ELF for the ARM® Architecture

Document number: ARM IHI 0044F, current through ABI release 2.10
Date of Issue: 24th November 2015

Abstract

This document describes the processor-specific definitions for ELF for the Application Binary Interface (ABI) for the ARM architecture.

Keywords

Object files, file formats, linking, EABI, ELF

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 *ARM Software development tools* section, *ABI for the ARM Architecture* subsection).

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

Licence

Proprietary notice

Contents

1 ABOUT THIS DOCUMENT 5

1.1 Change control 5
1.1.1 Current status and anticipated changes 5
1.1.2 Change history 5

1.2 References 6

1.3 Terms and abbreviations 7

1.4 Your licence to use this specification 8

1.5 Acknowledgements 9

2 SCOPE 10

3 PLATFORM STANDARDS 11

3.1 Base Platform ABI (BPABI) 11
3.1.1 Symbol Versioning 11
3.1.1.1 Symbol versioning sections 11
3.1.1.2 Locating symbol versioning sections 11
3.1.1.3 Version definition section 12
3.1.1.4 Symbol version section 12
3.1.1.5 Versions needed section 13
3.1.2 Symbol Pre-emption in DLLs 13
3.1.2.1 Pre-emption Map Format 13
3.1.3 PLT Sequences and Usage Models 14
3.1.3.1 Symbols for which a PLT entry must be generated 14
3.1.3.2 Overview of PLT entry code generation 14
3.1.3.3 PLT relocation 15

4 OBJECT FILES 16

4.1 Introduction 16
4.1.1 Registered Vendor Names 16

4.2 ELF Header 17
4.2.1 ELF Identification 18

4.3 Sections 18
4.3.1 Special Section Indexes 18
4.3.2 Section Types 18
4.3.3 Section Attribute Flags 19
4.3.3.1 Merging of objects in sections with SHF_MERGE 19
4.3.4 Special Sections 19
4.3.5 Section Alignment 20
4.3.6 Build Attributes 20
4.3.6.1 Syntactic structure 20
4.3.6.2 Top level structure tags 21

4.4 String Table **22**

4.5 Symbol Table **22**
4.5.1 Weak Symbols 22
4.5.1.1 Weak References 22
4.5.1.2 Weak Definitions 22
4.5.2 Symbol Types 22
4.5.3 Symbol Values 22
4.5.4 Symbol names 23
4.5.4.1 Reserved symbol names 23
4.5.5 Mapping symbols 24
4.5.5.1 Section-relative mapping symbols 24
4.5.5.2 Absolute mapping symbols 24

4.6 Relocation **24**
4.6.1 Relocation codes 25
4.6.1.1 Addends and PC-bias compensation 25
4.6.1.2 Relocation types 25
4.6.1.3 Static Data relocations 30
4.6.1.4 Static ARM relocations 30
4.6.1.5 Static Thumb16 relocations 34
4.6.1.6 Static Thumb32 relocations 35
4.6.1.7 Static miscellaneous relocations 37
4.6.1.8 Proxy generating relocations 37
4.6.1.9 Relocations for thread-local storage 38
4.6.1.10 Dynamic relocations 39
4.6.1.11 Deprecated relocations 40
4.6.1.12 Obsolete relocations 40
4.6.1.13 Private relocations 40
4.6.1.14 Unallocated relocations 40
4.6.2 Idempotency 40

5 PROGRAM LOADING AND DYNAMIC LINKING **42**

5.1 Introduction **42**

5.2 Program Header **42**
5.2.1 Platform architecture compatibility data 42
5.2.1.1 Platform architecture compatibility data (ABI format) 44

5.3 Program Loading **44**

5.4 Dynamic Linking **44**
5.4.1 Dynamic Section 44

5.5 Post-Link Processing **45**
5.5.1 Production of BE-8 images 45

APPENDIX A SPECIMEN CODE FOR PLT SEQUENCES **46**

A.1 DLL-like, single address space, PLT linkage **46**

A.2 DLL-like, multiple virtual address space, PLT linkage **46**

A.3 SVr4 DSO-like PLT linkage 47

A.4 SVr4 executable-like PLT linkage 47

APPENDIX B CONVENTIONS FOR SYMBOLS CONTAINING $ 48

B.1 Base, Length and Limit symbols 48

B.2 Sub-class and Super-class Symbols 48

B.3 Symbols for Veneering and Interworking Stubs 48

1 ABOUT THIS DOCUMENT

1.1 Change control

1.1.1 Current status and anticipated changes

This document supersedes ARM ELF, Document Number SWS ESPC 0003 B-02.

Anticipated changes to this document include:

- Typographical corrections.
- Clarifications.
- Compatible extensions.

1.1.2 Change history

Issue	Date	By	Change
1.0	24th March 2005	RE	First public release.
1.01	5th July 2005	LS	Defined in §4.3.2, 4.3.4 SHT_ARM_PREEMPTMAP; corrected the erroneous value of SHT_ARM_ATTRIBUTES.
1.02	6th January 2006	RE	Minor correction to definition of e_entry (§4.2).Clarified restrictions on local symbol removal in relocatable files (§4.5.4).Clarified the definition of R_ARM_RELATIVE when S = 0 (§4.6.1.10). Added material describing architecture compatibility for executable files (§5.2.1).
1.03	5th May 2006	RE	Clarified that bit[0] of [e_entry] controls the instruction set selection on entry. Added rules governing SHF_MERGE optimizations (§4.3.3.1). Added material describing initial addends for REL-type relocations (§4.6.1.1).
1.04	25th January 2007	RE	In §4.6 corrected the definition of R_ARM_ALU_(PC|SB)_G*n*_NC, R_ARM_THM_PC8, R_ARM_THM_PC12, and R_ARM_THM_ALU_PREL_11_0. Added a table of 32-bit thumb relocations. In §4.6.1.2 and §4.6.1.9, added new relocations to support an experimental Linux TLS addressing model In §5.2.1 reduced the field masked by PT_ARM_ARCHEXT_ARCHMSK to 8 bits (no current value exceeds 4 bits).
1.05	25th September 2007	RE	Correct definition of Pa in §4.6.1.2 (the bit-mask was incorrect). Corrected spelling of TLS relocations in §4.6.1.9.
A	25th October 2007	LS	Document renumbered (formerly GENC-003538 v1.05).
B	2nd April 2008	RE	Corrected error in Table 4-14 where instructions for R_ARM_THM_PC12 and R_ARM_THM_ALU_PREL_11_0 had been transposed.
C	10th October 2008	RE / LS	In §4.6.1.4, specified which relocations are permitted to generate veneers corrupting *ip*. In §4.6.1.10 specified the meaning of dynamic relocations R_ARM_TLS_DTPMOD32 and R_ARM_TLS_TPOFF32 when the symbol is NULL. Reserved vendor-specific section numbers and names to the [DBGOVL] ABI extension. Clarified use of the symbol by R_ARM_V4BX.
D	28th October 2009	LS	Added http://infocenter.arm.com/ references to the recently published [ARM

			ARM] and the [ARMv5 ARM]; in §4.6.1.6 (Thumb relocations) cross-referenced permitted veneer-generation. In §4.6.1.5, Table 4-13, extended R_ARM_THM_PC8 to ADR as well as LDR(literal). Updated and tidied §5.2.1 and added §5.2.1.1 as a *proposal* for recording executable file attributes.
E	30th November 2012	AC	In §4.2 Table 4-2, added ELF header e_flags to indicate floating point PCS conformance and a mask for legacy bits. In §4.6, standardized instruction descriptions to use ARM ARM terminology. In §4.6.1.1, clarified initial addend formulation for `MOVW`/`MOVT` and `R_ARM_THM_PC8`. In §4.6.1.2 Table 4-9, reserved relocation 140 for a specific future use. In §4.6.1.4, Table 4-12, added entries for `MOVW` and `MOVT`; in subsection Call and Jump Relocations: grouped `R_ARM_THM_CALL` with the other Thumb relocations, and in the final paragraph changed the behaviour of jump relocations to unresolved weak references to be implementation-defined rather than undefined. In §4.6.1.5, Table 4-13, added Overflow column. In §4.6.1.6, Table 4-14, corrected Result Mask for `R_ARM_THM_PC12`; added Table 4-15 *Thumb relocation actions by instruction type*; corrected final paragraph to clarify the cross-reference to call and jump relocations. In §4.6.1.2, §4.6.1.6, §4.6.1.8, added `R_ARM_THM_GOT_BREL12`. In §4.6.1.10, Table 4-18, clarified the wording for `R_ARM_RELATIVE`. In §5.2.1.1, corrected off-by-one error in size of *array*.
F	24th November 2015	CR	§4.6.1.2, Table 4-9, Changed the subdivisions within the reserved/unallocated relocation space (136-255). Renumbered `R_ARM_IRELATIVE` from 140 to 160 (the number agreed with stakeholders; publication as 140 was incorrect). In §4.6.1.4 Table 4-11, removed incorrect overflow check on `R_ARM_MOVT_ABS`, `R_ARM_MOVT_PREL` and `R_ARM_MOVT_BREL`. Clarified in §4.6.1.2 that relocation expression values are computed mod 2^32. In §4.6, added `R_ARM_THM_ALU_ABS_Gn[_NC]` relocations. In §4.3.3, added `SHF_ARM_NOREAD` processor specific section attribute flag.

1.2 References

This document refers to, or is referred to by, the documents listed in the following table.

Ref	Reference	Title
AAELF		ELF for the ARM Architecture (*This document*).
AAPCS		Procedure Call Standard for the ARM Architecture.
BSABI		ABI for the ARM Architecture (Base Standard)
EHABI		Exception Handling ABI for the ARM Architecture
ABI-addenda		Addenda to the ABI for the ARM Architecture
DBGOVL		Support for Debugging Overlaid Programs

Ref	Reference	Title
ARM ARM	(From http://infocenter.arm.com/help/index.jsp, via links ARM architecture, Reference manuals) (Registration required)	ARM DDI 0406: ARM Architecture Reference Manual ARM v7-A and ARM v7-R edition ARM DDI 0403C: ARMv7-M Architecture Reference Manual
ARMv5 ARM	(As for ARM ARM; no registration needed)	ARM DDI 0100I: ARMv5 Architecture Reference Manual
GDWARF	http://dwarfstd.org/Dwarf3Std.php	DWARF 3.0, the generic debug table format
LSB	http://www.linuxbase.org/	Linux Standards Base
SCO-ELF	http://www.sco.com/developers/gabi/2003-12-17/contents.html	System V Application Binary Interface – DRAFT – 17 December 2003
SYM-VER	http://www.akkadia.org/drepper/symbol-versioning	GNU Symbol Versioning

1.3 Terms and abbreviations

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

Term	Meaning
AAPCS	Procedure Call Standard for the ARM Architecture
ABI	Application Binary Interface: 1. The specifications to which an executable must conform in order to execute in a specific execution environment. For example, the *Linux ABI for the ARM Architecture*. 2. A particular aspect of the specifications to which independently produced relocatable files must conform in order to be statically linkable and executable. For example, the *C++ ABI for the ARM Architecture*, the *Run-time ABI for the ARM Architecture*, the *C Library ABI for the ARM Architecture.*
AEABI	(Embedded) ABI for the ARM architecture (*this* ABI…)
ARM-based	… based on the ARM architecture …
core registers	The general purpose registers visible in the ARM architecture's programmer's model, typically r0-r12, SP, LR, PC, and CPSR.
EABI	An ABI suited to the needs of embedded, and deeply embedded (sometimes called *free standing*), applications.
Q-o-I	Quality of Implementation – a quality, behavior, functionality, or mechanism not required by this standard, but which might be provided by systems conforming to it. Q-o-I is often used to describe the tool-chain-specific means by which a standard requirement is met.
VFP	The ARM architecture's Floating Point architecture and instruction set. In this ABI, this abbreviation includes all floating point variants regardless of whether or not vector (V) mode is supported.

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 SCOPE

This specification provides the processor-specific definitions required by ELF [SCO-ELF] for ARM based systems.

The ELF specification is part of the larger System V ABI specification where it forms chapters 4 and 5. However, the specification can be used in isolation as a generic object and executable format.

Section 3 of this document covers ELF related matters that are platform specific. Most of this material is related to the Base Platform ABI.

Sections 4 and 5 of this document are structured to correspond to chapters 4 and 5 of the ELF specification. Specifically:

- Section 4 covers object files and relocations
- Section 5 covers program loading and dynamic linking.

There are several drafts of the ELF specification on the SCO web site. This specification is based on the December 2003 draft, which was the most recent stable draft at the time this specification was developed.

3 PLATFORM STANDARDS

3.1 Base Platform ABI (BPABI)

The BPABI is an abstract platform standard. Platforms conforming to the BPABI can generally share a common toolchain with minimal post-processing requirements.

3.1.1 Symbol Versioning

The BPABI uses the GNU-extended Solaris symbol versioning mechanism [SYM-VER].

Concrete data structure descriptions can be found in /usr/include/sys/link.h (Solaris), /usr/include/elf.h (Linux), in the Linux base specifications [LSB], and in Drepper's paper [SYM-VER]. Drepper provides more detail than the summary here.

An object or executable file using symbol versioning shall set the `EI_OSABI` field in the ELF header to `ELFOSABI_ARM_AEABI` or some other appropriate operating-system specific value.

3.1.1.1 Symbol versioning sections

Symbol versioning adds three sections to an executable file (under the SVr4 ABI these are included in the RO program segment). Each section can be located via a `DT_`*`xxx`* entry in the file's dynamic section.

- The version definitions section. This section defines:
  - The symbol versions associated with symbols exported from this executable file.
  - The version of the file itself.
- The version section.

  This section extends the dynamic symbol table with an extra Elf32_Half field for each symbol. The N^{th} entry gives the index in the virtual *table of versions* (described below) of the version associated with the N^{th} symbol.
- The versions needed section.

  This section describes the versions referred to by symbols not defined in this executable file. Each entry names a DSO and points to a list of versions needed from it. In effect this represents `FROM` *`DSO`* `IMPORT` *`Ver1`*`,` *`Ver2`*`,` …. This section provides a record of the symbol bindings used by the static linker when the executable file was created.

In standard ELF style, both the version definitions section and the versions needed section identify (via the `sh_link` field in their section headers) a string table section (often `.dynstr`) containing the textual values they refer to.

The (virtual) table of versions

When an executable file uses symbol versioning there is also a virtual *table of versions*. This is not represented in the file (there is no corresponding file component). It contains a row for each distinct version defined by, and needed by, this file.

Each version defined, and each version needed, by this file carries its row index in this virtual table, so the table can be constructed on demand. Indexes 2, 3, 4, and so on, are local to this file. Indexes 0 and 1 have predefined global meanings, as do indexes with the top bit (0x8000) set.

3.1.1.2 Locating symbol versioning sections

The version definition section can be located via keys in the dynamic section, as follows.

```
DT_VERDEF      (0x6FFFFFFC),     address
DT_VERDEFNUM   (0x6FFFFFFD),     count
```

This key pair identifies the head and length, of a list of version definitions exported from this executable file. The list is not contiguous – each member points to its successor.

The versions needed section can be located via keys in the dynamic section, as follows.

```
DT_VERNEED     (0x6FFFFFFE),     address
DT_VERNEEDNUM  (0x6FFFFFFF),     count
```

This key pair identifies the head and length of a list of needed versions. Each list member identifies a DSO imported from, and points to a sub-list of versions used by symbols imported from that DSO at the time this executable file was created by the static linker. Neither list need be contiguous – each member points to its successor.

The version section can be located via a key in the dynamic section, as follows.

```
DT_VERSYM      (0x6FFFFFF0),     address
```

The version section adds a field to each dynamic symbol that identifies the version of that symbol's definition, or the version of that symbol needed to satisfy that reference. The number of entries must be same as the number of entries in the dynamic symbol table identified by `DT_SYMTAB` and `DT_HASH` (and by the ARM-specific tag `DT_ARM_SYMTABSZ`).

3.1.1.3 Version definition section

The version definition section has the name `.`*XXX*`_verdef` and the section type `SHT_`*XXX*`_verdef` (the names vary but the section type – `0x6FFFFFFD` – is the same for Solaris and Linux). Its `sh_link` field identifies the string table section (often `.dynstr`) it refers to.

The version definition section defines a set of versions exported from this file and the successor relationships among them.

Each version has a textual name, and two versions are the same if their names compare equal. Textual names are represented by offsets into the associated string table section. Names that must be processed during dynamic linking are also hashed using the standard ELF hash function [SCO-ELF].

Each version definition is linked to the next version definition via it vd_next field which contains the byte offset from the start of this version definition to the start of the next one. Zero marks the end of the list.

Each symbol exported from this shared object refers, via an index in the version section, to one of these version definitions. If bit 15 of the index is set, the symbol is hidden from static binding because it has an old version.

During static linking against this shared object, an undefined symbol can only match an identically named `STB_GLOBAL` definition which refers to one of these version definitions via an index with bit 15 clear.

Each top-level version definition links via its `vd_aux` field to a list of version names. Each link contains the byte offset between the start of the structure containing it and the start of the structure linked to. Zero marks the end of the list. The first member of the list names the latest version, hashed in the version definition's `vd_hash` field. Subsequent members name predecessor versions, but these are irrelevant to both static and dynamic linking.

3.1.1.4 Symbol version section

The symbol version section has the name `.`*XXX*`_versym` and the section type `SHT_`*XXX*`_versym` (the names vary but the section type – `0x6FFFFFFF` – is the same for Solaris and Linux).

The symbol version section is a table of ELF32_Half values. The N^{th} entry in the section corresponds to the N^{th} symbol in the dynamic symbol table.

- 0 if the symbol is local to this executable file.
- 1 if the symbol is undefined and unbound (to be bound dynamically), or if the symbol is defined and names the version of the executable file (usually a shared object) itself.
- The index (> 1) of the corresponding version definition, or version needed, in the virtual table of versions (described in §3.1.1.1).

  This is the same value as is stored in the `vd_ndx` field of a version definition structure and the `vna_other` field of a version needed auxiliary structure.

  Bit 15 of the index is set to denote that this is an old version of the symbol. Such symbols are not used during static binding, but may be linked to during dynamic linking.

3.1.1.5 Versions needed section

The versions needed section has the name `.`*`XXX`*`_verneed` and the section type `SHT_`*`XXX`*`_verneed` (the names vary but the section type – `0x6FFFFFFE` – is the same for Solaris and Linux). Its `sh_link` field identifies the string table section (often `.dynstr`) it refers to.

The versions needed section contains a list of needed DSOs, and the symbol versions needed from them.

Within each version needed structure, the `vn_file` field is the offset in the associated string section of the `SONAME` of the needed DSO, and the `vn_next` field contains the byte offset from the start of this version needed structure to the start of its successor.

Each version needed structure links to a sub-list of needed versions via a byte offset to the start of the first member in its `vn_aux` field. In effect this represents `FROM` *`DSO`* `IMPORT` *`Ver1`*`,` *`Ver2`*`, …`

Each version needed auxiliary structure contains its index in the virtual table of versions in its `vna_other` field. The `vna_name` field contains the offset in the associated string table of the name of the required version.

3.1.2 Symbol Pre-emption in DLLs

Under SVr4, symbol pre-emption occurs at dynamic link time, controlled by the dynamic linker, so there is nothing to encode in a DSO.

In the DLL-creating tool flow, pre-emption happens off line and must be recorded in a BPABI executable file in a form that can be conveniently processed by a post linker. If there is to be any pre-emption when a process is created, what to do must be recorded in the platform executable produced by the post linker.

3.1.2.1 Pre-emption Map Format

Static preemption data is recorded in a special section in the object file. The map is recorded in the dynamic section with the tag `DT_ARM_PREEMPTMAP`, which contains the virtual address of the map.

In the section view, the pre-emption map special section is called `.ARM.preemptmap`. It has type `SHT_ARM_PREEMPTMAP`. In common with other sections that refer to a string table, its `sh_link` field contains the section index of an associated string table.

The map contains a sequence of entries of the form:

```
Elf32_Word count               // Count of pre-empted definitions following
Elf32_Word symbol-name         // Offset in the associated string table
Elf32_Word pre-empting-DLL     // Offset in the associated string table
Elf32_Word pre-empted-DLL      // Offset in the associated string table
…                              //
```

The map is terminated by a count of zero.

If `count` is non-zero, the next two words identify the name of the symbol being pre-empted and the name (`SONAME`) of the executable file providing the pre-empting definition. This structure is followed by `count` words each of which identifies the `SONAME` of an executable file whose definition of `symbol-name` is pre-empted.

`Symbol-name` is the offset in the associated string table section of a NUL-terminated byte string (NTBS) that names a symbol defined in a dynamic symbol table. This value must not be `0`.

Each of `pre-empting-DLL` and `pre-empted-DLL` is an offset in the associated string table section of an NTBS naming a DLL. The name used is the shared object name (`SONAME`) cited by `DT_NEEDED` dynamic tags. The root executable file does not have a `SONAME`, so its name is encoded as `0`.

3.1.3 PLT Sequences and Usage Models

3.1.3.1 Symbols for which a PLT entry must be generated

A PLT entry implements a long-branch to a destination outside of this executable file. In general, the static linker knows only the name of the destination. It does not know its address or instruction-set state. Such a location is called an *imported* location or *imported* symbol.

Some targets (specifically SVr4-based DSOs) also require functions *exported* from an executable file to have PLT entries. In effect, exported functions are treated as if they were imported, so that their definitions can be overridden (pre-empted) at dynamic link time.

A linker must generate a PLT entry for each *candidate* symbol cited by a BL-class relocation directive.

- For an SVr4-based DSO, each `STB_GLOBAL` symbol with `STV_DEFAULT` visibility is a candidate.
- For all other platforms conforming to this ABI, only non-`WEAK`, not hidden (by `STV_HIDDEN`), undefined, `STB_GLOBAL` symbols are candidates.

Note When targeting DLL-based and bare platforms, relocations that cite `WEAK` undefined symbols must be performed by the static linker using the appropriate NULL value of the relocation. No `WEAK` undefined symbols are copied to the dynamic symbol table. `WEAK` definitions may be copied to the dynamic table, but it is Q-o-I whether a dynamic linker will take any account of the `WEAK` attribute. In contrast, SVr4-based platforms process `WEAK` at dynamic link time.

3.1.3.2 Overview of PLT entry code generation

A PLT entry must be able to branch any distance to either instruction-set state. The span and state are fixed when the executable is linked dynamically. A PLT entry must therefore end with code similar to the following.

<table>
<tr><th>ARM V5 and later</th><th>ARM V4T</th></tr>
<tr><td><code>LDR pc, Somewhere</code></td><td><code>LDR ip, Somewhere</code>
<code>BX ip</code></td></tr>
<tr><td colspan="2"><code>Somewhere: DCD   Destination</code></td></tr>
</table>

Note There is no merit in making the final step PC-relative. A location must be written at dynamic link time and at that time the target address must be known [even if dynamic linking is performed off line]. Similarly, it is generally pointless trying to construct a PLT entry entirely in 16-bit Thumb instructions. Even with the overhead of an inline Thumb-to-ARM state change, an ARM-state entry is usually smaller and always faster.

The table below summarizes the code generation variants a static linker must support. *PLT* refers to the read-only component of the veneer and *PLTGOT* to the corresponding writable function pointer.

Table 3-1, PLT code generation options

Platform family	**Neither ROM replaceable nor free of dynamic relocations**	**ROM replaceable, or PLT is free of dynamic relocations**

DLL-like, single address space (Palm OS-like)	PLT code loads a function pointer from the PLT, for example:
	`      LDR pc, LX,`
	`LX    DCD R_ARM_GLOB_DAT(X)`
		PLT code loads the PLTGOT entry SB-relative (§A.1)

DLL-like, multiple virtual address spaces (Symbian OS-like)	PLT code loads a function pointer from the PLT (code and dynamic relocation as shown above).	**PLT code loads the PLTGOT entry via an address constant in the PLT (§A.2)**
SVr4-like (Linux-like)	**Not applicable, but as above if it were.**	**PLT code loads the PLTGOT entry PC-relative (§A.3)**

Following subsections present specimen ARM code sequences appropriate to the right hand column. In each case simplification to the direct (no PLTGOT) case is shown in the left hand column.

Note also that:

- In each case we assume ARM architecture V5 or later, and omit the 4-byte Thumb-to-ARM prelude that is needed to support Thumb-state callers.
- Under ARM architecture V4T, in the two DLL cases shown in the first column above, the final `LDR pc, …`, can be replaced by `LDR ip, …; BX ip`.
- In the case of SVr4 linkage there is an additional constraint to support incremental dynamic linking, namely that `ip` must address the corresponding PLTGOT entry. This constraint is most easily met under architecture V4T by requiring DSOs to be entered in ARM state (but more complex solutions are possible).
- Other platforms are free to impose the same constraint if they support incremental dynamic linking.

3.1.3.3 PLT relocation

A post linker may need to distinguish PLTGOT-generating relocations from GOT-generating ones.

If the static linker were generating a relocatable ELF file it would naturally generate the PLT into its own section (`.plt`, say), subject to relocations from a corresponding relocation section (`.rel.plt` say). No other GOT-generating relocations can occur in `.rel.plt`, so that section would contain all the PLTGOT-generating relocations. By the usual collation rules of static linking, in a subsequent executable file-producing link step those relocations would end up in a contiguous sub-range of the dynamic relocation section.

The ELF standard requires that the GOT-generating relocations of the PLT are emitted into a contiguous sub-range of the dynamic relocation section. That sub-range is denoted by the standard tags `DT_JMPREL` and `DT_PLTRELSZ`. The type of relocations (`REL` or `RELA`) is stored in the `DT_PLTREL` tag.

4 OBJECT FILES

4.1 Introduction

4.1.1 Registered Vendor Names

Various symbols and names may require a vendor-specific name to avoid the potential for name-space conflicts. The list of currently registered vendors and their preferred short-hand name is given in *Table 4-1, Registered Vendors*. Tools developers not listed are requested to co-ordinate with ARM to avoid the potential for conflicts.

Table 4-1, Registered Vendors

Name	**Vendor**
`ADI`	Analog Devices
`acle`	Reserved for use by ARM C *Language* Extensions.
`aeabi`	Reserved to the ABI for the ARM Architecture (EABI pseudo-vendor)
`AnonXyz` `anon`*`Xyz`*	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
`FSL`	Freescale Semiconductor Inc.
`GHS`	Green Hills Systems
`gnu`	GNU compilers and tools (Free Software Foundation)
`iar`	IAR Systems
`icc`	ImageCraft Creations Inc (*ImageCraft C Compiler*)
`intel`	Intel Corporation
`ixs`	Intel Xscale
`llvm`	The LLVM/Clang projects
`PSI`	PalmSource Inc.
`RAL`	Rowley Associates Ltd
`somn`	SOMNIUM Technologies Limited.
`TASKING`	Altium Ltd.
`TI`	TI Inc.
`tls`	Reserved for use in thread-local storage routines.
`WRS`	Wind River Systems.

To register a vendor prefix with ARM, please E-mail your request to `arm.eabi` at `arm.com`.

4.2 ELF Header

The ELF header provides a number of fields that assist in interpretation of the file. Most of these are specified in the base standard. The following fields have ARM-specific meanings.

e_type

There are currently no ARM-specific object file types. All values between ET_LOPROC and ET_HIPROC are reserved to future revisions of this specification.

e_machine

An object file conforming to this specification must have the value EM_ARM (40, 0x28).

e_entry

The value stored in this field is treated like any other code pointer. Specifically, if bit[0] is 0b1 then the entry point contains Thumb code; while bit[1:0] = 0b00 implies that the entry point contains ARM code. The combination bit[1:0] = 0b10 is reserved.

The base ELF specification requires this field to be zero if an application does not have an entry point. Nonetheless, some applications may require an entry point of zero (for example, via the reset vector).

A platform standard may specify that an executable file always has an entry point, in which case e_entry specifies that entry point, even if zero.

e_flags

The processor-specific flags are shown in *Table 4-2, ARM-specific e_flags*. Unallocated bits, and bits allocated in previous versions of this specification, are reserved to future revisions of this specification.

Table 4-2, ARM-specific e_flags

Value	Meaning
EF_ARM_ABIMASK (0xFF000000) (current version is 0x05000000)	This masks an 8-bit version number, the version of the ABI to which this ELF file conforms. This ABI is version 5. A value of 0 denotes unknown conformance.
EF_ARM_BE8 (0x00800000)	The ELF file contains BE-8 code, suitable for execution on an ARM Architecture v6 processor. This flag must only be set on an executable file.
EF_ARM_GCCMASK (0x00400FFF)	Legacy code (ABI version 4 and earlier) generated by gcc-arm-xxx might use these bits.
EF_ARM_ABI_FLOAT_HARD (0x00000400) (ABI version 5 and later)	Set in executable file headers (e_type = ET_EXEC or ET_DYN) to note that the executable file was built to conform to the hardware floating-point procedure-call standard. Compatible with legacy (pre version 5) gcc use as EF_ARM_VFP_FLOAT.
EF_ARM_ABI_FLOAT_SOFT (0x00000200) (ABI version 5 and later)	Set in executable file headers (e_type = ET_EXEC or ET_DYN) to note *explicitly* that the executable file was built to conform to the software floating-point procedure-call standard (the *base standard*). If both EF_ARM_ABI_FLOAT_XXXX bits are clear, conformance to the base procedure-call standard is implied. Compatible with legacy (pre version 5) gcc use as EF_ARM_SOFT_FLOAT.

4.2.1 ELF Identification

The 16-byte ELF identification (e_ident) provides information on how to interpret the file itself. The following values shall be used on ARM systems

EI_CLASS

An ARM ELF file shall contain ELFCLASS32 objects.

EI_DATA

This field may be either ELFDATA2LSB or ELFDATA2MSB. The choice will be governed by the default data order in the execution environment. On an Architecture v6 processor operating in BE8 mode all instructions are in little-endian format. An executable image suitable for operation in this mode will have EF_ARM_BE8 set in the e_flags field.

EI_OSABI

This field shall be zero unless the file uses objects that have flags which have OS-specific meanings (for example, it makes use of a section index in the range SHN_LOOS through SHN_HIOS). There is currently one processor-specific values for this field, defined in *Table 4-3 ARM-specific EI_OSABI values*.

Table 4-3 ARM-specific EI_OSABI values

Value	Meaning
ELFOSABI_ARM_AEABI (64)	The object contains symbol versioning extensions as described in §3.1.1 *Symbol Versioning*.

4.3 Sections

4.3.1 Special Section Indexes

There are no processor-specific special section indexes defined. All processor-specific values are reserved to future revisions of this specification.

4.3.2 Section Types

The defined processor-specific section types are listed in *Table 4-4, Processor specific section types*. All other processor-specific values are reserved to future revisions of this specification.

Table 4-4, Processor specific section types

Name	Value	Comment
SHT_ARM_EXIDX	0x70000001	Exception Index table
SHT_ARM_PREEMPTMAP	0x70000002	BPABI DLL dynamic linking pre-emption map
SHT_ARM_ATTRIBUTES	0x70000003	Object file compatibility attributes
SHT_ARM_DEBUGOVERLAY	0x70000004	See ***DBGOVL*** for details
SHT_ARM_OVERLAYSECTION	0x70000005	

Pointers in sections of types `SHT_INIT_ARRAY`, `SHT_PREINIT_ARRAY` and `SHT_FINI_ARRAY` shall be expressed either as absolute values or relative to the address of the pointer; the choice is platform defined. In object files the relocation type `R_ARM_TARGET1` may be used to indicate this target-specific relocation processing.

`SHT_ARM_EXIDX` marks a section containing index information for exception unwinding. See ***EHABI*** for details.

`SHT_ARM_PREEMPTMAP` marks a section containing a BPABI DLL dynamic linking pre-emption map. See §3.1.2.1, *Pre-emption Map Format*.

`SHT_ARM_ATTRIBUTES` marks a section containing object compatibility attributes. See §4.3.6 *Build Attributes*.

4.3.3 Section Attribute Flags

The defined processor-specific section attribute flags are listed in *Table 4-5, Processor specific section attribute flags*. All other processor-specific values are reserved to future revisions of this specification.

Table 4-5, Processor specific section attribute flags

Name	Value	Comment
`SHF_ARM_NOREAD`	`0x20000000`	The content of this section should not be read by program executor

If all the sections contained by a segment have the SHF_ARM_NOREAD section attribute set, the PF_R attribute should be unset in the program header for the segment.

4.3.3.1 Merging of objects in sections with SHF_MERGE

In a section with the SHF_MERGE flag set, duplicate used objects may be merged and unused objects may be removed. An object is *used* if:

- A relocation directive addresses the object via the section symbol with a suitable addend to point to the object.
- A relocation directive addresses a symbol within the section. *The used object is the one addressed by the symbol irrespective of the addend used.*

4.3.4 Special Sections

Table 4-6, ARM special sections lists the special sections defined by this ABI.

Table 4-6, ARM special sections

Name	Type	Attributes
`.ARM.exidx*`	`SHT_ARM_EXIDX`	`SHF_ALLOC + SHF_LINK_ORDER`
`.ARM.extab*`	`SHT_PROGBITS`	`SHF_ALLOC`
`.ARM.preemptmap`	`SHT_ARM_PREEMPTMAP`	`SHF_ALLOC`
`.ARM.attributes`	`SHT_ARM_ATTRIBUTES`	none
`.ARM.debug_overlay`	`SHT_ARM_DEBUGOVERLAY`	none
`.ARM.overlay_table`	`SHT_ARM_OVERLAYSECTION`	See ***DBGOVL*** for details

Names beginning `.ARM.exidx` name sections containing index entries for section unwinding. Names beginning `.ARM.extab` name sections containing exception unwinding information. See [EHABI] for details.

`.ARM.preemptmap` names a section that contains a BPABI DLL dynamic linking pre-emption map. See §3.1.2.1, *Pre-emption Map Format*.

`.ARM.attributes` names a section that contains build attributes. See §4.3.6 *Build Attributes*.

`.ARM.debug_overlay` and `.ARM.overlay_table` name sections used by the *Debugging Overlaid Programs* ABI extension described in [DBGOVL].

Additional special sections may be required by some platforms standards.

4.3.5 Section Alignment

There is no minimum alignment required for a section. However, sections containing thumb code must be at least 16-bit aligned and sections containing ARM code must be at least 32-bit aligned.

Platform standards may set a limit on the maximum alignment that they can guarantee (normally the page size).

4.3.6 Build Attributes

Build attributes are encoded in a section of type `SHT_ARM_ATTRIBUTES`, and name `.ARM.attributes`.

The content of the section is a stream of bytes. Numbers other than subsection sizes are encoded numbers using unsigned LEB128 encoding (ULEB128), DWARF-3 style [GDWARF].

Attributes are divided into sub-sections. Each subsection is prefixed by the name of the vendor. There is one subsection that is defined by the “aeabi” pseudo-vendor and contains general information about compatibility of the object file. Attributes defined in vendor-specific sections are private to the vendor. In a conforming object file the information recorded in a vendor-specific section may be safely ignored if it is not understood.

Most build attributes naturally apply to a whole translation unit; however, others might apply more naturally to a section or to a function (symbol of type `STT_FUNC`). To permit precise description of attributes the syntax permits three granularities of translation at which an attribute can be expressed.

A section inherits the attributes of the file of which it is a component. A symbol definition inherits the attributes of the section in which it is defined. Attributes that cannot apply to the smaller entity are not inherited.

Note Attributes that naturally apply to a translation unit may, nonetheless, end up applying to a section if sections from distinct relocatable files are combined into a single relocatable file by "partial linking". Similar exceptions may occur at the function level through use of #pragma and other Q-o-I tool chain behavior.

Explicit per-section and per-symbol data should be generated only when it cannot be implied by this inheritance. Being explicit is more verbose, and the explicit options are intended to capture exceptions.

4.3.6.1 Syntactic structure

The overall syntactic structure of an attributes section is:

```
<format-version>
[ <section-length> "vendor-name"
     [ <file-tag> <size> <attribute>*
     | <section-tag> <size> <section-number>* 0 <attribute>*
     | <symbol-tag> <size> <symbol-number>* 0 <attribute>*
     ]+
]*
```

Format-version describes the format of the following data. It is a single byte (not ULEB128). This is version 'A' (0x41). This field exists to permit future incompatible changes in format.

Section-length is a 4-byte unsigned integer in the byte order of the ELF file. It contains the length of the vendor-specific data, including the length field itself, the vendor name string and its terminating NUL byte, and the following attribute data. That is, it is the offset from the start of this vendor subsection to the start of the next vendor subsection.

Vendor-name is a NUL-terminated byte string in the style of a C string. Vendor names begining “Anon” or “anon” are reserved to unregistered private use.

Note In general, a `.ARM.attributes` section in a relocatable file will contain a vendor subsection from the "aeabi" pseudo vendor and, optionally, one from the generating tool chain (e.g. "ARM", "gnu", "WRS", etc) as listed in §4.1.1 *Registered Vendor Names*.

It is required that:

- Attributes that record facts about the compatibility of this relocatable file with other relocatable files are recorded in the public "aeabi" subsection.
- Attributes meaningful only to the producer are recorded in the private vendor subsection. These must not affect compatibility between relocatable files unless that is recorded in the "aeabi" subsection using generic compatibility tags.
- Generic compatibility tags must record a "safe" approximation. A tool chain may record more precise information that only that tool chain comprehends.

Note The intent is that a "foreign" tool chain should not mistakenly link incompatible binary files. The consequence is that a foreign tool chain might sometimes refuse to link files that could be safely linked, because their incompatibility has been crudely approximated.

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.

A vendor-attributes subsection may contain any number of sub-subsections. Each records attributes relating to:

- The whole relocatable file. These sub-subsections contain just a list of attributes.
- A set of sections within the relocatable file. These sub-subsections contain a list of section numbers followed by a list of attributes.
- A set of (defined) symbols in the relocatable file. These sub-subsections contain a list of symbol numbers followed by a list of attributes.

A sub-subsection starts with a tag that identifies the type of the sub-subsection (file, section, or symbol), followed by a 4-byte unsigned integer size in the byte-order of the ELF file. The size is the total size of the sub-subsection including the tag, the size itself, and the sub-subsection content.

Both section indexes and defined symbol indexes are non-zero, so a NUL byte ends a string and a list of indexes.

There are no constraints on the order or number of sub-subsections in a vendor subsection. A consumer that needs the data in inheritance order can obtain the file attributes, the section-related attributes, and the symbol-related attributes, by making three passes over the subsection.

A public attribute is encoded as a tag (non zero, ULEB128-encoded followed by a value. A public value is either an enumeration constant (ULEB128-encoded) or a NUL-terminated string.

Some examples of tags and their argument sorts include:

```
Tag_CPU_raw_name <string>  -- 0x04, "ML692000"
Tag_CPU_name     <string>  -- 0x05, "ARM946E-S"
Tag_PCS_R9_use   <uleb128> -- 0x0E, 0x01 (R9 used as SB)
Tag_PCS_config   <uleb128> -- 0x0D, 0x03 (Linux DSO [/fpic] configuration)
```

4.3.6.2 Top level structure tags

The following tags are defined globally


```
Tag_File, (=1), uleb128:byte-size
Tag_Section, (=2), uleb128:byte-size
Tag_Symbol, (=3), uleb128:byte-size
```

4.4 String Table

There are no processor-specific extensions to the string table.

4.5 Symbol Table

There are no processor-specific symbol types or symbol bindings. All processor-specific values are reserved to future revisions of this specification.

4.5.1 Weak Symbols

There are two forms of weak symbol:

- A *weak reference* — This is denoted by `st_shndx=SHN_UNDEF`, `ELF32_ST_BIND()=STB_WEAK`.
- A *weak definition* — This is denoted by `st_shndx!=SHN_UNDEF`, `ELF32_ST_BIND()=STB_WEAK`.

4.5.1.1 Weak References

Libraries are not searched to resolve weak references. It is not an error for a weak reference to remain unsatisfied.

During linking, the value of an undefined weak reference is:

- Zero if the relocation type is absolute
- The address of the place if the relocation type is pc-relative
- The nominal base address if the relocation type is base-relative.

See *§4.6 Relocation* for further details.

4.5.1.2 Weak Definitions

A weak definition does not change the rules by which object files are selected from libraries. However, if a link set contains both a weak definition and a non-weak definition, the non-weak definition will always be used.

4.5.2 Symbol Types

All code symbols exported from an object file (symbols with binding `STB_GLOBAL`) shall have type `STT_FUNC`.

All extern data objects shall have type `STT_OBJECT`. No `STB_GLOBAL` data symbol shall have type `STT_FUNC`.

The type of an undefined symbol shall be `STT_NOTYPE` or the type of its expected definition.

The type of any other symbol defined in an executable section can be `STT_NOTYPE`. The linker is only required to provide interworking support for symbols of type `STT_FUNC` (interworking for untyped symbols must be encoded directly in the object file).

4.5.3 Symbol Values

In addition to the normal rules for symbol values the following rules shall also apply to symbols of type `STT_FUNC`:

- If the symbol addresses an ARM instruction, its value is the address of the instruction (in a relocatable object, the offset of the instruction from the start of the section containing it).
- If the symbol addresses a Thumb instruction, its value is the address of the instruction with bit zero set (in a relocatable object, the section offset with bit zero set).
- For the purposes of relocation the value used shall be the address of the instruction (`st_value & ~1`).

Note This allows a linker to distinguish ARM and Thumb code symbols without having to refer to the map. An ARM symbol will always have an even value, while a Thumb symbol will always have an odd value. However, a linker should strip the discriminating bit from the value before using it for relocation.

4.5.4 Symbol names

A symbol that names a C or assembly language entity should have the name of that entity. For example, a C function called `calculate` generates a symbol called `calculate` (not `_calculate`).

Symbol names are case sensitive and are matched exactly by linkers.

Any symbol with binding STB_LOCAL may be removed from an object and replaced with an offset from another symbol in the same section under the following conditions:

- The original symbol and replacement symbol are not of type STT_FUNC, or both symbols are of type STT_FUNC and describe code of the same execution type (either both ARM or both Thumb).
- All relocations referring to the symbol can accommodate the adjustment in the addend field (it is permitted to convert a REL type relocation to a RELA type relocation).
- The symbol is not described by the debug information.
- The symbol is not a mapping symbol.
- The resulting object, or image, is not required to preserve accurate symbol information to permit decompilation or other post-linking optimization techniques.
- If the symbol labels an object in a section with the SHF_MERGE flag set, the relocation using symbol may be changed to use the section symbol only if the initial addend of the relocation is zero.

No tool is required to perform the above transformations; an object consumer must be prepared to do this itself if it might find the additional symbols confusing.

Note Multiple conventions exist for the names of compiler temporary symbols (for example, ARMCC uses L*xxx.yyy*, while GNU uses .L*xxx*).

4.5.4.1 Reserved symbol names

The following symbols are reserved to this and future revisions of this specification:

- Local symbols (STB_LOCAL) beginning with '$'
- Global symbols (STB_GLOBAL, STB_WEAK) beginning with '`__aeabi_`' (double '_' at start).
- Global symbols (STB_GLOBAL, STB_WEAK) ending with any of '`$$base`', '`$$length`' or '`$$limit`'
- Symbols matching the pattern ${Ven|*other*}${AA|AT|TA|TT}${I|L|S}[$PI]$$*symbol*
- Local symbols (STB_LOCAL) beginning with '`Lib$Request$$`' or '`BuildAttributes$$`'
- Symbols beginning with '`$Sub$$`' or '`$Super$$`'

Note that global symbols beginning with '*__vendor_*' (double '_' at start), where *vendor* is listed in §4.1.1, *Registered Vendor Names*, are reserved to the named vendor for the purpose of providing vendor-specific tool-chain support functions.

Conventions for reserved symbols for which support is not required by this ABI are described in *APPENDIX A, Conventions for Symbols containing $*.

4.5.5 Mapping symbols

A section of an ELF file can contain a mixture of ARM code, Thumb code and data.

There are inline transitions between code and data at literal pool boundaries. There can also be inline transitions between ARM code and Thumb code, for example in ARM-Thumb inter-working veneers.

Linkers, and potentially other tools, need to map images correctly (for example, to support byte swapping to produce a BE-8 image from a BE-32 object file). To support this, a number of symbols, termed mapping symbols appear in the symbol table to denote the start of a sequence of bytes of the appropriate type. All mapping symbols have type `STT_NOTYPE` and binding `STB_LOCAL`. The st_size field is unused and must be zero.

The mapping symbols are defined in *Table 4-7, Mapping symbols*. It is an error for a relocation to reference a mapping symbol. Two forms of mapping symbol are supported:

- a short form, that uses a dollar character and a single letter denoting the class. This form can be used when an object producer creates mapping symbols automatically, and minimizes symbol table space
- a longer form, where the short form is extended with a period and then any sequence of characters that are legal for a symbol. This form can be used when assembler files have to be annotated manually and the assembler does not support multiple definitions of symbols.

Table 4-7, Mapping symbols

Name	Meaning
`$a` `$a.`*<any…>*	Start of a sequence of ARM instructions
`$d` `$d.`*<any…>*	Start of a sequence of data items (for example, a literal pool)
`$t` `$t.`*<any…>*	Start of a sequence of Thumb instructions

4.5.5.1 Section-relative mapping symbols

Mapping symbols defined in a section define a sequence of half-open address intervals that cover the address range of the section. Each interval starts at the address defined by the mapping symbol, and continues up to, but not including, the address defined by the next (in address order) mapping symbol or the end of the section. A section must have a mapping symbol defined at the beginning of the section; however, if the section contains only data then the mapping symbol may be omitted.

4.5.5.2 Absolute mapping symbols

Mapping symbols are no-longer required for the absolute section. The equivalent information is now conveyed by the type of the absolute symbol.

4.6 Relocation

Relocation information is used by linkers in order to bind symbols and addresses that could not be determined when the initial object was generated.

In these descriptions, references in the style `LDR(1)` refer to the *ARMv5 Architecture Reference Manual* [ARMv5 ARM] while those in the style `LDR(`immediate, Thumb`)` give the corresponding reference to the ARM Architecture Reference Manual ARM v7-A and ARM v7-R edition [ARM ARM].

4.6.1 Relocation codes

The relocation codes for ARM are divided into four categories:

- Mandatory relocations that must be supported by all static linkers
- Platform-specific relocations that are required for specific virtual platforms
- Private relocations that are guaranteed never to be allocated in future revisions of this specification, but which must never be used in portable object files.
- Unallocated relocations that are reserved for use in future revisions of this specification.

4.6.1.1 Addends and PC-bias compensation

A binary file may use `REL` or `RELA` relocations or a mixture of the two (but multiple relocations for the same address must use only one type). If the relocation is pc-relative then compensation for the PC bias (the PC value is 8 bytes ahead of the executing instruction in ARM state and 4 bytes in Thumb state) must be encoded in the relocation by the object producer.

Unless specified otherwise, the initial addend for `REL` type relocations is formed according to the following rules.

- If the place is subject to a data-type relocation, the initial value in the place is sign-extended to 32 bits.
- If the place contains an instruction, the immediate field for the instruction is extracted from it and used as the initial addend. If the instruction is a `SUB`, or an `LDR/STR` type instruction with the ‘up’ bit clear, then the initial addend is formed by negating the unsigned immediate value encoded in the instruction.

Some examples are shown in *Table 4-8, Examples of REL format initial addends*.

Table 4-8, Examples of REL format initial addends

Instruction	Relocation	Encoding	Initial Addend
`SUB  R0, R1, #1020`	`R_ARM_ALU_PC_G0`	`0xe2410fff`	`-1020`
`LDR  R0, [R2, #16]`	`R_ARM_LDR_PC_G2`	`0xe59f0010`	`16`
`BL   .`	`R_ARM_THM_CALL`	`0xf7ff, 0xfffe`	`-4`
`DCB  0xf0`	`R_ARM_ABS8`	`0xf0`	`-16`

If the initial addend cannot be encoded in the space available then a `RELA` format relocation must be used.

There are three special cases for forming the initial addend of `REL`-type relocations where the immediate field cannot normally hold small signed integers:

- For relocations processing `MOVW` and `MOVT` instructions (in both ARM and Thumb state), the initial addend is formed by interpreting the 16-bit literal field of the instruction as a 16-bit signed value in the range -32768 <= A < 32768. The interpretation is the same whether the relocated place contains a `MOVW` instruction or a `MOVT` instruction.
- For `R_ARM_THM_JUMP6` the initial addend is formed by the formula `(((imm + 4) & 0x7f) – 4)`, where `imm` is the contatenation of `bit[9]:bit[7:3]:’0’` from the Thumb `CBZ` or `CBNZ` instruction being relocated.
- For `R_ARM_THM_PC8` the initial addend is formed by the formula `(((imm + 4) & 0x3ff) – 4)`, where `imm` is the 32-bit value encoded in the 8-bit place, as defined in the `LDR(3)/LDR(literal)` Thumb instructions section of the ARM ARM.

4.6.1.2 Relocation types

Table 4-9, Relocation codes lists the relocation codes for ARM. The table shows:

- The *code* which is stored in the `ELF32_R_TYPE` component of the `r_info` field.
- The mnemonic *name* for the relocation.
- The type of the relocation. This field substantially divides the relocations into Static and Dynamic relocations. Static relocations are processed by a static linker; they are normally either fully resolved or used to produce dynamic relocations for processing by a post-linking step or a dynamic loader. A well formed image will have no static relocations after static linking is complete, so a post-linker or dynamic loader will normally only have to deal with dynamic relocations. This field is also used to describe deprecated, obsolete, private and unallocated relocation codes. Deprecated codes should not be generated by fully conforming toolchains; however it is recognized that there may be substantial existing code that makes use of these forms, so it is expected that a linker may well be required to handle them at this time. Obsolete codes should not be used, and it is believed that there is little or no common use of these values. All unallocated codes are reserved for future allocation.
- The *class* of the relocation describes the type of place being relocated: these are Data, ARM, Thumb16 and Thumb32 (32-bit long-format instructions). A special class of Miscellaneous is used when the operation is not a simple mathematical expression.
- The *operation* field describes how the symbol and addend are processed by the relocation code. It does not describe how the addend is formed (for a REL type relocation), what overflow checking is done, or how the value is written back into the place: this information is given in subsequent sections. In all cases, relocation expression values are computed mod 2^32.

The following nomenclature is used for the operation:

- S (when used on its own) is the address of the symbol.
- A is the addend for the relocation.
- P is the address of the *place* being relocated (derived from r_offset).
- Pa is the adjusted address of the place being relocated, defined as (P & 0xFFFFFFFC).
- T is 1 if the target symbol S has type `STT_FUNC` and the symbol addresses a Thumb instruction; it is 0 otherwise.
- B(S) is the addressing *origin* of the output segment defining the symbol S. The origin is not required to be the base address of the segment. This value must always be word-aligned.
- GOT_ORG is the addressing origin of the Global Offset Table (the indirection table for imported data addresses). This value must always be word-aligned. See *§4.6.1.8, Proxy generating relocations*.
- GOT(S) is the address of the GOT entry for the symbol S.

Table 4-9, Relocation codes

Code	Name	Type	Class	Operation
0	`R_ARM_NONE`	Static	Miscellaneous	
1	`R_ARM_PC24`	Deprecated	ARM	`((S + A) | T) – P`
2	`R_ARM_ABS32`	Static	Data	`(S + A) | T`
3	`R_ARM_REL32`	Static	Data	`((S + A) | T) – P`
4	`R_ARM_LDR_PC_G0`	Static	ARM	`S + A – P`
5	`R_ARM_ABS16`	Static	Data	`S + A`
6	`R_ARM_ABS12`	Static	ARM	`S + A`
7	`R_ARM_THM_ABS5`	Static	Thumb16	`S + A`
8	`R_ARM_ABS8`	Static	Data	`S + A`
9	`R_ARM_SBREL32`	Static	Data	`((S + A) | T) – B(S)`

Code	Name	Type	Class	Operation
10	R_ARM_THM_CALL	Static	Thumb32	((S + A) | T) – P
11	R_ARM_THM_PC8	Static	Thumb16	S + A – Pa
12	R_ARM_BREL_ADJ	Dynamic	Data	ΔB(S) + A
13	R_ARM_TLS_DESC	Dynamic	Data	
14	*R_ARM_THM_SWI8*	Obsolete	Encodings reserved for future Dynamic relocations	
15	*R_ARM_XPC25*	Obsolete		
16	*R_ARM_THM_XPC22*	Obsolete		
17	R_ARM_TLS_DTPMOD32	Dynamic	Data	Module[S]
18	R_ARM_TLS_DTPOFF32	Dynamic	Data	S + A – TLS
19	R_ARM_TLS_TPOFF32	Dynamic	Data	S + A – tp
20	R_ARM_COPY	Dynamic	Miscellaneous	
21	R_ARM_GLOB_DAT	Dynamic	Data	(S + A) | T
22	R_ARM_JUMP_SLOT	Dynamic	Data	(S + A) | T
23	R_ARM_RELATIVE	Dynamic	Data	B(S) + A [Note: see Table 4-18]
24	R_ARM_GOTOFF32	Static	Data	((S + A) | T) – GOT_ORG
25	R_ARM_BASE_PREL	Static	Data	B(S) + A – P
26	R_ARM_GOT_BREL	Static	Data	GOT(S) + A – GOT_ORG
27	R_ARM_PLT32	Deprecated	ARM	((S + A) | T) – P
28	R_ARM_CALL	Static	ARM	((S + A) | T) – P
29	R_ARM_JUMP24	Static	ARM	((S + A) | T) – P
30	R_ARM_THM_JUMP24	Static	Thumb32	((S + A) | T) – P
31	R_ARM_BASE_ABS	Static	Data	B(S) + A
32	*R_ARM_ALU_PCREL_7_0*	Obsolete	Note – Legacy (ARM ELF B02) names have been retained for these obsolete relocations.	
33	*R_ARM_ALU_PCREL_15_8*	Obsolete		
34	*R_ARM_ALU_PCREL_23_15*	Obsolete		
35	R_ARM_LDR_SBREL_11_0_NC	Deprecated	ARM	S + A – B(S)
36	R_ARM_ALU_SBREL_19_12_NC	Deprecated	ARM	S + A – B(S)
37	R_ARM_ALU_SBREL_27_20_CK	Deprecated	ARM	S + A – B(S)
38	R_ARM_TARGET1	Static	Miscellaneous	(S + A) | T *or* ((S + A) | T) – P
39	R_ARM_SBREL31	Deprecated	Data	((S + A) | T) – B(S)
40	R_ARM_V4BX	Static	Miscellaneous	
41	R_ARM_TARGET2	Static	Miscellaneous	
42	R_ARM_PREL31	Static	Data	((S + A) | T) – P

Code	Name	Type	Class	Operation
43	R_ARM_MOVW_ABS_NC	Static	ARM	(S + A) | T
44	R_ARM_MOVT_ABS	Static	ARM	S + A
45	R_ARM_MOVW_PREL_NC	Static	ARM	((S + A) | T) – P
46	R_ARM_MOVT_PREL	Static	ARM	S + A – P
47	R_ARM_THM_MOVW_ABS_NC	Static	Thumb32	(S + A) | T
48	R_ARM_THM_MOVT_ABS	Static	Thumb32	S + A
49	R_ARM_THM_MOVW_PREL_NC	Static	Thumb32	((S + A) | T) – P
50	R_ARM_THM_MOVT_PREL	Static	Thumb32	S + A – P
51	R_ARM_THM_JUMP19	Static	Thumb32	((S + A) | T) – P
52	R_ARM_THM_JUMP6	Static	Thumb16	S + A – P
53	R_ARM_THM_ALU_PREL_11_0	Static	Thumb32	((S + A) | T) – Pa
54	R_ARM_THM_PC12	Static	Thumb32	S + A – Pa
55	R_ARM_ABS32_NOI	Static	Data	S + A
56	R_ARM_REL32_NOI	Static	Data	S + A – P
57	R_ARM_ALU_PC_G0_NC	Static	ARM	((S + A) | T) – P
58	R_ARM_ALU_PC_G0	Static	ARM	((S + A) | T) – P
59	R_ARM_ALU_PC_G1_NC	Static	ARM	((S + A) | T) – P
60	R_ARM_ALU_PC_G1	Static	ARM	((S + A) | T) – P
61	R_ARM_ALU_PC_G2	Static	ARM	((S + A) | T) – P
62	R_ARM_LDR_PC_G1	Static	ARM	S + A – P
63	R_ARM_LDR_PC_G2	Static	ARM	S + A – P
64	R_ARM_LDRS_PC_G0	Static	ARM	S + A – P
65	R_ARM_LDRS_PC_G1	Static	ARM	S + A – P
66	R_ARM_LDRS_PC_G2	Static	ARM	S + A – P
67	R_ARM_LDC_PC_G0	Static	ARM	S + A – P
68	R_ARM_LDC_PC_G1	Static	ARM	S + A – P
69	R_ARM_LDC_PC_G2	Static	ARM	S + A – P
70	R_ARM_ALU_SB_G0_NC	Static	ARM	((S + A) | T) – B(S)
71	R_ARM_ALU_SB_G0	Static	ARM	((S + A) | T) – B(S)
72	R_ARM_ALU_SB_G1_NC	Static	ARM	((S + A) | T) – B(S)
73	R_ARM_ALU_SB_G1	Static	ARM	((S + A) | T) – B(S)
74	R_ARM_ALU_SB_G2	Static	ARM	((S + A) | T) – B(S)
75	R_ARM_LDR_SB_G0	Static	ARM	S + A – B(S)
76	R_ARM_LDR_SB_G1	Static	ARM	S + A – B(S)

Code	Name	Type	Class	Operation
77	R_ARM_LDR_SB_G2	Static	ARM	S + A – B(S)
78	R_ARM_LDRS_SB_G0	Static	ARM	S + A – B(S)
79	R_ARM_LDRS_SB_G1	Static	ARM	S + A – B(S)
80	R_ARM_LDRS_SB_G2	Static	ARM	S + A – B(S)
81	R_ARM_LDC_SB_G0	Static	ARM	S + A – B(S)
82	R_ARM_LDC_SB_G1	Static	ARM	S + A – B(S)
83	R_ARM_LDC_SB_G2	Static	ARM	S + A – B(S)
84	R_ARM_MOVW_BREL_NC	Static	ARM	((S + A) | T) – B(S)
85	R_ARM_MOVT_BREL	Static	ARM	S + A – B(S)
86	R_ARM_MOVW_BREL	Static	ARM	((S + A) | T) – B(S)
87	R_ARM_THM_MOVW_BREL_NC	Static	Thumb32	((S + A) | T) – B(S)
88	R_ARM_THM_MOVT_BREL	Static	Thumb32	S + A – B(S)
89	R_ARM_THM_MOVW_BREL	Static	Thumb32	((S + A) | T) – B(S)
90	R_ARM_TLS_GOTDESC	Static	Data	
91	R_ARM_TLS_CALL	Static	ARM	
92	R_ARM_TLS_DESCSEQ	Static	ARM	TLS relaxation
93	R_ARM_THM_TLS_CALL	Static	Thumb32	
94	R_ARM_PLT32_ABS	Static	Data	PLT(S) + A
95	R_ARM_GOT_ABS	Static	Data	GOT(S) + A
96	R_ARM_GOT_PREL	Static	Data	GOT(S) + A – P
97	R_ARM_GOT_BREL12	Static	ARM	GOT(S) + A – GOT_ORG
98	R_ARM_GOTOFF12	Static	ARM	S + A – GOT_ORG
99	R_ARM_GOTRELAX	Static	Miscellaneous	
100	R_ARM_GNU_VTENTRY	Deprecated	Data	???
101	R_ARM_GNU_VTINHERIT	Deprecated	Data	???
102	R_ARM_THM_JUMP11	Static	Thumb16	S + A – P
103	R_ARM_THM_JUMP8	Static	Thumb16	S + A – P
104	R_ARM_TLS_GD32	Static	Data	GOT(S) + A – P
105	R_ARM_TLS_LDM32	Static	Data	GOT(S) + A – P
106	R_ARM_TLS_LDO32	Static	Data	S + A – TLS
107	R_ARM_TLS_IE32	Static	Data	GOT(S) + A – P
108	R_ARM_TLS_LE32	Static	Data	S + A – tp
109	R_ARM_TLS_LDO12	Static	ARM	S + A – TLS
110	R_ARM_TLS_LE12	Static	ARM	S + A – tp

Code	Name	Type	Class	Operation
111	R_ARM_TLS_IE12GP	Static	ARM	GOT(S) + A – GOT_ORG
112-127	R_ARM_PRIVATE_<*n*>	Private (n = 0, 1, … 15)		
128	R_ARM_ME_TOO	Obsolete		
129	R_ARM_THM_TLS_DESCSEQ16	Static	Thumb16	
130	R_ARM_THM_TLS_DESCSEQ32	Static	Thumb32	
131	R_ARM_THM_GOT_BREL12	Static	Thumb32	GOT(S) + A – GOT_ORG
132	R_ARM_THM_ALU_ABS_G0_NC	Static	Thumb16	(S + A) | T
133	R_ARM_THM_ALU_ABS_G1_NC	Static	Thumb16	S + A
134	R_ARM_THM_ALU_ABS_G2_NC	Static	Thumb16	S + A
135	R_ARM_THM_ALU_ABS_G3	Static	Thumb16	S + A
136-159		Static	Reserved for future allocation	
160	R_ARM_IRELATIVE	Dynamic	Reserved for future functionality	
161-255		Dynamic	Reserved for future allocation	

4.6.1.3 Static Data relocations

Except as indicated in *Table 4-10, Static Data relocations with non-standard size or processing* all static data relocations have size 4, alignment 1 and write the full 32-bit result to the place; there is thus no need for overflow checking.

The overflow ranges for R_ARM_ABS16 and R_ARM_ABS8 permit either signed or unsigned results. It is therefore not possible to detect an unsigned value that has underflowed by a small amount, or a signed value that has overflowed by a small amount.

Table 4-10, Static Data relocations with non-standard size or processing

Code	Name	Size	REL Addend	Overflow
5	R_ARM_ABS16	2	sign_extend(P[16:0])	-32768 ≤ X ≤ 65535
8	R_ARM_ABS8	1	sign_extend(P[8:0])	-128 ≤ X ≤ 255
42	R_ARM_PREL31	4	sign_extend(P[30:0])	31-bit 2's complement

4.6.1.4 Static ARM relocations

The relocations that can modify fields of an ARM instruction are listed in *Table 4-11, Static ARM instruction relocations*. All relocations in this class relocate a 32-bit aligned ARM instruction by modifying part of the instruction. In most cases the modification is to change the offset, but in some cases the opcode itself may be changed (for example, an ADD may be converted to a SUB and *vice-versa*). In the table:

- X is the 32-bit result of normal relocation processing

- G_n is a mask operation that is instruction dependent. See *Group Relocations* below for rules on how the mask is formed for each case.

Table 4-11, Static ARM instruction relocations

Code	Name	Overflow	Instruction	Result Mask
4	R_ARM_LDR_PC_G0	Yes	LDR, STR LDRB, STRB	ABS(X) & G_0(LDR)
6	R_ARM_ABS12	Yes	LDR, STR	ABS(X) & 0xFFF
28	R_ARM_CALL	Yes	BL/BLX	X & 0x03FFFFFE
29	R_ARM_JUMP24	Yes	B/BL<cond>	X & 0x03FFFFFE
43	R_ARM_MOVW_ABS_NC	No	MOVW	X & 0xFFFF
44	R_ARM_MOVT_ABS	No	MOVT	X & 0xFFFF0000
45	R_ARM_MOVW_PREL_NC	No	MOVW	X & 0xFFFF
46	R_ARM_MOVT_PREL	No	MOVT	X & 0xFFFF0000
57	R_ARM_ALU_PC_G0_NC	No	ADD, SUB	ABS(X) & G_0
58	R_ARM_ALU_PC_G0	Yes	ADD, SUB	ABS(X) & G_0
59	R_ARM_ALU_PC_G1_NC	No	ADD, SUB	ABS(X) & G_1
60	R_ARM_ALU_PC_G1	Yes	ADD, SUB	ABS(X) & G_1
61	R_ARM_ALU_PC_G2	Yes	ADD, SUB	ABS(X) & G_2
62	R_ARM_LDR_PC_G1	Yes	LDR, STR, LDRB, STRB	ABS(X) & G_1(LDR)
63	R_ARM_LDR_PC_G2	Yes	LDR, STR, LDRB, STRB	ABS(X) & G_2(LDR)
64	R_ARM_LDRS_PC_G0	Yes	LDRD, STRD, LDRH, STRH, LDRSH, LDRSB	ABS(X) & G_0(LDRS)
65	R_ARM_LDRS_PC_G1	Yes	LDRD, STRD, LDRH, STRH, LDRSH, LDRSB	ABS(X) & G_1(LDRS)
66	R_ARM_LDRS_PC_G2	Yes	LDRD, STRD, LDRH, STRH, LDRSH, LDRSB	ABS(X) & G_2(LDRS)
67	R_ARM_LDC_PC_G0	Yes	LDC, STC	ABS(X) & G_0(LDC)
68	R_ARM_LDC_PC_G1	Yes	LDC, STC	ABS(X) & G_1(LDC)
69	R_ARM_LDC_PC_G2	Yes	LDC, STC	ABS(X) & G_2(LDC)
70	R_ARM_ALU_SB_G0_NC	No	ADD, SUB	ABS(X) & G_0
71	R_ARM_ALU_SB_G0	Yes	ADD, SUB	ABS(X) & G_0
72	R_ARM_ALU_SB_G1_NC	No	ADD, SUB	ABS(X) & G_1
73	R_ARM_ALU_SB_G1	Yes	ADD, SUB	ABS(X) & G_1

Code	Name	Overflow	Instruction	Result Mask
74	R_ARM_ALU_SB_G2	Yes	ADD, SUB	ABS(X) & G_2
75	R_ARM_LDR_SB_G0	Yes	LDR, STR, LDRB, STRB	ABS(X) & G_0(LDR)
76	R_ARM_LDR_SB_G1	Yes	LDR, STR, LDRB, STRB	ABS(X) & G_1(LDR)
77	R_ARM_LDR_SB_G2	Yes	LDR, STR, LDRB, STRB	ABS(X) & G_2(LDR)
78	R_ARM_LDRS_SB_G0	Yes	LDRD, STRD, LDRH, STRH, LDRSH, LDRSB	ABS(X) & G_0(LDRS)
79	R_ARM_LDRS_SB_G1	Yes	LDRD, STRD, LDRH, STRH, LDRSH, LDRSB	ABS(X) & G_1(LDRS)
80	R_ARM_LDRS_SB_G2	Yes	LDRD, STRD, LDRH, STRH, LDRSH, LDRSB	ABS(X) & G_2(LDRS)
81	R_ARM_LDC_SB_G0	Yes	LDC, STC	ABS(X) & G_0(LDC)
82	R_ARM_LDC_SB_G1	Yes	LDC, STC	ABS(X) & G_1(LDC)
83	R_ARM_LDC_SB_G2	Yes	LDC, STC	ABS(X) & G_2(LDC)
84	R_ARM_MOVW_BREL_NC	No	MOVW	X & 0xFFFF
85	R_ARM_MOVT_BREL	No	MOVT	X & 0xFFFF0000
86	R_ARM_MOVW_BREL	Yes	MOVW	X & 0xFFFF
97	R_ARM_GOT_BREL12	Yes	LDR	ABS(X) & 0xFFF
98	R_ARM_GOTOFF12	Yes	LDR, STR	ABS(X) & 0xFFF
109	R_ARM_TLS_LDO12	Yes	LDR, STR	ABS(X) & 0xFFF
110	R_ARM_TLS_LE12	Yes	LDR, STR	ABS(X) & 0xFFF
111	R_ARM_TLS_IE12GP	Yes	LDR	ABS(X) & 0xFFF

The formation of the initial addend in a REL type relocation for the various instruction classes is described in *Table 4-12, ARM relocation actions by instruction type*. Insn modification describes how the 32-bit result X is written back to the instruction; Result_Mask is the value of X after the masking operation described in Table 4-11 has been applied.

Table 4-12, ARM relocation actions by instruction type

Instruction	REL Addend	Insn modification
BL, BLX	sign_extend (insn[23:0] << 2)	See *call and jump relocations*
B, BL<cond>	sign_extend (insn[23:0] << 2)	See *call and jump relocations*
LDR, STR, LDRB, STRB	insn[11:0] * $-1^{(\text{insn}[23] == 0)}$	insn[23] = (X >= 0) insn[11:0] = Result_Mask(X)
LDRD, STRD, LDRH, STRH, LDRSH, LDRSB	((insn[11:8] << 4) | insn[3:0]) * $-1^{(\text{insn}[23] == 0)}$	insn[23] = (X >= 0) insn[11:0] = Result_Mask(X)

Instruction	REL Addend	Insn modification
LDC, STC	(insn[7:0] << 2) * $-1^{(insn[23] == 0)}$	insn[23] = (X >= 0) insn[7:0] = Result_Mask(X) >> 2
ADD, SUB	Imm(insn) * $-1^{(opcode(insn) == SUB)}$	opcode(insn) = X >= 0 ? ADD : SUB Imm(insn) = Result_Mask(X)
MOVW	See §4.6.1.1	insn[19:16] = Result_Mask(X) >> 12 insn[11:0] = Result_Mask(X) & 0xFFF
MOVT	See §4.6.1.1. The effect permits executing MOVW and later MOVT to create a 32-bit link-time constant in a register.	insn[19:16] = (Result_Mask(X) >> 16) >> 12 insn[11:0] = (Result_Mask(X) >> 16) & 0xFFF

Call and Jump relocations

There is one relocation (R_ARM_CALL) for unconditional function call instructions (BLX and BL with the condition field set to 0xe), and one for jump instructions (R_ARM_JUMP24). The principal difference between the two relocation values is the handling of ARM/Thumb inter-working: on ARM architecture 5 and above, an instruction relocated by R_ARM_CALL that calls a function that is entered in Thumb state may be relocated by changing the instruction to BLX; an instruction relocated by R_ARM_JUMP24 must use a veneer to effect the transition to Thumb state. Conditional function call instructions (BL<cond>) must be relocated using R_ARM_JUMP24.

A linker may use a veneer (a sequence of instructions) to implement the relocated branch if the relocation is one of R_ARM_PC24, R_ARM_CALL, R_ARM_JUMP24, (or, in Thumb state, R_ARM_THM_CALL, R_ARM_THM_JUMP24, or R_ARM_THM_JUMP19) and:

- The target symbol has type STT_FUNC
- Or, the target symbol and relocated place are in separate sections input to the linker

In all other cases a linker shall diagnose an error if relocation cannot be effected without a veneer. A linker generated veneer may corrupt register r12 (IP) and the condition flags, but must preserve all other registers. On M-profile processors a veneer may also assume the presence of a stack with at least 8 bytes (2 words) of memory. Linker veneers may be needed for a number of reasons, including, but not limited to:

- Target is outside the addressable span of the branch instruction (+/- 32Mb)
- Target address and execution state will not be known until run time, or the address might be pre-empted

In some systems indirect calls may also use veneers in order to support dynamic linkage while preserving pointer equivalence.

On platforms that do not support dynamic pre-emption of symbols an unresolved weak reference to a symbol relocated by R_ARM_CALL (or, in Thumb state, R_ARM_THM_CALL) shall be treated as a jump to the next instruction (the call becomes a no-op). The behaviour of R_ARM_JUMP24 and static Thumb jump relocations in these conditions is implementation-defined.

Group relocations

Relocation codes 4 and 57-83 are intended to relocate sequences of instructions that generate a single address. They are encoded to extract the maximum flexibility from the ARM ADD- and SUB-immediate instructions without need to determine during linking the full sequence being used. The relocations operate by performing the basic relocation calculation and then partitioning the result into a set of groups of bits that can be statically determined. All processing for the formation of the groups is done on the absolute value of X; the sign of X is used to determine whether ADD or SUB instructions are used, or, if the sequence concludes with a load/store operation, the setting of the U bit (bit 23) in the instruction.

A group, G_n, is formed by examining the residual value, Y_n, after the bits for group G_{n-1} have been masked off. Processing for group G_0 starts with the absolute value of X. For ALU-type relocations a group is formed by determining the most significant bit (MSB) in the residual and selecting the smallest constant K_n such that

$$MSB(Y_n) \ \& \ (255 \ll 2K_n) \ != 0,$$

except that if Y_n is 0, then K_n is 0. The value G_n is then

$$Y_n \ \& \ (255 \ll 2K_n),$$

and the residual, Y_{n+1}, for the next group is

$$Y_n \ \& \ \sim G_n.$$

Note that if Y_n is 0, then G_n will also be 0.

For group relocations that access memory the residual value is examined in its entirety (i.e. after the appropriate sequence of ALU groups have been removed): if the relocation has not overflowed, then the residual for such an instruction will always be a valid offset for the indicated type of memory access.

Overflow checking is always performed on the highest-numbered group in a sequence. For ALU-type relocations the result has overflowed if Y_{n+1} is not zero. For memory access relocations the result has overflowed if the residual is not a valid offset for the type of memory access.

Note The unchecked (_NC) group relocations all include processing of the Thumb bit of a symbol. However, the memory forms of group relocations (eg R_ARM_LDR_G0) ignore this bit. Therefore the use of the memory forms with symbols of type STT_FUNC is unpredictable.

4.6.1.5 Static Thumb16 relocations

Relocations for 16-bit thumb instructions are shown in *Table 4-13, Static Thumb-16 Relocations*. In general the addressing range of these relocations is too small for them to reference external symbols and they are documented here for completeness. A linker is not required to generate trampoline sequences (or veneers) to extend the branching range of the jump relocations.

Relocation R_ARM_THM_JUMP6 is only applicable to the Thumb-2 instruction set.

Table 4-13, Static Thumb-16 Relocations

Code	Name	Overflow	Instruction	Result Mask
7	R_ARM_THM_ABS5	Yes	LDR(1)/LDR(immediate, Thumb), STR(1)/STR(immediate, Thumb)	X & 0x7C
11	R_ARM_THM_PC8	Yes	LDR(3)/LDR(literal), ADD(5)/ADR	X & 0x3FC
52	R_ARM_THM_JUMP6	Yes	CBZ, CBNZ	X & 0x7E
102	R_ARM_THM_JUMP11	Yes	B(2)/B	X & 0xFFE
103	R_ARM_THM_JUMP8	Yes	B(1)/B<cond>	X & 0x1FE
132	R_ARM_THM_ALU_ABS_G0_NC	No	ADD(2)/ADD (immediate, Thumb, 8-bit immediate), MOV(1)/MOV (immediate)	X & 0x000000FF
133	R_ARM_THM_ALU_ABS_G1_NC	No	ADD(2)/ADD (immediate, Thumb, 8-bit immediate), MOV(1)/MOV (immediate)	X & 0x0000FF00

Code	Name	Overflow	Instruction	Result Mask
134	R_ARM_THM_ALU_ABS_G2_NC	No	ADD(2)/ADD (immediate, Thumb, 8-bit immediate), MOV(1)/MOV (immediate)	X & 0x00FF0000
135	R_ARM_THM_ALU_ABS_G3	No	ADD(2)/ADD (immediate, Thumb, 8-bit immediate), MOV(1)/MOV (immediate)	X & 0xFF000000

4.6.1.6 Static Thumb32 relocations

Relocations for 32-bit Thumb instructions are shown in *Table 4-14, Static Thumb-32 instruction relocations*. With the exception of R_ARM_THM_CALL, these relocations are only applicable to 32-bit Thumb instructions.

Table 4-14, Static Thumb-32 instruction relocations

Code	Name	Overflow	Instruction	Result Mask
10	R_ARM_THM_CALL	Yes	BL	X & 0x01FFFFFE
30	R_ARM_THM_JUMP24	Yes	B.W	X & 0x01FFFFFE
47	R_ARM_THM_MOVW_ABS_NC	No	MOVW	X & 0x0000FFFF
48	R_ARM_THM_MOVT_ABS	No	MOVT	X & 0xFFFF0000
49	R_ARM_THM_MOVW_PREL_NC	No	MOVW	X & 0x0000FFFF
50	R_ARM_THM_MOVT_PREL	No	MOVT	X & 0xFFFF0000
51	R_ARM_THM_JUMP19	Yes	B<cond>.W	X & 0x001FFFFE
53	R_ARM_THM_ALU_PREL_11_0	Yes	ADR.W	X & 0x00000FFF
54	R_ARM_THM_PC12	Yes	LDR<,B,SB,H,SH> (literal)	ABS(X) & 0x00000FFF
87	R_ARM_THM_MOVW_BREL_NC	No	MOVW	X & 0x0000FFFF
88	R_ARM_THM_MOVT_BREL	No	MOVT	X & 0xFFFF0000
89	R_ARM_THM_MOVW_BREL	Yes	MOVW	X & 0x0000FFFF
131	R_ARM_THM_GOT_BREL12	Yes	LDR(immediate, Thumb) 12-bit immediate	X & 0x00000FFF

The formation of the initial addend in a REL type relocation for the various instruction classes is described in *Table 4-15 Thumb relocation actions by instruction type*. Insn modification describes how the result X is written back to the instruction; Result_Mask is the value of X after the masking operation described in Table 4-13 or Table 4-14 has been applied.

Table 4-15 Thumb relocation actions by instruction type

Instruction	REL Addend	Insn modification
Thumb-16 instructions		
LDR(1)/LDR(immediate, Thumb), STR(1)/(immediate, Thumb)	insn[10:6] << 2	insn[10:6] = Result_Mask(X) >> 2
LDR(3)/LDR(literal), ADD(5)/ADR	See §4.6.1.1	insn[7:0] = Result_Mask(X) >> 2

Instruction	REL Addend	Insn modification
`CBZ, CBNZ`	See §4.6.1.1	insn [9] = Result_Mask(X) >> 6 insn[7:0] = (Result_Mask(X) >> 1) & 0x1F
`B(2)/B`	sign_extend(insn[10:0] << 1)	insn[10:0] = Result_Mask(X) >> 1
`B(1)/B<cond>`	sign_extend(insn[7:0] << 1)	insn[7:0] = Result_Mask(X) >> 1
`ADD(2)/ADD` (immediate, Thumb, 8-bit immediate), `MOV(1)/MOV` (immediate)	insn[7:0]	insn[7:0] = Result_Mask(X) >> (8**n*) when relocated by `R_ARM_THM_ALU_ABS_G`*n*`[_NC]`
Thumb-32 instructions		
`BL`	See *Thumb call and jump relocations*	See *Thumb call and jump relocations*
`B.W`	See *Thumb call and jump relocations*	See *Thumb call and jump relocations*
`B<cond>.W`	See *Thumb call and jump relocations*	See *Thumb call and jump relocations*
`MOVW`	See §4.6.1.1	insn[19:16] = Result_Mask(X) >> 12 insn[26] = (Result_Mask(X) >> 11) & 0x1 insn[14:12] = (Result_Mask(X) >> 8) & 0x7 insn[7:0] = Result_Mask(X) & 0xFF (encodes the least significant 16 bits)
`MOVT`	See §4.6.1.1. The effect permits executing `MOVW` and later `MOVT` to create a 32-bit link-time constant in a register.	insn[19:16] = Result_Mask(X) >> 28 insn[26] = (Result_Mask(X) >> 27) & 0x1 insn[14:12] = (Result_Mask(X) >> 24) & 0x7 insn[7:0] = (Result_Mask(X) >> 16) & 0xFF (encodes the most significant 16 bits)
`ADR.W`	(insn[26] << 11) | (insn[14:12] << 8) | insn[7:0]	insn[26] = Result_Mask(X) >> 11 insn[14:12] = (Result_Mask(X) >> 8) & 0x7 insn[7:0] = Result_Mask(X) & 0xFF
`LDR<,B,SB,H,SH>(`literal`)`	insn[11:0] $* -1^{(insn[23]==0)}$	insn[23] = (X >= 0) insn[11:0] = Result_Mask(X)
`LDR(`immediate, Thumb`)`12-bit immediate	insn[11:0]	insn[11:0] = Result_Mask(X)

Thumb call and jump relocations

`R_ARM_THM_CALL` is used to relocate Thumb `BL` (and ARMv5 Thumb `BLX`) instructions. It is the Thumb equivalent of `R_ARM_CALL` and the same rules on conversion apply. Bits 0-10 of the first half-word encode the most significant bits of the branch offset, bits 0-10 of the second half-word encode the least significant bits and the offset is in units of half-words. Thus 22 bits encode a branch offset of +/- 2^{22} bytes. When linking ARMv6 (and later, see [ARM ARM]) Thumb code the range of the branch is increased by 2 bits, increasing the offset range to +/- 2^{24} bytes. The same relocation is used for both cases since a linker need only know that the code will run on a Thumb-2 (ARMv6 and later) capable processor to exploit the additional range.

The addend for `B.W` and `B<cond>.W` is the signed immediate quantity encoded in the instruction, extracted in a similar way to `BL`; for details see [ARM ARM].

The conditions under which call and jump relocations are permitted to generate an *ip*-corrupting intra-call veneer, and their behaviour in conjunction with unresolved weak references, are specified in §4.6.1.4 under the heading *Call and Jump relocations*.

4.6.1.7 Static miscellaneous relocations

R_ARM_NONE records that the section containing the place to be relocated depends on the section defining the symbol mentioned in the relocation directive in a way otherwise invisible to the static linker. The effect is to prevent removal of sections that might otherwise appear to be unused.

R_ARM_V4BX records the location of an ARMv4t BX instruction. This enables a static linker to generate ARMv4 compatible images from ARMv4t objects containing only ARM code by converting the instruction to MOV PC, r, where r is the register used in the BX instruction. See [AAPCS] for details. The symbol is unused and may even be the NULL symbol (index 0).

R_ARM_TARGET1 is processed in a platform-specific manner. It may only be used in sections with the types SHT_INIT_ARRAY, SHT_PREINIT_ARRAY, and SHT_FINI_ARRAY. The relocation must be processed either in the same way as R_ARM_REL32 or as R_ARM_ABS32: a virtual platform must specify which method is used. If the relocation is processed as R_ARM_REL32 then the section may be marked read-only and coalesced with other read-only data, otherwise it may only be marked read-only if it does not require dynamic linking.

R_ARM_TARGET2 is processed in a platform-specific manner. It is used to encode a data dependency that will only be dereferenced by code in the run-time support library.

4.6.1.8 Proxy generating relocations

A number of relocations generate proxy locations that are then subject to dynamic relocation. The proxies are normally gathered together in a single table, called the Global Offset Table or GOT. *Table 4-16, Proxy generating relocations* lists the relocations that generate proxy entries.

Table 4-16, Proxy generating relocations

Code	Relocation	Comment
26	R_ARM_GOT_BREL	Offset of the GOT entry relative to the GOT origin
95	R_ARM_GOT_ABS	Absolute address of the GOT entry
96	R_ARM_GOT_PREL	Offset of the GOT entry from the place
97	R_ARM_GOT_BREL12	Offset of the GOT entry from the GOT origin. Stored in the offset field of an ARM LDR instruction
131	R_ARM_THM_GOT_BREL12	Offset of the GOT entry from the GOT origin. Stored in the offset field of a Thumb LDR instruction

All of the GOT entries generated by these relocations are subject to dynamic relocation by R_ARM_GLOB_DAT of the symbol indicated in the generating relocation. There is no provision for generating an addend for the dynamic entry. GOT entries must always be 32-bit aligned words. Multiple GOT-generating relocations referencing the same symbol may share a single entry in the GOT.

R_ARM_GOT_BREL, R_ARM_GOT_BREL12 and R_ARM_THM_GOT_BREL12 generate an offset from the addressing origin of the GOT. To calculate the absolute address of an entry it is necessary to add in the GOT's addressing origin. How the origin is established depends on the execution environment and several relocations are provided in support of it.

- R_ARM_BASE_PREL with the NULL symbol (symbol 0) will give the offset of the GOT origin from the address of the place.
- R_ARM_BASE_ABS with the NULL symbol will give the absolute address of the GOT origin.
- Other execution environments may require that the GOT origin be congruent with some other base. In these environments the appropriate means of establishing that base will apply.

In addition to the data generating relocations listed above the call and branch relocations (R_ARM_CALL, R_ARM_THM_CALL, R_ARM_JUMP24, R_ARM_THM_JUMP24, R_ARM_THM_JUMP19) may also require a proxy to be generated if the symbol will be defined in an external executable or may be pre-empted at execution time. The details of proxy sequences and locations are described in *§3.1.3, PLT Sequences and Usage Models*.

R_ARM_GOTRELAX is reserved to permit future-linker based optimizations of GOT addressing sequences.

4.6.1.9 Relocations for thread-local storage

The static relocations needed to support thread-local storage in a SVr4-type environment are listed in *Table 4-17, Static TLS relocations*.

Table 4-17, Static TLS relocations

Code	Relocation	Place	Comment
104	R_ARM_TLS_GD32	Data	General Dynamic Model
105	R_ARM_TLS_LDM32	Data	Local Dynamic Model
106	R_ARM_TLS_LDO32	Data	Local Dynamic Model
107	R_ARM_TLS_IE32	Data	Initial Exec Model
108	R_ARM_TLS_LE32	Data	Local Exec Model
109	R_ARM_TLS_LDO12	ARM LDR	Local Dynamic Model
110	R_ARM_TLS_LE12	ARM LDR	Local Exec Model
111	R_ARM_TLS_IE12GP	ARM LDR	Initial Exec Model

R_ARM_TLS_GD32 causes two adjacent entries to be added to the dynamically relocated section (the Global Offset Table, or GOT). The first of these is dynamically relocated by R_ARM_TLS_DTPMOD32, the second by R_ARM_TLS_DTPOFF32. The place resolves to the offset of the first of the GOT entries from the place.

R_ARM_TLS_LDM32 is the same as R_ARM_TLS_GD32 except that the second slot in the GOT is initialized to zero and has no dynamic relocation.

R_ARM_TLS_LDO32 resolves to the offset of the referenced data object (which must be local to the module) from the origin of the TLS block for the current module.

R_ARM_TLS_LDO12 is the same as R_ARM_TLS_LDO32 except that the result of the relocation is encoded as the 12-bit offset of an ARM LDR instruction.

R_ARM_TLS_LE32 resolves to the offset of the referenced data object (which must be in the initial data block) from the thread pointer ($tp).

R_ARM_TLS_LE12 is the same as R_ARM_TLS_LE32 except that the result of the relocation is encoded as the 12-bit offset of an ARM LDR instruction.

R_ARM_TLS_IE32 allocates an entry in the GOT that is dynamically relocated by R_ARM_TLS_TPOFF32. The place resolves to the offset of the GOT entry from the place.

R_ARM_TLS_IE12GP allocates an entry in the GOT that is dynamically relocated by R_ARM_TLS_TPOFF32. The place resolved to the offset of the GOT entry from the origin of the GOT and is encoded in the 12-bit offset of an ARM LDR instruction.

New experimental TLS relocations

http://www.lsd.ic.unicamp.br/~oliva/writeups/TLS/RFC-TLSDESC-ARM.txt contains a proposal for enhanced performance of TLS code. At this stage the proposal is still experimental, but the relocations R_ARM_TLS_DESC,

R_ARM_TLS_GOTDESC, R_ARM_TLS_CALL, R_ARM_TLS_DESCSEQ, R_ARM_THM_TLS_CALL, R_ARM_THM_TLS_DESCSEQ16 and R_ARM_THM_TLS_DESCSEQ32 have been reserved to support this.

Note The relocation R_ARM_TLS_DESC re-uses relocation code from the now-obsolete R_ARM_SWI24, but since the former was a static relocation and the new relocation is dynamic there are no practical conflicts in usage.

4.6.1.10 Dynamic relocations

The dynamic relocations for those execution environments that support only a limited number of run-time relocation types are listed in *Table 4-18, Dynamic relocations*.

Table 4-18, Dynamic relocations

Code	Relocation	Comment
17	R_ARM_TLS_DTPMOD32	(S ≠ 0) Resolves to the module number of the module defining the specified TLS symbol, S. (S = 0) Resolves to the module number of the current module (ie. the module containing this relocation).
18	R_ARM_TLS_DTPOFF32	Resolves to the index of the specified TLS symbol within its TLS block
19	R_ARM_TLS_TPOFF32	(S ≠ 0) Resolves to the offset of the specified TLS symbol, S, from the Thread Pointer, TP. (S = 0) Resolves to the offset of the current module's TLS block from the Thread Pointer, TP (the addend contains the offset of the local symbol within the TLS block).
20	R_ARM_COPY	See below
21	R_ARM_GLOB_DAT	Resolves to the address of the specified symbol
22	R_ARM_JUMP_SLOT	Resolves to the address of the specified symbol
23	R_ARM_RELATIVE	(S ≠ 0) B(S) resolves to the difference between the address at which the segment defining the symbol S was loaded and the address at which it was linked. (S = 0) B(S) resolves to the difference between the address at which the segment being relocated was loaded and the address at which it was linked.

With the exception of R_ARM_COPY all dynamic relocations require that the place being relocated is a word-aligned 32-bit object.

R_ARM_JUMP_SLOT is used to mark code targets that will be executed. On platforms that support dynamic binding the relocations may be performed lazily on demand. The unresolved address stored in the place will initially point to the entry sequence stub for the dynamic linker and must be adjusted during initial loading by the offset of the load address of the segment from its link address. Addresses stored in the place of these relocations may not be used for pointer comparison until the relocation has been resolved. In a REL form of this relocation the addend, A, is always 0.

R_ARM_COPY may only appear in executable objects where e_type is set to ET_EXEC. The effect is to cause the dynamic linker to locate the target symbol in a shared library object and then to copy the number of bytes specified by the st_size field to the place. The address of the place is then used to pre-empt all other references to the specified symbol. It is an error if the storage space allocated in the executable is insufficient to hold the full copy of the symbol. If the object being copied contains dynamic relocations then the effect must be as if those relocations were performed before the copy was made.

Note R_ARM_COPY is normally only used in SVr4 type environments where the executable is not position independent and references by the code and read-only data sections cannot be relocated dynamically to refer to an object that is defined in a shared library.

The need for copy relocations can be avoided if a compiler generates all code references to such objects indirectly through a dynamically relocatable location, and if all static data references are placed in relocatable regions of the image. In practice, however, this is difficult to achieve without source-code annotation; a better approach is to avoid defining static global data in shared libraries.

4.6.1.11 Deprecated relocations

Deprecated relocations are in the process of being retired from the specification and may be removed or marked obsolete in future revisions. An object file containing these codes is still conforming, but producers should be changed to use the new alternatives.

The relocations `R_ARM_GNU_VTENTRY` and `R_ARM_GNU_VTINHERIT` have been used by some toolchains to facilitate unused virtual function elimination during linking. This method is not recommended and these relocations may be made obsolete in a future revision of this specification. These relocations may be safely ignored.

Table 4-19, Deprecated relocations

Relocation	Replacement
`R_ARM_PC24`	Use `R_ARM_CALL` or `R_ARM_JUMP24`
`R_ARM_PLT32`	Use `R_ARM_CALL` or `R_ARM_JUMP24`
`R_ARM_LDR_SBREL_11_0_NC`	Use `R_ARM_LDR_SB_G`*`xxx`*
`R_ARM_ALU_SBREL_19_12_NC`	Use `R_ARM_ALU_SB_G`*`xxx`*
`R_ARM_ALU_SBREL_27_20_CK`	Use `R_ARM_ALU_SB_G`*`xxx`*
`R_ARM_SBREL31`	Use new exception table format. Previous drafts of this document sometimes referred to this relocation as `R_ARM_ROSEGREL32`.
`R_ARM_GNU_VTENTRY`	None
`R_ARM_GNU_VTINHERIT`	None

4.6.1.12 Obsolete relocations

Obsolete relocations are no-longer used in this revision of the specification (but had defined meanings in a previous revision). Unlike deprecated relocations, there is no, or little known, use of these relocation codes. Conforming object producers must not generate these relocation codes and conforming linkers are not required to process them. Future revisions of this specification may re-assign these codes for a new relocation type.

4.6.1.13 Private relocations

Relocation types 112-127 are reserved for private experiments. These values will never be allocated by future revisions of this specification. They must not be used in portable object files.

4.6.1.14 Unallocated relocations

All unallocated relocation types are reserved for use by future revisions of this specification.

4.6.2 Idempotency

All `RELA` type relocations are idempotent. They may be reapplied to the place and the result will be the same. This allows a static linker to preserve full relocation information for an image by converting all `REL` type relocations into `RELA` type relocations.

Note A REL type relocation can never be idempotent because the act of applying the relocation destroys the original addend.

5 PROGRAM LOADING AND DYNAMIC LINKING

5.1 Introduction

This section provides details of ARM-specific definitions and changes relating to executable images.

5.2 Program Header

The Program Header provides a number of fields that assist in interpretation of the file. Most of these are specified in the base standard. The following fields have ARM-specific meanings.

p_type

Table 5-1, Processor-specific segment types lists the processor-specific segment types.

Table 5-1, Processor-specific segment types

Name	p_type	Meaning
`PT_ARM_ARCHEXT`	`0x70000000`	Platform architecture compatibility information
`PT_ARM_EXIDX` `PT_ARM_UNWIND`	`0x70000001`	Exception unwind tables

A segment of type `PT_ARM_ARCHEXT` contains information describing the platform capabilities required by the executable file. The segment is optional, but if present it must appear before segment of type `PT_LOAD`. The platform independent parts of this segment are described in *§5.2.1, Platform architecture compatibility* .

`PT_ARM_EXIDX` (alias `PT_ARM_UNWIND`) describes the location of a program's unwind tables.

p_flags

There are no processor-specific flags. All bits in the PT_MASKPROC part of this field are reserved to future revisions of this specification.

5.2.1 Platform architecture compatibility data

This data describes the platform capabilities required by an executable file. It can be constructed by a linker using the attributes [ABI-addenda, §2] found in its input relocatable files, or otherwise.

If this segment is present it shall contain at least one 32-bit word with meaning defined by Table 5-2, Table 5-3, Table 5-4, and Table 5-5.

Table 5-2, Common architecture compatibility data masks

Name	Value	Meaning
`PT_ARM_ARCHEXT_FMTMSK`	`0xff000000`	Masks bits describing the format of data in subsequent words. The masked value is described in Table 5-3, below.
`PT_ARM_ARCHEXT_PROFMSK`	`0x00ff0000`	Masks bits describing the architecture profile required by the executable. The masked value is described in Table 5-4, below.

Name	Value	Meaning
`PT_ARM_ARCHEXT_ARCHMSK`	`0x000000ff`	Masks bits describing the *base architecture* required by the executable. The masked value is described in Table 5-5, below.

Table 5-3, Architecture compatibility data formats lists the architecture compatibility data formats defined by this ABI. All other format identifiers are reserved to future revisions of this specification.

Table 5-3, Architecture compatibility data formats

Name	Value	Meaning
`PT_ARM_ARCHEXT_FMT_OS`	`0x00000000`	There are no additional words of data. However, if EF_OSABI is non-zero, the relevant platform ABI may define additional data that follows the initial word.
`PT_ARM_ARCHEXT_FMT_ABI`	`0x01000000`	§5.2.1.1, below describes the format of the following data words.

Table 5-4, Architecture profile compatibility data, lists the values specifying the architectural profile needed by an executable file.

Table 5-4, Architecture profile compatibility data

Name	Value	Meaning
`PT_ARM_ARCHEXT_PROF_NONE`	`0x00000000`	The architecture has no profile variants, or the image has no profile-specific constraints
`PT_ARM_ARCHEXT_PROF_ARM`	`0x00410000` (‘A’<<16)	The executable file requires the Application profile
`PT_ARM_ARCHEXT_PROF_RT`	`0x00520000` (‘R’<<16)	The executable file requires the Real-Time profile
`PT_ARM_ARCHEXT_PROF_MC`	`0x004D0000` (‘M’<<16)	The executable file requires the Microcontroller profile
`PT_ARM_ARCHEXT_PROF_CLASSIC`	`0x00530000` (‘S’<<16)	The executable file requires the ‘classic’ (‘A’ or ‘R’ profile) exception model.

Table 5-5, Architecture version compatibility data defines the values that specify the minimum architecture version needed by this executable file. These values are identical to those of the `Tag_CPU_arch` attribute used in the attributes section [ABI-addenda, §2] of a relocatable file.

Table 5-5, Architecture version compatibility data

Name	Value	Meaning the executable file needs (at least) …
`PT_ARM_ARCHEXT_ARCH_UNKN`	`0x00`	The needed architecture is unknown or specified in some other way
`PT_ARM_ARCHEXT_ARCHv4`	`0x01`	Architecture v4
`PT_ARM_ARCHEXT_ARCHv4T`	`0x02`	Architecture v4T
`PT_ARM_ARCHEXT_ARCHv5T`	`0x03`	Architecture v5T

Name	Value	Meaning the executable file needs (at least) …
PT_ARM_ARCHEXT_ARCHv5TE	0x04	Architecture v5TE
PT_ARM_ARCHEXT_ARCHv5TEJ	0x05	Architecture v5TEJ
PT_ARM_ARCHEXT_ARCHv6	0x06	Architecture v6
PT_ARM_ARCHEXT_ARCHv6KZ	0x07	Architecture v6KZ
PT_ARM_ARCHEXT_ARCHv6T2	0x08	Architecture v6T2
PT_ARM_ARCHEXT_ARCHv6K	0x09	Architecture v6K
PT_ARM_ARCHEXT_ARCHv7	0x0A	Architecture v7 (in this case the architecture profile may also be required to fully specify the needed execution environment)
PT_ARM_ARCHEXT_ARCHv6M	0x0B	Architecture v6M (e.g. Cortex M0)
PT_ARM_ARCHEXT_ARCHv6SM	0x0C	Architecture v6S-M (e.g. Cortex M0)
PT_ARM_ARCHEXT_ARCHv7EM	0x0D	Architecture v7E-M

5.2.1.1 Platform architecture compatibility data (ABI format)

The status of this section is *informative*. It records a proposal that *might* be adopted.

The data following the word defined by §5.2.1 consists of an array of 2-byte signed integers (starting at offset 4 in the architecture compatibility data segment) followed by a number of null-terminated byte strings (NTBS). The p_filesz field of the segment header gives the total size in bytes of the architecture compatibility data.

The integer *array* maps the ABI public attribute tags [ABI-addenda, §2] as follows.

- *Array*[0] contains the number of elements in *array*.
- If *tag* ≥ *array*[0], the value of *tag* for the executable file is 0. Only tags with non-0 values need to be mapped.
- If 4 ≤ *tag* < *array*[0], the value of *tag* for the executable file is *array*[*tag*]. A negative value *v* denotes that *tag* has the NTBS value found at offset –*v* from the start of the segment.
- *Array*[1] contains the major version number and *array*[2] the minor version number of the ABI release to which the data conforms (at least 2, 8). *Array*[3] is reserved and should be 0.

5.3 Program Loading

There are no processor-specific definitions relating to program loading.

5.4 Dynamic Linking

5.4.1 Dynamic Section

Table 5-6, ARM-specific dynamic array tags lists the processor-specific dynamic array tags.

Table 5-6, ARM-specific dynamic array tags

Name	Value	d_un	Executable	Shared Object
DT_ARM_RESERVED1	0x70000000			

Name	Value	d_un	Executable	Shared Object
DT_ARM_SYMTABSZ	0x70000001	d_val	Platform specific	Platform specific
DT_ARM_PREEMPTMAP	0x70000002	d_ptr	Platform specific	Platform specific
DT_ARM_RESERVED2	0x70000003			

DT_ARM_SYMTABSZ gives the number of entries in the dynamic symbol table, including the initial dummy symbol.

DT_ARM_PREEMPTMAP holds the address of the pre-emption map for platforms that use the DLL static binding model. See *§3.1.2 Symbol Pre-emption in DLLs* for details. On platforms that permit use of a pre-emption map, the DT_SONAME tag must be present in all shared objects.

Note Some executable images may exist that use DT_ARM_RESERVED1 and DT_ARM_RESERVED2 instead of DT_ARM_SYMTABSZ and DT_ARM_PREEMPTMAP respectively. These tags use the d_un field in a manner incompatible with the Generic ELF requirements.

5.5 Post-Link Processing

For some execution environments a further processing step may be needed after linking before an executable can be run on the target environment. The precise processing may depend on both the target platform. Depending on the nature of the post-processing it may be done in any of following places

- As a final step during linking
- As a preliminary step during execution of the image
- As a separate post-linking step

In some cases the result may still be an ELF executable image, in others it may produce an image that is in some other format more appropriate to the operating system.

5.5.1 Production of BE-8 images

Images that are expected to execute in big-endian mode on processors that implement Architecture version 6 or higher will normally need to be post-processed to convert the instructions that are in big-endian byte order to little-endian as expected by the processor. The mapping symbol information can be used to do this transformation accurately. In all segments that contain executable code:

- For areas mapped as data ($d or $d.<*any…*>) no changes are made
- For areas mapped as Thumb ($t or $t.<*any…*>) each half-word aligned pair of bytes are swapped
- For areas mapped as ARM ($a or $a.<*any…*>) each word-aligned object is swapped so that the first and fourth bytes are exchanged and the second and third exchanged.

An ELF image that has been transformed in this manner is marked by setting EF_ARM_BE8 in the e_flags field.

Note If BE-8 images are subject to further relocation of instructions (either by a dynamic linker or by further post-linking operations) account must be taken of the fact that the instructions are now in little-endian format.

APPENDIX A SPECIMEN CODE FOR PLT SEQUENCES

The status of this appendix is informative.

A.1 DLL-like, single address space, PLT linkage

The simplest code sequence for the PLT entry corresponding to imported symbol X is:

```
LDR   ip, [pc, #0]         ; Load the 32-bit offset of my PLTGOT entry from SB
LDR   pc, [ip, sb]!        ; Branch indirect through the PLTGOT entry leaving
                           ; ip addressing the PLTGOT slot
DCD   R_ARM_GOT_BREL(X)    ; GOT_BASE = SB
```

The final DCD is subject to relocation by a PLTGOT-generating relocation directive. This directive may be processed by a target-specific linker or by a target-specific post-linker. After processing:

- The place contains the 32-bit offset from the static base (sb) of the PLTGOT entry for X.
- The PLTGOT entry for X is subject to an R_ARM_JUMP_SLOT(X) dynamic relocation.

A more complicated sequence that avoids one of the memory accesses is:

```
ADD   ip, sb, #:SB_OFFSET_27_20:__PLTGOT(X)     ; R_ARM_ALU_SB_G0_NC(__PLTGOT(X))
ADD   ip, ip, #:SB_OFFSET_19_12:__PLTGOT(X)     ; R_ARM_ALU_SB_G1_NC(__PLTGOT(X))
LDR   pc, [ip, #:SB_OFFSET_11_0:__PLTGOT(X)]!   ; R_ARM_LDR_SB_G2(__PLTGOT(X))
```

If the linker can place all PLTGOT entries within 1MB of SB, the sequence becomes:

```
ADD   ip, sb, #:SB_OFFSET_19_12:__PLTGOT(X)     ; R_ARM_ALU_SB_G0_NC(__PLTGOT(X))
LDR   pc, [ip, #:SB_OFFSET_11_0:__PLTGOT(X)]!   ; R_ARM_LDR_SB_G1(__PLTGOT(X))
```

The write-back on the final LDR ensures that ip contains the address of the PLTGOT entry. This is critical to incremental dynamic linking.

A.2 DLL-like, multiple virtual address space, PLT linkage

The code sequence for the PLT entry corresponding to imported symbol X is:

```
LDR   ip, [pc, #0]         ; Load the 32-bit address of my PLTGOT entry
LDR   pc, [ip]             ; Branch indirect through the PLTGOT entry
DCD   R_ARM_GOT_ABS(X)     ; GOT_BASE = 0
```

Note that ip addresses the PLTGOT entry, which is critical to incremental dynamic linking.

The final DCD is subject to relocation by a PLTGOT-generating relocation directive. This directive may be processed by a target-specific linker or by a target-specific post-linker. After processing:

- The place contains the 32-bit address of the PLTGOT entry for X.
- The PLTGOT entry for X is subject to an R_ARM_JUMP_SLOT(X) dynamic relocation.

Because a DLL has two segments that can be loaded independently, there is no more efficient address generating sequence – analogous to the SB-relative sequence shown in above – that does not require complex instruction field-relocating directives to be processed at dynamic link time.

This ABI requires dynamic relocations to relocate 32-bit fields, so there is no sequence analogous to that of the preceding subsection.

A.3 SVr4 DSO-like PLT linkage

The simplest code sequence for the PLT entry corresponding to imported symbol X is:

```
     LDR   ip, L2          ; Load the 32-bit pc-relative offset of my PLTGOT entry
L1:  ADD   ip, ip, pc      ; formulate its address...
     LDR   pc, [ip]        ; Branch through the PLTGOT entry addressed by ip
L2:  DCD   R_ARM_GOT_PREL(X) + (L2 – L1 – 8)
```

The dynamic linker relies on ip addressing the PLTGOT entry for X.

The final DCD is subject to static relocation by a PLTGOT-generating relocation directive. This directive may be processed by a target-specific linker or by a target-specific post-linker. After processing:

- The place contains the 32-bit offset from L1+8 to the PLTGOT entry for X.
- The PLTGOT entry for X is subject to an R_ARM_JUMP_SLOT(X) dynamic relocation.

A more complicated, pc-relative, sequence that avoids one of the memory accesses is shown below. Because an SVr4 executable file is compact (usually < 2^{28} bytes) and rigid (it has only one base address, whereas a DLL has two), all the relocations can be fully resolved during static linking.

```
ADD   ip, pc, #-8:PC_OFFSET_27_20: __PLTGOT(X)      ; R_ARM_ALU_PC_G0_NC(__PLTGOT(X))
ADD   ip, ip, #-4:PC_OFFSET_19_12: __PLTGOT(X)      ; R_ARM_ALU_PC_G1_NC(__PLTGOT(X))
LDR   pc, [ip, #0:PC_OFFSET_11_0: __PLTGOT(X)]!     ; R_ARM_LDR_PC_G2(__PLTGOT(X))
```

The write-back on the final LDR ensures that ip contains the address of the PLTGOT entry. This is critical to incremental dynamic linking.

In effect, the sequence constructs a 28-bit offset for the LDR. The first relocation does the right thing because pc addresses the LDR, so, in general, it picks out bits [27-20] of that offset. The third relocation picks out bits [11-0] of the same offset. The second relocation needs to construct bits [19-12] of the offset from dot+4 to X., that is, from dot to X-4. Ignoring the -4 sometimes produces the wrong answer!

Encoding such a small addend requires that the initial value not be shifted by the shift applied to the result value. This is expected for a RELA-type relocation that can encode -4 directly. However, a REL-type must encode the initial value of the addend using SUB ip, ip, #4.

In small enough DSOs (< 2^{20} bytes from the PLT to the PLTGOT) the first instruction can be omitted, and the sequence collapses to the following.

```
SUB   ip, pc, #4:PC_OFFSET_19_12: __PLTGOT(X)       ; R_ARM_ALU_PC_G0_NC(__PLTGOT(X))
LDR   pc, [ip, #0:PC_OFFSET_11_0: __PLTGOT(X)]!     ; R_ARM_LDR_PC_G2(__PLTGOT(X))
```

A.4 SVr4 executable-like PLT linkage

An SVr4 executable does not need be position independent, its writable segment can be relocated dynamically, and it is compact and rigid. Therefore, its PLT entries can use the simple, absolute code sketched in §A.2 or the more complex, pc-relative, versions sketched in §A.3, as the tool chain chooses.

In both cases, ip must address the corresponding PLTGOT slot at the point where the PLT calls through it.

APPENDIX B CONVENTIONS FOR SYMBOLS CONTAINING $

The status of this appendix is informative.

A toolchain is not required to support any of the conventions described in this appendix; however, it is recommended that if symbols matching the patterns described are used, then the following conventions are adhered to.

B.1 Base, Length and Limit symbols

A number of symbols may be used to delimit the addresses and sizes of aspects of a linked image. These symbols are of the following general forms:

```
Load$$region_name$$Base
Image$$region_name$${Base|Length|Limit}
Image$$region_name$$ZI$${Base|Length|Limit}
Image$${RO|RW|ZI}$${Base|Limit}
SectionName$${Base|Limit}
```

A toolchain may define these symbols unconditionally, or only if they are referred to by the application: so a post-linker must not depend on the existence of any of these symbols.

B.2 Sub-class and Super-class Symbols

A symbol `$Sub$$`*`name`* is the sub-class version of *`name`*. A symbol `$Super$$`*`name`* is the super-class version of *`name`*. In the presence of a definition of both *`name`* and `$Sub$$`*`name`*:

- A reference to *`name`* resolves to the definition of `$Sub$$`*`name`*.
- A reference to `$Super$$`*`name`* resolves to the definition of *`name`*.

It is an error to refer to `$Sub$$`*`name`*, or to define `$Super$$`*`name`*, or to use `$Sub$$`… or `$Super$$`… recursively.

B.3 Symbols for Veneering and Interworking Stubs

A veneer symbol has the same binding as the symbol it veneers. They are used to label sequences of instructions that are automatically generated during linking. The general format of the symbols is:

`${Ven|`*`other`*`}${AA|AT|TA|TT}${I|L|S}[$PI]$$`*`symbol_name`*

where AA, AT, TA, or TT denotes the type of the veneer — ARM to ARM, ARM to Thumb, etc; `I`, `L`, or `S` denotes inline (the target follows immediately), long reach (32-bit), or short reach (typically 26-bit); and `$PI` denotes that the veneer is position independent.