __cxa_throw initialize the UCB and LEO fields.

5.7.2 Errors fixed

(v2.02, r2.02) In **§8.4.1**, *Compiler helper functions*, we have added the specification of __cxa_get_exception_ptr. Adding this function and having compilers generate calls to it, corrects a non-conformance with the C++ Standard. This function was recently added to the Itanium ABI, from which the ARM EHABI is derived. There are consequential changes to **§8.2**, bullet 1 to mandate that barrier_cache.bitpattern[0] is valid on catch handler entry and to **§8.4.1** in the definition of __cxa_begin_catch, regarding initialisation of barrier_cache.bitpattern[0]. **§8.5.4** adds an example of using __cxa_get_exception_ptr in a handler entry sequence.

Detailed rationale: The C++ Standard states that std::uncaught_exception() should return *true* after completing evaluation of the object to be thrown until completing the initialization of the exception-declaration in the matching catch handler. Thus for example:

```
try {
  .......
  throw S();
  ........
} catch (S s) {
  ......
}
```

uncaught_exception() should return *true* between the end of the S() call and the end of the initialization of s. This has not been the case in several implementations, such as g++ and RVCT 2.2, where uncaught_exception() was incorrectly *false* during any copy construction of s. Version 2.0 of the EHABI cannot handle this case correctly.

The original Itanium spec declared `void __cxa_begin_catch(void *exceptionObject)` and required it be called on entry to a catch handler, and in some other circumstances, to perform some housekeeping including updating the uncaught_exception count. The void *exceptionObject is really intended to be a pointer to (IA_64 terminology) an _Unwind_Exception object, the language-independent sub-object within a propagated exception object. The handler code itself (as ARM understands it) was supposed to obtain a pointer to the matched C++ object by being passed such a pointer in a machine register, or by some other unspecified means. In EHABI terminology, _Unwind_Exception is _Unwind_Control_Block.

The g++ community, HP, and ARM all changed __cxa_begin_catch to return the pointer to the matched C++ object, thus avoiding the need to save the register containing the matched object pointer over the call to

__cxa_begin_catch. On return from __cxa_begin_catch, initialization of the catch parameter then proceeds. In other words, the code sequence is `BL __cxa_begin_catch, initialize catch parameter` if there is one.

Unfortunately, because __cxa_begin_catch has updated the uncaught_exception count, uncaught_exception() will return the wrong value if it is called during the initialization – as is possible if the catch parameter is a class type with non-trivial copy constructor. This is corrected by using the new routine, when the code sequence for catches with a parameter of class type with a non-trivial copy constructor becomes:

```
save r0 somewhere
BL __cxa_get_exception_ptr
initialize catch parameter
recover r0
BL __cxa_begin_catch
```

The wording change in **§8.4.1** precludes the unlikely (and undesirable) possibility that handler code received the matched object pointer in some other place, then copied it to barrier_cache.bitpattern[0], without overly constraining the implementation.

5.7.3 Additions and omissions fixed

(v2.02, r2.02) In **§7.2**, *Language-independent unwinding types and functions*, we have added _Unwind_DeleteException whose behaviour is described in **§7.6**, *Cross-language support*. This function is present in the Itanium ABI and is a convenience function with no cost if not used by an implementation. Some vendors have requested ARM add this to remove a need for conditional compilation when targeting the ARM ABI verses other ABIs. The ARM definition is compatible with the Itanium requirements.

(v2.02, r2.02) In **§7.5.2**, *Assignment to VRS registers*, **§7.5.3**, *Reading from VRS registers*, **§7.5.4**, *Moving from stack to VRS*, and **§9.3**, *Frame unwinding instructions*, we have added **support for VFPv3**. The ARM VFP v3 adds 16 double precision registers, D16-D31. It should be possible to restore these during unwinding but because the registers are intended for scratch use, this is expected to be uncommon. Specifically:

- In **§7.5.2**, *_Unwind_VRS_Set*, we extend the register range for _UVRSC_VFP/_UVRSD_DOUBLE to 0-31.
- In **§7.5.3,** *_Unwind_VRS_Get*, we extend the register range for _UVRSC_VFP/_UVRSD_DOUBLE to 0-31.
- In **§7.5.4,** *_Unwind_VRS_Pop*, we clarify that _UVRSC_VFP/_UVRSD_DOUBLE undoes the effect or one or more FSTMD instructions.
- In **§9.3** we allocate unwinding instruction "11001000 sssscccc" to popping D[16+ssss]-D[16+ssss+cccc], to permit recovery of a range of the new registers (including any range containing just one register).
- In **§9.3** Update the *remarks* (split remark d and add extra words) to clarify the limitations of VFP unwinding instructions (No incompatible change)

5.8 Run-time ABI for the ARM Architecture

5.8.1 Clarifications

(v2.03, r2.06) §3.8, *Private names for private and AEABI-specific helper functions*: Inserted the complete table of registered vendor names, now shared among AAELF, CLIBABI, and RTABI.

(v2.03, r2.06) §4.3.1, *Integer (32/32 → 32) division functions*: Clarified the meaning of signed integer division when the result cannot be represented (MIN_INT/-1).

5.8.2 Errors fixed and features withdrawn or deprecated

(v2.02, r2.05) §4.1.2, *The floating-point helper functions*, **deprecated** *use* of `fneg` and `dneg` because inlining these functions is always more efficient, even with the Thumb-1 ISA. Conforming library implementations must still provide implementations.

5.8.3 Additions and omissions fixed

(v2.01, r2.02) In (new) **§4.3.5,** *Thread-local storage*, we define __aeabi_read_tp() that returns the thread pointer denoted by **$tp** in §3.3.2, *Linux for ARM static (initial exec) model*, of this specification.

(v2.01, r2.02) In **§4.4.7**, *Exception-handling support*, brought the list of __cxa_ functions up to date and in line with [EHABI].