ELF for the ARM® Architecture

Development systems Division

Compiler Tools Group

Document number:	GENC-003538
Date of Issue:	1st December 2003
Author:	Richard Earnshaw
Authorized by:	

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

Licence

Proprietary notice

Contents

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.

This DRAFT specification can be changed or updated by ARM without notice. Issue and version number will change on republication. The material contained herein is believed to be accurate, but is known to be incomplete. Anticipated changes include:

- Typographical corrections.
- Clarifications.
- Outstanding defect reports.
- Addition of detail and further relocation types to §4.6, *Relocation*.
- Completion and correction of sections flagged by yellow highlight.
- Completion of skeleton §5, *Program Loading and Dynamic Linking*.

1.1.2 Change history

Issue	Date	By	Change
0.2	31st October 2003	Lee Smith	First public release.
0.3	1st December 2003	Richard Earnshaw	Second public release.

1.2 References

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

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
SCO-ELF	http://www.sco.com/developers/gabi/2001-04-24/contents.html	System V Application Binary Interface - DRAFT - 24 April 2001

1.3 Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
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	EABI (see below) for the ARM Architecture, *this* [E]ABI.
ARM-based	… based on the ARM architecture …
EABI	An ABI suited to the needs of embedded, and deeply embedded (sometimes called *free standing*), applications.
ELF	Executable and Linking Format
OS	Operating System

1.4 About the licence to use this specification

Use of these *ABI for the ARM Architecture* specifications published by ARM is governed by the simple licence agreement shown on the cover page of this document, and on the cover page of each major component document. Without formalities or payment, you are licensed to use any IP rights ARM might hold in these ABI specifications for the purpose of producing products that comply with these ABI specifications.

Because these specifications may be updated by ARM without notice, we prefer that these specifications should not be copied, but that third parties should refer directly to them, in the same way that we refer directly to the specifications underpinning this ABI, such as the specifications of ELF, DWARF, and the generic C++ ABI.

1.5 Acknowledgements

This specification could not have been developed without contributions from, and the active support of, the following organizations. In alphabetical order: ARM, Intel, Metrowerks, Montavista, Nexus Electronics, PalmSource, Symbian, 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.

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 April 2001 draft, which was the most recent stable draft at the time this specification was developed.

3 INTRODUCTION

This section is a place holder for additional material…

3.1 Platform Standards

4 OBJECT FILES

4.1 Introduction

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 ARM.

e_machine

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

e_entry

The base ELF specification requires this field to be non-zero if an application has an entry point. Some applications may require an entry point of zero (for example, via the reset vector); a platform standard may specify that an executable image always has an entry point, in which case e_entry always specifies the entry point, even if zero.

e_flags

The processor-specific flags are shown in *Table 4-1, ARM-specific e_flags*. Unallocated bits, and bits allocated in previous versions of this specification, are reserved to ARM.

Table 4-1, ARM-specific e_flags

Value	Meaning
`EF_ARM_EABIMASK` (0xFF000000) (current version is 0x04000000)	This masks an 8-bit version number, the version of the ABI to which this ELF file conforms. This ABI is version 4. 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 will normally only be set on an Executable file.
`EF_ARM_LE8` (0x00400000)	The ELF file contains LE-8 code, suitable for execution on an ARM Architecture v6 processor. This flag will normally only be set on an Executable file, and only when the ELF file is itself in big-endian format (`e_ident[EI_DATA]=ELFDATA2MSB`).

XXX More information from V6BE.txt

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 ARM Architecture v6 it is possible to execute programs that are in the “opposite endianness”; objects with this requirement will be marked with either `EF_ARM_BE8` or `EF_ARM_LE8` 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 are currently no processor-specific values for this field and all such values are reserved to ARM.

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 ARM.

4.3.2 Section Types

The defined processor-specific section types are listed in *Table 4-2, Processor specific section types*. All other processor-specific values are reserved to ARM.

Table 4-2, Processor specific section types

Name	Value	Comment
`SHT_ARM_EXIDX`	`0x70000001`	Exception Index table

Pointers in sections of types `SHT_INIT_ARRAY`, `SHT_PREINIT_ARRAY` and `SHT_FINI_ARRAY` shall be expressed relative to the address of the pointer.

SHT_ARM_EXIDX marks a section that contains index information for exception unwinding. See *EHABI* for details.

4.3.3 Section Attribute Flags

There are no processor-specific section attribute flags defined. All processor-specific values are reserved to ARM.

4.3.4 Special Sections

Table 4-3, ARM special sections lists the special sections that are defined.

Table 4-3, ARM special sections

Name	Type	Attributes
`.ARM.exidx`	`SHT_ARM_EXIDX`	`SHF_ALLOC + SHF_LINK_ORDER`
`.ARM.extab`	`SHT_PROGBITS`	`SHF_ALLOC`

`.ARM.exidx` names a section that contains index entries for section unwinding. See *EHABI* for details.

`.ARM.extab` names a section that contains exception unwinding information. See *EHABI* for details.

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 impose a limit on the alignment that they can guarantee to provide (normally the page size).

4.4 String Table

There a 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 ARM.

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 address of 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 defintion 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`).

[aside — 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`).

All symbol names containing a dollar character (‘`$`’) are reserved to ARM.

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

Multiple conventions exist for the names of compiler temporary symbols (for example, ARMCC uses `L`*`xxx.yyy`*, while GNU uses `.L`*`xxx`*). More generally, any symbol with binding `STB_LOCAL` and type `STT_NOTYPE` may be removed from an object and replaced with an offset from another symbol in the same section under the following conditions:

- The replacement symbol is not of type `STT_FUNC`.
- 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.

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.

4.5.5 Sub-class and super-class symbols [optional]

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 defintion of both *`name`* and `$Sub$$`*`name`*:

- A reference to *`name`* resovles 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.

A platform standard may mandate support of sub- and super-class symbols.

There are outstanding defects for sub- and super-class symbols DE-316140.

4.5.6 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 mapping symbols are defined in *Table 4-4, 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 4-4, 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

4.5.6.1 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.

4.5.6.2 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.

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 Mandatory relocation types

Table 4-5, Mandatory relocation types lists the relocation types that must be supported by all linkers. The table shows:

- The *type* which is stored in the ELF32_R_TYPE component of the r_info field.
- The name of the relocation type.
- The type of *place* that can be relocated by this relocation. For instructions this is sub-divided into ARM and Thumb instructions and then the type of underlying instruction is further described. From this information it is possible to determine:
  - The initial addend, for a REL type relocation
  - The appropriate limits for overflow checking
  - Any further modifications that must be necessary when writing out the relocated value.
- The *size* and *alignment* of the place being relocated (in bytes) and the type of overflow checking that must be performed: **S**igned, **U**nsigned or **N**one.
- The computation that must be performed in order to determine the relocation result. The following nomenclature is used
  - S denotes the value of symbol referenced in ELF32_R_SYM component of the r_info field.
  - A denotes the initial addend. For a RELA type relocation the value is used unmodified. For a REL type relocation the value must be extracted from the place in a manner that is determined by the type of the place.
  - P denotes the address of the place being relocated. It is the sum of the r_offset field and the base address of the section being relocated (note that all relocations involving P are of the form S – P, where the symbol referenced is in the same consolidated output section as P, so it is not necessary to know the absolute address of the section being relocated).

- B is the *nominal base address* used for accessing objects in the read-write data areas.
- E is the *nominal base address* used for accessing objects in the executable and read-only areas.

The precise definition of a *nominal base address* is platform defined, but it must be possible for the application to retrieve the value at run time by one of the following methods:

- A pre-determined value
- A value in a known register
- A suitable symbol
- A library call

The platform documentation must describe the appropriate model for each of B and E (they need not be the same).

Table 4-5, Mandatory relocation types

Type	Name	Place	Size Alignment Overflow	Computation
0	`R_ARM_NONE`	None	0/1/n	No relocation. Encodes dependencies between section
1	`R_ARM_PC24`	ARM B/BL/BLX	4/4/s	`S – P + A`
2	`R_ARM_ABS32`	Data	4/1/n	`S + A`
3	`R_ARM_REL32`	Data	4/1/n	`S – P + A`
4	`R_ARM_PC13`	ARM LDR r, [pc,…]	4/4/s	`S – P + A`
5	`R_ARM_ABS16`	Data	2/1/u	`S + A`
6	`R_ARM_ABS12`	ARM LDR/STR	4/4/s	`S + A`
7	`R_ARM_THM_ABS5`	Thumb LDR/STR	2/2/u	`S + A`
8	`R_ARM_ABS8`	Data	1/1/u	`S + A`
9	`R_ARM_SBREL32`	Data	4/1/n	`S – B + A`
10	`R_ARM_THM_PC22`	Thumb BL/BLX pair	4/2/s	`S – P + A`
11	`R_ARM_THM_PC8`	Thumb LDR r, [pc,…]	2/2/u	`S – P + A`
12				Reserved
13	`R_ARM_SWI24`	ARM SWI	4/4/u	`S + A`
14	`R_ARM_THM_SWI8`	Thumb SWI	2/2/u	`S + A`
15	*`R_ARM_XPC25`*			Obsolete. Use `R_ARM_PC24`
16	*`R_ARM_THM_XPC22`*			Obsolete. Use `R_ARM_THM_PC22`

Type	Name	Place	Size Alignment Overflow	Computation
32	R_ARM_ALU_PCREL_7_0	ARM ADD/SUB	4/4/n	(S – P + A) & 0x000000FF
33	R_ARM_ALU_PCREL_15_8	ARM ADD/SUB	4/4/n	(S – P + A) & 0x0000FF00
34	R_ARM_ALU_PCREL_23_15	ARM ADD/SUB	4/4/n	(S – P + A) & 0x00FF0000
35	R_ARM_LDR_SBREL_11_0	ARM LDR/STR	4/4/n	(S – B + A) & 0x00000FFF
36	R_ARM_ALU_SBREL_19_12	ARM ADD/SUB	4/4/n	(S – B + A) & 0x000FF000
37	R_ARM_ALU_SBREL_27_20	ARM ADD/SUB	4/4/n	(S – B + A) & 0x0FF00000
38	R_ARM_RELABS32	Data	4/1/n	S + A *or* S – P + A
39	R_ARM_ROSEGREL32	Data	4/1/n	S – E + A
40	R_ARM_V4BX	ARM BX r	4/4/n	None. Used to mark BX instructions in ARMv4T code.
41	R_ARM_STKCHK	ARM ??	4/4/s	Reserved for stack-limit checking
42	R_ARM_THM_STKCHK	Thumb ??	4/2/s	Reserved for stack-limit checking
43-52				Reserved for Thumb-2

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_PC24 is used to relocate an ARM B or BL instruction (and on ARMv5 an ARM BLX instruction). Bits 0-23 encode a signed offset, in units of 4-byte instructions (thus 24 bits encode a branch offset of +/- 2^{25} bytes). For a BLX instruction bit 24 additionally encodes the appropriate half-word address of the destination and there is an implicit transition to Thumb state. A static linker may convert a BL to a BLX instruction (or vice-versa) if generating an image for ARMv5 or later. If it is unable to do this (as is the case for B, or BL<cond> or on ARMv4T) then it must generate a suitable sequence of instructions that will perform the transition to the target. The instruction sequence may make use of the intra-procedure scratch register (IP) and does not need to preserve its value. The relocation must then be recalculated using the address of the sequence instead of S. Compensation for the PC bias (8 bytes) must be factored into the relocation expression by the object producer.

R_ARM_PC13 is used to relocate an ARM LDR instruction where the base register for the address is PC. Bits 0-11 encode an unsigned offset in bytes and bit 23 encodes an inverted sign bit from a 13-bit sign-magnitude representation. Compensation for the PC bias (8 bytes) must be factored into the relocation expression by the object producer.

R_ARM_THM_PC22 is used to relocate Thumb BL (and on ARMv5 Thumb BLX) instructions. It is thumb equivalent of R_ARM_PC24 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. Compensation for the PC bias (4 bytes) must be factored into the relocation expression by the object producer.

`R_ARM_V4BX` records the location of an ARMv4t `BX` instruction. This enables a static linker to generate ARMv4 compatible images from ARMv4t objects that contain 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 unnamed.

4.6.1.2 Platform specific relocation types

Add these (particularly SVr4 types).

4.6.1.3 Private relocation types

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.

4.6.1.4 Unallocated relocation types

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

This section will be added in a future draft.

5.1 Introduction

5.2 Program Header

5.3 Program Loading

5.4 Dynamic Linking