ELF for the ARM® 64-bit Architecture (AArch64)

Document number: ARM IHI 0056A, current through AArch64 ABI release 00bet3
Date of Issue: 20th December 2011

Abstract

This document describes the use of the ELF binary file format in the Application Binary Interface (ABI) for the ARM 64-bit architecture.

Keywords

ELF, AArch64 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 3 months old (navigate to the *Software Development Tools* section, *Application Binary Interface for the ARM Architecture* subsection).

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

Licence

Proprietary notice

Contents

1 ABOUT THIS DOCUMENT

1.1 Change control

1.1.1 Current status and anticipated changes

This is a beta release of the specification. Clarifications, extensions and minor changes should be expected.

Text highlighted in yellow denotes recent changes..

1.1.2 Change history

Issue	Date	By	Change
A 0.01	24th June 2010	LS	First release
A 0.02	25th June 2010	LS	Corrected section 5; better trademark wording.
A 0.03	2nd July 2010	LS	Subsumed ARM-internal reaction to A 02.
A 0.04	14th July 2011	NS	Add R_AARCH64_LDST128_ABS_LO12_NC relocation type.
A0.05	20th December 2011	LS	Removed references to BPABI64/EHABI64; these will be replaced not ported from AArch32. Fixed relocations 275, 311 (^ADRH^ADRP^), 311 (missing `Page( )`) and 1024 (^ARM^AARCH64^). Fixed broken definition of `GOT` and added `R_AARCH64_LD64_GOTPAGE_LO15`. Fixed broken wording in tables 4-11 and 4-12. Improved wording and formatting in §4.6.

1.2 References

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

Ref	External reference or URL	Title
AAELF64	*This document*	ELF for the ARM 64-bit Architecture (AArch64).
AAPCS64	IHI 0055A	Procedure Call Standard for the ARM 64-bit Architecture
Addenda32	IHI 0045C	Addenda to, and Errata in, the ABI for the ARM Architecture
LSB	http://www.linuxbase.org/	Linux Standards Base
SCO-ELF	http://www.sco.com/developers/gabi/	System V Application Binary Interface – DRAFT
SYM-VER	http://people.redhat.com/drepper/symbol-versioning	GNU Symbol Versioning

1.3 Terms and abbreviations

The *ABI for the ARM 64-bit Architecture* uses the following terms and abbreviations.

Term	Meaning
A32	The instruction set named *ARM* in the ARMv7 architecture; A32 uses 32-bit fixed-length instructions.
A64	The instruction set available when in AArch64 state.
AAPCS64	Procedure Call Standard for the ARM 64-bit Architecture (AArch64)
AArch32	The 32-bit general-purpose register width state of the ARMv8 architecture, broadly compatible with the ARMv7-A architecture.
AArch64	The 64-bit general-purpose register width state of the ARMv8 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*, *ELF for the ARM Architecture*, …
ARM-based	… based on the ARM architecture …
Floating point	Depending on context *floating point* means or qualifies: (a) floating-point arithmetic conforming to IEEE 754 2008; (b) the ARMv8 floating point instruction set; (c) the register set shared by (b) and the ARMv8 SIMD instruction set.
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.
SIMD	Single Instruction Multiple Data – A term denoting or qualifying: (a) processing several data items in parallel under the control of one instruction; (b) the ARM v8 SIMD instruction set: (c) the register set shared by (b) and the ARMv8 floating point instruction set.
SIMD and floating point	The ARM architecture's SIMD and Floating Point architecture comprising the floating point instruction set, the SIMD instruction set and the register set shared by them.
T32	The instruction set named *Thumb* in the ARMv7 architecture; T32 uses 16-bit and 32-bit instructions.

1.4 Your licence to 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

2 ABOUT THIS SPECIFICATION

This specification provides the processor-specific definitions required by ELF [SCO-ELF] for AArch64-based systems.

The ELF specification is part of the larger System V ABI specification where it forms chapters 4 and 5. However, the ELF 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.

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.

Text highlighted in yellow denotes unresolved issues likely to be revised, perhaps substantially.

3 PLATFORM STANDARDS (EXAMPLE ONLY)

We expect that each operating system that adopts components of this ABI specification will specify additional requirements and constraints that must be met by application code in binary form and the code-generation tools that generate such code.

As an example of the kind of issue that must be addressed §3.1, below, lists some of the issues addressed by the *Linux Standard Base* [LSB] specifications.

3.1 Linux Platform ABI (example only)

3.1.1 Symbol Versioning

The Linux ABI 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 Standard Base specifications* [LSB], and in Drepper's paper [SYM-VER].

A binary file intended to be specific to Linux shall set the `EI_OSABI` field to the value required by Linux [LSB].

3.1.2 Program Linkage Table (PLT) Sequences and Usage Models

3.1.2.1 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. Such a location is called an *imported* location or *imported* symbol.

SVr4-based *Dynamic Shared Objects* (DSOs) (e.g. for Linux) 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 relocation directive that relocates an AArch64 B/BL-class instruction (§4.6.7). For a Linux/SVr4 DSO, each `STB_GLOBAL` symbol with `STV_DEFAULT` visibility is a candidate.

3.1.2.2 Overview of PLT entry code generation

A PLT entry must be able to branch any distance. This is typically achieved by loading the destination address from the corresponding *Global Object Table* (GOT) entry.

On-demand dynamic linking constrains the code sequences that can be generated for a PLT entry. Specifically, there is a requirement from the dynamic linker for certain registers to contain certain values. Typically these are:

- The address or index of the of not-yet-linked PLT entry.
- The return address of the call to the PLT entry.

The register interface to the dynamic linker is specified by the host operating system.

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
aeabi	Reserved to the ABI for the ARM Architecture (EABI pseudo-vendor)
AnonXyz anonXyz	Reserved to private experiments by the *Xyz* vendor. Guaranteed not to clash with any registered vendor name.
ARM	ARM Ltd (Note: the company, not the processor).
cxa	C++ ABI pseudo-vendor
FSL	Freescale Semiconductor Inc.
GHS	Green Hills Systems
gnu	GNU compilers and tools (Free Software Foundation)
iar	IAR Systems
intel	Intel Corporation
ixs	Intel Xscale
PSI	PalmSource Inc.
RAL	Rowley Associates Ltd
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_machine

An object file conforming to this specification must have the value EM_AARCH64 (183, 0xB7).

e_entry

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 a 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

There are no processor-specific flags so this field should be zero.

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 AArch64 ELF file shall contain `ELFCLASS64` 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.

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`).

4.3 Sections

4.3.1 Special Section Indexes

No processor-specific special section indexes are 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 4-2, Processor specific section types*. All other processor-specific values are reserved to future revisions of this specification.

Table 4-2, Processor specific section types

Name	Value	Comment
`SHT_ARM_EXIDX`	`0x70000001`	Reserved for Exception Index table
`SHT_ARM_ATTRIBUTES`	`0x70000003`	Reserved for Object file compatibility attributes

4.3.3 Section Attribute Flags

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

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 4-3, AArch64 special sections lists the special sections defined by this ABI.

Table 4-3, AArch64 special sections

Name	Type	Attributes
.ARM.exidx*	SHT_ARM_EXIDX	SHF_ALLOC + SHF_LINK_ORDER
.ARM.extab*	SHT_PROGBITS	SHF_ALLOC
.ARM.attributes	SHT_ARM_ATTRIBUTES	none

Names beginning .ARM.exidx name sections containing index entries for section unwinding. Names beginning .ARM.extab name sections containing exception unwinding information.

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

Additional special sections may be required by some platforms standards.

4.3.5 Section Alignment

There is no minimum alignment required for a section. Sections containing code must be at least 4-byte 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.

Build attributes are unnecessary when a platform ABI operating system is fully specified. At this time no public build attributes have been defined for AArch64, however, software development tools are free to use attributes privately. For an introduction to AArch32 build attributes see [Addenda32].

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`, `ELF64_ST_BIND()=STB_WEAK`.
- A *weak definition* — This is denoted by `st_shndx!=SHN_UNDEF`, `ELF64_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`. A linker is only required to provide long-branch and PLT support for symbols of type `STT_FUNC`.

4.5.3 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`.
- 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 (§4.5.4).
- The resulting object, or image, is not required to preserve accurate symbol information to permit de-compilation 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 tools use `.L`*xxx*).

4.5.3.1 Reserved symbol names

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

- Local symbols (STB_LOCAL) beginning with '$'
- Symbols matching the pattern *non-empty-prefix*$$*non-empty-suffix.*
- Global symbols (STB_GLOBAL, STB_WEAK) beginning with '`__aeabi_`' (double '_' at start).

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.

4.5.4 Mapping symbols

A section of an ELF file can contain a mixture of A64 code and data. There are inline transitions between code and data at literal pool boundaries.

Linkers, file decoders and other tools need to map binaries correctly. To support this, a number of symbols, termed *mapping symbols* appear in the symbol table to label the start of each sequence of bytes of the appropriate class. 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 4-4, 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. Its use minimizes string table size.
- A longer form in which 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.

Mapping symbols defined in a section (relocatable view) or segment (executable view) define a sequence of half-open intervals that cover the address range of the section or segment. 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 or segment. A section that contains instructions must have a mapping symbol defined at the beginning of the section. If a section contains only data no mapping symbol is required. A platform ABI should specify whether or not mapping symbols are present in the executable view; they will never be present in a *stripped* executable file.

Table 4-4, Mapping symbols

Name	Meaning
`$x` `$x.`<*any…*>	Start of a sequence of A64 instructions
`$d` `$d.`<*any…*>	Start of a sequence of data items (for example, a literal pool)

4.6 Relocation

Relocation information is used by linkers to bind symbols to addresses that could not be determined when the binary file was generated. Relocations are classified as *Static* or *Dynamic*.

- A *static relocation* relocates a place in an ELF relocatable file (`e_type` = `ET_REL`); a static linker processes it.
- A *dynamic relocation* is designed to relocate a place in an ELF executable file (`e_type` = `ET_EXEC`, `ET_DYN`) and to be handled by a dynamic linker, program loader, or other post-linking tool (*dynamic linker* henceforth).
- A dynamic linker need only process dynamic relocations; a static linker must handle any defined relocation.
- Dynamic relocations are designed to be processed quickly.
  - There are a small number of dynamic relocations whose codes are contiguous from 1024.
  - Dynamic relocations relocate simple places and do not need complex field extraction or insertion.
- A static linker either:
  - Fully resolves a relocation directive.
  - Or, generates a dynamic relocation from it for processing by a dynamic linker.
- A well formed executable file has no static relocations after static linking.

4.6.1 Relocation codes

The relocation codes for AArch64 are divided into four categories:

- Mandatory relocations that must be supported by all static linkers.
- Platform-specific relocations required by specific platform ABIs.
- 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.

4.6.2 Addends and PC-bias

A binary file may use `REL` or `RELA` relocations or a mixture of the two (but multiple relocations of the same place must use only one type).

The initial addend for a `REL`-type relocation is formed according to the following rules.

- If the relocation relocates data (§4.6.5) the initial value in the place is sign-extended to 64 bits.
- If the relocation relocates an instruction the immediate field of the instruction is extracted, scaled as required by the instruction field encoding, and sign-extended to 64 bits.

A `RELA` format relocation must be used if the initial addend cannot be encoded in the place.

There is no PC bias to accommodate in the relocation of a place containing an instruction that formulates a PC-relative address. The program counter reflects the address of the currently executing instruction.

4.6.3 Relocation types

Tables in the following sections list the relocation codes for AArch64 and record the following.

- The *relocation code* which is stored in the `ELF64_R_TYPE` component of the `r_info` field.
- The preferred mnemonic *name* for the relocation. This has no significance in a binary file.
- The *relocation operation* required. This field describes how a symbol and addend are processed by a linker. It does not describe how an initial addend value is extracted from a place (§4.6.2) or how the resulting relocated value is inserted or encoded into a place.
- A *comment* describing the kind of place that can be relocated, the part of the result value inserted into the place, and whether or not field overflow should be checked.

Static relocation codes begin at 256; dynamic ones at 1024. Both 0 and 256 should be accepted as values of R_AARCH64_NONE, the null relocation. All unallocated type codes are reserved for future allocation.

The following nomenclature is used in the descriptions of relocation operations:

- 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).
- Page(P) is the page address of the place being relocated, defined as (P & ~0xFFF).
- GOT is the address of the Global Offset Table, the table of code and data addresses to be resolved at dynamic link time.
- G (S) is the address of the GOT entry for the symbol S.

The value written into a target field is always reduced to fit the field. It is Q-o-I whether a linker generates a diagnostic when a relocated value overflows its target field.

Relocation types whose names end with "_NC" are *non-checking* relocation types. These *must not* generate diagnostics in case of field overflow. Usually, a non-checking type relocates an instruction that computes one of the less significant parts of a single value computed by a group of instructions (§4.6.8). Only the instruction computing the most significant part of the value can be checked for field overflow because, in general, a relocated value *will* overflow the fields of instructions computing the less significant parts.

4.6.4 Static miscellaneous relocations

`R_ARM_NONE` and `R_AARCH64_NONE` (null relocation codes) record 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 a static linker. The effect is to prevent removal of sections that might otherwise appear to be unused.

Table 4-5, Null relocation codes

Code	Name	Operation
0	`R_ARM_NONE`	`None`
256	`R_AARCH64_NONE`	`None`

4.6.5 Static Data relocations

See also Table 4-13, GOT-relative data relocations.

Table 4-6, Data relocations

Code	Name	Operation	Overflow check
257	R_AARCH64_ABS64	S + A	None
258	R_AARCH64_ABS32	S + A	-2147483648 ≤ result ≤ 4294967295
259	R_AARCH64_ABS16	S + A	-32768 ≤ result ≤ 65535
260	R_AARCH64_PREL64	S + A - P	None
261	R_AARCH64_PREL32	S + A - P	-2147483648 ≤ result ≤ 4294967295
262	R_AARCH64_PREL16	S + A - P	-32768 ≤ result ≤ 65535

These overflow ranges permit either signed or unsigned narrow values to be created from the intermediate result viewed as a 64-bit signed integer. If the place is intended to hold a narrow signed value and INT*n*_MAX < result ≤ UINT*n*_MAX, no overflow will be detected but the positive result will be interpreted as a negative value.

4.6.6 Static AArch64 relocations

The following tables record single instruction relocations and relocations that allow a group or sequence of instructions to compute a single relocated value.

Table 4-7, Group relocations to create a 16, 32, 48, or 64 bit unsigned data value or address inline

Note Non-checking (_NC) forms relocate MOVK; checking forms relocate MOVZ.

Code	Name	Operation	Comment
263	R_AARCH64_MOVW_UABS_G0	S + A	Set a MOVZ immediate field to bits 00000000 0000FFFF of S+A; check that S+A < 2^{16}
264	R_AARCH64_MOVW_UABS_G0_NC	S + A	Set a MOVK immediate field to bits 00000000 0000FFFF of S+A with no overflow check
265	R_AARCH64_MOVW_UABS_G1	S + A	Set a MOVZ immediate field to bits 00000000 FFFF0000 of S+A; check that S+A < 2^{32}
266	R_AARCH64_MOVW_UABS_G1_NC	S + A	Set a MOVK immediate field to bits 00000000 FFFF0000 of S+A with no overflow check
267	R_AARCH64_MOVW_UABS_G2	S + A	Set a MOVZ immediate field to bits 0000FFFF 00000000 of S+A; check that S+A < 2^{48}
268	R_AARCH64_MOVW_UABS_G2_NC	S + A	Set a MOVK immediate field to bits 0000FFFF 00000000 of S+A with no overflow check
269	R_AARCH64_MOVW_UABS_G3	S + A	Set a MOV[KZ] immediate field to bits FFFF0000 00000000 of S+A (no overflow check needed)

Table 4-8, Group relocations to create a 16, 32, 48, or 64 bit signed data or offset value inline

Note These checking forms relocate MOVN or MOVZ.

Code	Name	Operation	Comment
270	R_AARCH64_MOVW_SABS_G0	S + A	Set a MOV[NZ] immediate field using bits 00000000 0000FFFF of S+A (see notes below); check $-2^{16} \le S+A < 2^{16}$
271	R_AARCH64_MOVW_SABS_G1	S + A	Ses a MOV[NZ] immediate field using bits 00000000 FFFF0000 of S+A (see notes below); check $-2^{32} \le S+A < 2^{32}$
272	R_AARCH64_MOVW_SABS_G2	S + A	Set a MOV[NZ] immediate field using bits 0000FFFF 00000000 of S+A (see notes below); check $-2^{48} \le S+A < 2^{48}$

Note S+A ≥ 0: Set the instruction to MOVZ and its immediate field to the selected bits of S+A.

Note S+A < 0: Set the instruction to MOVN and its immediate field to NOT (selected bits of S+A).

Table 4-9, Relocations to generate 19, 21 and 33 bit PC-relative addresses

Code	Name	Operation	Comment
273	R_AARCH64_ LD_PREL_LO19	S + A - P	Set a load-literal immediate value to bits 00000000 001FFFFC of S+A-P; check that $-2^{20} \le S+A-P < 2^{20}$
274	R_AARCH64_ ADR_PREL_LO21	S + A - P	Set an ADR immediate value to bits 00000000 001FFFFF of S+A-P; check that $-2^{20} \le S+A-P < 2^{20}$
275	R_AARCH64_ ADR_PREL_PG_HI21	Page(S+A) -Page(P)	Set an ADRP immediate value to bits 00000001 FFFFF000 of the operation result; check that $-2^{32} \le$ *result* $< 2^{32}$
276	R_AARCH64_ ADR_PREL_PG_HI21_NC	Page(S+A) -Page(P)	Set an ADRP immediate value to bits 00000001 FFFFF000 of the result of the operation, with no overflow check
277	R_AARCH64_ ADD_ABS_LO12_NC	S + A	Set an ADD immediate value to bits 00000000 00000FFF of S+A, with no overflow check; used with relocations 275, 276
278	R_AARCH64_ LDST8_ABS_LO12_NC	S + A	Set an LD/ST immediate value to bits 00000000 00000FFF of S+A, with no overflow check; used with relocations 275, 276
284	R_AARCH64_ LDST16_ABS_LO12_NC	S + A	Set an LD/ST immediate value to bits 00000000 00000FFE of S+A, with no overflow check
285	R_AARCH64_ LDST32_ABS_LO12_NC	S + A	Set the LD/ST immediate value to bits 00000000 00000FFC of S+A, with no overflow check
286	R_AARCH64_ LDST64_ABS_LO12_NC	S + A	Set the LD/ST immediate value to bits 00000000 00000FF8 of S+A, with no overflow check
299	R_AARCH64_ LDST128_ABS_LO12_NC	S + A	Set the LD/ST immediate value to bits 00000000 00000FF0 of S+A, with no overflow check

Note Relocations 284, 285, 286 and 299 are intended to be used with R_AARCH64_ADR_PREL_PG_HI21 (275) so they pick out the low 12 bits of the address and, in effect, scale that by the access size. The increased address range provided by scaled addressing is not supported by these relocations because the extra range is unusable in conjunction with R_AARCH64_ADR_PREL_PG_HI21. Although overflow must not be checked, a linker *should* check that the value of S+A is aligned to a multiple of the datum size.

Table 4-10, Relocations for control-flow instructions - all offsets are a multiple of 4

Code	Name	Operation	Comment
279	R_AARCH64_TSTBR14	S+A-P	Set the immediate field of a TBZ/TBNZ instruction to bits 00000000 0000FFFC of S+A-P; check $-2^{15} \le$ S+A-P $< 2^{15}$
280	R_AARCH64_CONDBR19	S+A-P	Set the immediate field of a *conditional branch* instruction to bits 00000000 001FFFFC of S+A-P; check $-2^{20} \le$ S+A-P $< 2^{20}$
282	R_AARCH64_JUMP26	S+A-P	Set a B immediate field to bits 00000000 0FFFFFFC of S+A-P; check that $-2^{27} \le$ S+A-P $< 2^{27}$
283	R_AARCH64_CALL26	S+A-P	Set a CALL immediate field to bits 00000000 0FFFFFFC of S+A-P; check that $-2^{27} \le$ S+A-P $< 2^{27}$

Table 4-11, Group relocations to create a 16, 32, 48, or 64 bit PC-relative offset inline

Note Non-checking (_NC) forms relocate MOVK; checking forms relocate MOVN or MOVZ.

Code	Name	Operation	Comment
287	R_AARCH64_MOVW_PREL_G0	S+A-P	Set a MOV[NZ]immediate field to bits 00000000 0000FFFF of X=S+A-P (see below)
288	R_AARCH64_MOVW_PREL_G0_NC	S+A-P	Set a MOVK immediate field to bits 00000000 0000FFFF of S+A-P, with no overflow check
289	R_AARCH64_MOVW_PREL_G1	S+A-P	Set a MOV[NZ]immediate field to bits 00000000 FFFF0000 of X=S+A-P (see notes below)
290	R_AARCH64_MOVW_PREL_G1_NC	S+A-P	Set a MOVK immediate field to bits 00000000 FFFF0000 of S+A-P, with no overflow check
291	R_AARCH64_MOVW_PREL_G2	S+A-P	Set a MOV[NZ]immediate value to bits 0000FFFF 00000000 of X=S+A-P (see notes below)
292	R_AARCH64_MOVW_PREL_G2_NC	S+A-P	Set a MOVK immediate field to bits 0000FFFF 00000000 of S+A-P, with no overflow check
293	R_AARCH64_MOVW_PREL_G3	S+A-P	Set a MOV[NZ]immediate value to bits FFFF0000 00000000 of X=S+A-P (see below)

Note X ≥ 0: Set the instruction to MOVZ and its immediate value to the selected bits of X; for relocation R_…_G*n*, check that X < {G0: 2^{16}, G1: 2^{32}, G2: 2^{48}} (no check for R_..._G3).

Note X < 0: Set the instruction to MOVN and its immediate value to NOT (selected bits of X); for relocation R_…_G*n*, check that -{G0: 2^{16}, G1: 2^{32}, G2: 2^{48}} ≤ X (no check for R_..._G3).

Table 4-12, Group relocations to create a 16, 32, 48, or 64 bit GOT-relative offsets inline

Note Non-checking (_NC) forms relocate MOVK; checking forms relocate MOVN or MOVZ.

Code	Name	Operation	Comment
300	R_AARCH64_MOVW_GOTOFF_G0	G(S)-GOT	Set a MOV[NZ] immediate field to bits 00000000 0000FFFF of X=G(S)-GOT (see above)

Code	Name	Operation	Comment
301	R_AARCH64_MOVW_GOTOFF_G0_NC	G(S)-GOT	Set a MOVK immediate field to bits 00000000 0000FFFF of G(S)-GOT, with no overflow check
302	R_AARCH64_MOVW_GOTOFF_G1	G(S)-GOT	Set a MOV[NZ] immediate value to bits 00000000 FFFF0000 of X=G(S)-GOT (see notes above)
303	R_AARCH64_MOVW_GOTOFF_G1_NC	G(S)-GOT	Set a MOVK immediate value to bits 00000000 FFFF0000 of X=G(S)-GOT, with no overflow check
304	R_AARCH64_MOVW_GOTOFF_G2	G(S)-GOT	Set a MOV[NZ] immediate value to bits 0000FFFF 00000000 of X=G(S)-GOT (see above)
305	R_AARCH64_MOVW_GOTOFF_G2_NC	G(S)-GOT	Set a MOVK immediate value to bits 0000FFFF 00000000 of G(S)-GOT, with no overflow check
306	R_AARCH64_MOVW_GOTOFF_G3	G(S)-GOT	Set a MOV[NZ] immediate value to bits FFFF0000 00000000 of X=G(S)-GOT (see above)

Table 4-13, GOT-relative data relocations

Code	Name	Operation	Comment
307	R_AARCH64_GOTREL64	S+A-GOT	Set the data to a 64-bit offset relative to GOT, treated as signed
308	R_AARCH64_GOTREL32	S+A-GOT	Set the data to a 32-bit offset relative to GOT, treated as signed; check that $-2^{32} \le$ S+A-GOT $< 2^{32}$

Table 4-14, GOT-relative instruction relocations

Code	Name	Operation	Comment
309	R_AARCH64_GOT_LD_PREL19	G(S)-P	Set a load-literal immediate field to bits 00000000 001FFFFC of G(S)-P; check $-2^{20} \le$ G(S)-P $< 2^{20}$
310	R_AARCH64_LD64_GOTOFF_LO15	G(S)-GOT	Set a LD/ST immediate field to bits 00000000 00007FF8 of X=G(S)-GOT; check that $0 \le X < 2^{15}$, X&7 = 0
311	R_AARCH64_ADR_GOT_PAGE	Page(G(S))-Page(P)	Set the immediate value of an ADRP to bits 00000001 FFFFF000 of X=Page(G(S))-Page(GOT); check that $-2^{32} \le X < 2^{32}$
312	R_AARCH64_LD64_GOT_LO12_NC	G(S)	Set the LD/ST immediate field to bits 00000000 00000FF8 of G(S), with no overflow check; check that X&7 = 0
313	R_AARCH64_LD64_GOTPAGE_LO15	G(S)-Page(GOT)	Set the LD/ST immediate field to bits 00000000 00007FF8 of X=G(S)-Page(GOT); check that $0 \le X < 2^{15}$, X&7 = 0

4.6.7 Call and Jump relocations

There is one relocation code (R_AARCH64_CALL26) for function call (BL) instructions and one (R_AARCH64_JUMP26) for jump (B) instructions.

A linker may use a veneer (a sequence of instructions) to implement a relocated branch if the relocation is either R_AARCH64_CALL26 or R_AARCH64_JUMP26 and:

- The target symbol has type STT_FUNC.
- Or, the target symbol and relocated place are in separate sections input to the linker.
- Or, the target symbol is undefined (external to the link unit).

In all other cases a linker shall diagnose an error if relocation cannot be effected without a veneer. A linker generated veneer may corrupt registers IP0 and IP1 [AAPCS64] and the condition flags, but must preserve all other registers. 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 (+/- 128MB).
- Target address will not be known until run time, or the target address might be pre-empted.

In some systems indirect calls may also use veneers in order to support dynamic linkage that preserves pointer comparability (all reference to the function resolve to the same address).

On platforms that do not support dynamic pre-emption of symbols an unresolved weak reference to a symbol relocated by R_AARCH64_CALL26 shall be treated as a jump to the next instruction (the call becomes a no-op). The behaviour of R_AARCH64_JUMP26 in these conditions is not specified by this standard.

4.6.8 Group relocations

A relocation code whose name ends in _G*n* or _G*n*_NC (*n* = 0, 1, 2, 3) relocates an instruction in a group of instructions that generate a single value or address (see Table 4-7, Table 4-8, Table 4-11, Table 4-12). Each such relocation relocates one instruction in isolation, with no need to determine all members of the group at link time.

These relocations operate by performing the relocation calculation then extracting a field from the result X. Generating the field for a G*n* relocation directive starts by examining the residual value Y*n* after the bits of abs(X) corresponding to less significant fields have been masked off from X. If M is the mask specified in the table recording the relocation directive, Y*n* = abs(X) & ~((M & -M) – 1).

Overflow checking is performed on Y*n* unless the name of the relocation ends in "_NC".

Finally the bit-field of X specified in the table (those bits of X picked out by 1-bits in M) is encoded into the instruction's literal field as specified in the table. In some cases other instruction bits may need to be changed according to the sign of X.

4.6.9 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 4-12, *Group relocations to create a 16, 32, 48, or 64 bit GOT-relative offsets inline* and

Table 4-14, *GOT-relative instruction relocations* list the relocations that generate proxy entries.

All of the GOT entries generated by these relocations are subject to dynamic relocations (§Note, *For scaled-addressing* relocations 554-559, a linker should check that X is a multiple of the datum size.

Dynamic relocations).

4.6.10 Relocations for thread-local storage

The static relocations needed to support thread-local storage in a SVr4-type environment are listed in tables in the following subsections

In addition to the terms defined in §4.6.3, *Relocation types*, the tables listing the static relocations relating to thread-local storage use the following terms in the column named *Operation (X = …)*.

- `LDM(S)` represents the 64-bit load module index of a thread local variable.
- `TLSIDX(S+A)` represents the index in the `GOT` of a tls_index structure describing the thread-local variable located at offset A from thread-local symbol S. A tls_index structure holds a 64-bit load module index and a 64-bit offset from the origin of that module's thread-local data area of the thread-local variable.
- `DTPREL(S+A)` resolves to the offset from its module's TLS block of the thread local variable located at offset A from thread-local symbol S.
- `TPREL(S+A)` resolves to the offset from the current thread pointer (TP) of the thread local variable located at offset A from thread-local symbol S.
- `G(`*`expression`*`)` requests the allocation of a GOT entry, or entries, containing *expression* and resolves to the address of that GOT entry.

4.6.10.1 General Dynamic thread-local storage model

Table 4-15, General Dynamic TLS relocations

Note Non-checking (`_NC`) `MOVW` forms relocate `MOVK`; checking forms relocate `MOVN` or `MOVZ`.

Code	Name	Operation (X = …)	Comment
512	R_AARCH64_TLSGD_ ADR_PREL21	G(TLSIDX(S+A)) - P	Set an ADR immediate field to bits 00000000 001FFFFF of X; check $-2^{20} \le X < 2^{20}$
513	R_AARCH64_TLSGD_ ADR_PAGE21	Page(G(TLSIDX(S+A))) - Page(P)	Set an ADRP immediate field to bits 00000001 FFFFF000 of X; check $-2^{32} \le X < 2^{32}$
514	R_AARCH64_TLSGD_ ADD_LO12_NC	G(TLSIDX(S+A))	Set an ADD immediate field to bits 00000000 00000FFF of X, with no overflow check
515	R_AARCH64_TLSGD_ MOVW_G1	G(TLSIDX(S+A)) - GOT	Set a MOV[NZ] immediate field to bits 00000000 FFFF0000 of X (see notes below)
516	R_AARCH64_TLSGD_ MOVW_G0_NC	G(TLSIDX(S+A)) - GOT	Set a MOVK immediate field to bits 00000000 0000FFFF of X, with no overflow check

Note $X \ge 0$: Set the instruction to `MOVZ` and its immediate value to the selected bits of X; check that $X < 2^{32}$.

Note $X < 0$: Set the instruction to `MOVN` and its immediate value to NOT (selected bits of X); check that $-2^{32} \le X$.

4.6.10.2 Local Dynamic thread-local storage model

Table 4-16, Local Dynamic TLS relocations

Note Non-checking (`_NC`) `MOVW` forms relocate `MOVK`; checking forms relocate `MOVN` or `MOVZ`.

Code	Name	Operation (X = …)	Comment
517	R_AARCH64_TLSLD_ ADR_PREL21	G(LDM(S)) - P	Set an ADR immediate field to bits 00000000 001FFFFF of X; check $-2^{20} \le X < 2^{20}$

Code	Name	Operation (X = …)	Comment
518	R_AARCH64_TLSLD_ ADR_PAGE21	Page(G(LDM(S))) -Page(P)	Set an ADRP immediate field to bits 00000001 FFFFF000 of X; check $-2^{32} \le X < 2^{32}$
519	R_AARCH64_TLSLD_ ADD_LO12_NC	G(LDM(S))	Set an ADD immediate field to bits 00000000 00000FFF of X, with no overflow check
520	R_AARCH64_TLSLD_ MOVW_G1	G(LDM(S)) - GOT	Set a MOV[NZ] immediate field to bits 00000000 FFFF0000 of X (notes above)
521	R_AARCH64_TLSLD_ MOVW_G0_NC	G(LDM(S)) - GOT	Set a MOVK immediate field to bits 00000000 0000FFFF of X, with no overflow check
522	R_AARCH64_TLSLD_ LD_PREL19	G(LDM(S)) - P	Set a load-literal immediate field to bits 00000000 001FFFFC of X; check $-2^{20} \le X < 2^{20}$
523	R_AARCH64_TLSLD_ MOVW_DTPREL_G2	DTPREL(S+A)	Set a MOV[NZ] immediate field to bits 0000FFFF 00000000 of X (notes below)
524	R_AARCH64_TLSLD_ MOVW_DTPREL_G1	DTPREL(S+A)	Set a MOV[NZ] immediate field to bits 00000000 FFFF0000 of X (notes below)
525	R_AARCH64_TLSLD_ MOVW_DTPREL_G1_NC	DTPREL(S+A)	Set a MOVK immediate field to bits 00000000 FFFF0000 of X, with no overflow check
526	R_AARCH64_TLSLD_ MOVW_DTPREL_G0	DTPREL(S+A)	Set a MOV[NZ] immediate field to bits 00000000 0000FFFF of X (notes below)
527	R_AARCH64_TLSLD_ MOVW_DTPREL_G0_NC	DTPREL(S+A)	Set a MOVK immediate field to bits 00000000 0000FFFF of X, with no overflow check
528	R_AARCH64_TLSLD_ ADD_DTPREL_HI12	DTPREL(S+A)	Set an ADD immediate field to bits 00000000 00FFF000 of X; check $0 \le X < 2^{24}$
529	R_AARCH64_TLSLD_ ADD_DTPREL_LO12	DTPREL(S+A)	Set an ADD immediate field to bits 00000000 00000FFF of X; check $0 \le X < 2^{12}$
530	R_AARCH64_TLSLD_ ADD_DTPREL_LO12_NC	DTPREL(S+A)	Set an ADD immediate field to bits 00000000 00000FFF of X, with no overflow check
531	R_AARCH64_TLSLD_ LDST8_DTPREL_LO12	DTPREL(S+A)	Set a LD/ST offset field to bits 00000000 00000FFF of X; check $0 \le X < 2^{12}$
532	R_AARCH64_TLSLD_ LDST8_DTPREL_LO12_NC	DTPREL(S+A)	Set a LD/ST offset field to bits 00000000 00000FFF of X, with no overflow check
533	R_AARCH64_TLSLD_ LDST16_DTPREL_LO12	DTPREL(S+A)	Set a LD/ST offset field to bits 00000000 00000FFE of X; check $0 \le X < 2^{12}$
534	R_AARCH64_TLSLD_ LDST16_DTPREL_LO12_NC	DTPREL(S+A)	Set a LD/ST offset field to bits 00000000 00000FFE of X, with no overflow check
535	R_AARCH64_TLSLD_ LDST32_DTPREL_LO12	DTPREL(S+A)	Set a LD/ST offset field to bits 00000000 00000FFC of X; check $0 \le X < 2^{12}$
536	R_AARCH64_TLSLD_ LDST32_DTPREL_LO12_NC	DTPREL(S+A)	Set a LD/ST offset field to bits 00000000 00000FFC of X, with no overflow check

Code	Name	Operation (X = …)	Comment
537	R_AARCH64_TLSLD_ LDST64_DTPREL_LO12	DTPREL(S+A)	Set a LD/ST offset field to bits 00000000 00000FF8 of X; check 0 ≤ X < 2^{12}
538	R_AARCH64_TLSLD_ LDST64_DTPREL_LO12_NC	DTPREL(S+A)	Set a LD/ST offset field to bits 00000000 00000FF8 of X, with no overflow check

Note X ≥ 0: Set the instruction to MOVZ and its immediate value to the selected bits S; for relocation R_…_G*n*, check that X < {G0: 2^{16}, G1: 2^{32}, G2: 2^{48}} (no check for R_. . ._G3).

Note X < 0: Set the instruction to MOVN and its immediate value to NOT (selected bits of); for relocation R_…_G*n*, check that -{G0: 2^{16}, G1: 2^{32}, G2: 2^{48}} ≤ X (no check for R_. . ._G3).

Note For scaled-addressing relocations 533-538, a linker should check that X is a multiple of the datum size.

4.6.10.3 Initial Exec thread-local storage model

Table 4-17, Initial Exec TLS relocations

Note Non-checking (_NC) MOVW forms relocate MOVK; checking forms relocate MOVN or MOVZ.

Code	Name	Operation (X = …)	Comment
539	R_AARCH64_TLSIE_ MOVW_GOTTPREL_G1	G(TPREL(S+A)) – GOT	Set a MOV[NZ] immediate field to bits 00000000 FFFF0000 of X (see notes above)
540	R_AARCH64_TLSIE_ MOVW_GOTTPREL_G0_NC	G(TPREL(S+A)) – GOT	Set MOVK immediate to bits 00000000 0000FFFF of X, with no overflow check
541	R_AARCH64_TLSIE_ ADR_GOTTPREL_PAGE21	Page(G(TPREL(S+A))) – Page(P)	Set an ADRP immediate field to bits 00000001 FFFFF000 of X; check -2^{32} ≤ X < 2^{32}
542	R_AARCH64_TLSIE_ LD64_GOTTPREL_LO12_NC	G(TPREL(S+A))	Set an LD offset field to bits 00000000 00000FF8 of X, with no overflow check; check that X&7=0
543	R_AARCH64_TLSIE_ LD_GOTTPREL_PREL19	G(TPREL(S+A)) – P	Set a load-literal immediate to bits 00000000 001FFFFC of X; check -2^{20} ≤ X < 2^{20}

4.6.10.4 Local Exec thread-local storage model

Table 4-18, Local Exec TLS relocations

Note Non-checking (_NC) MOVW forms relocate MOVK; checking forms relocate MOVN or MOVZ.

Code	Name	Operation (X = …)	Comment
544	R_AARCH64_TLSLE_ MOVW_TPREL_G2	TPREL(S+A)	Set a MOV[NZ] immediate field to bits 0000FFFF 00000000 of X (see notes above)
545	R_AARCH64_TLSLE_ MOVW_TPREL_G1	TPREL(S+A)	Set a MOV[NZ] immediate field to bits 00000000 FFFF0000 of X (see notes above)
546	…_MOVW_TPREL_G1_NC	TPREL(S+A)	Set a MOVK immediate field to bits 00000000 FFFF0000 of X, with no overflow check
547	R_AARCH64_TLSLE_ MOVW_TPREL_G0	TPREL(S+A)	Set a MOV[NZ] immediate field to bits 00000000 0000FFFF of X (see notes above)

Code	Name	Operation (X = …)	Comment
548	…_MOVW_TPREL_G0_NC	TPREL(S+A)	Set a MOVK immediate field to bits 00000000 0000FFFF of X, with no overflow check
549	R_AARCH64_TLSLE_ ADD_TPREL_HI12	TPREL(S+A)	Set an ADD immediate field to bits 00000000 00FFF000 of X; check $0 \le X < 2^{24}$.
550	R_AARCH64_TLSLE_ ADD_TPREL_LO12	TPREL(S+A)	Set an ADD immediate field to bits 00000000 00000FFF of X; check $0 \le X < 2^{12}$.
551	…_ADD_TPREL_LO12_NC	TPREL(S+A)	Set an ADD immediate field to bits 00000000 00000FFF of X, with no overflow check
552	R_AARCH64_TLSLE_ LDST8_TPREL_LO12	TPREL(S+A)	Set a LD/ST offset field to bits 00000000 00000FFF of X; check $0 \le X < 2^{12}$.
553	R_AARCH64_TLSLE_ LDST8_TPREL_LO12_NC	TPREL(S+A)	Set a LD/ST offset field to bits 00000000 00000FFF of X, with no overflow check
554	R_AARCH64_TLSLE_ LDST16_TPREL_LO12	TPREL(S+A)	Set a LD/ST offset field to bits 00000000 00000FFE of X; check $0 \le X < 2^{12}$
555	R_AARCH64_TLSLE_ LDST16_TPREL_LO12_NC	TPREL(S+A)	Set a LD/ST offset field to bits 00000000 00000FFE of X, with no overflow check
556	R_AARCH64_TLSLE_ LDST32_TPREL_LO12	TPREL(S+A)	Set a LD/ST offset field to bits 00000000 00000FFC of X; check $0 \le X < 2^{12}$
557	R_AARCH64_TLSLE_ LDST32_TPREL_LO12_NC	TPREL(S+A)	Set a LD/ST offset field to bits 00000000 00000FFC of X, with no overflow check
558	R_AARCH64_TLSLE_ LDST64_TPREL_LO12	TPREL(S+A)	Set a LD/ST offset field to bits 00000000 00000FF8 of X; check $0 \le X < 2^{12}$
559	R_AARCH64_TLSLE_ LDST64_TPREL_LO12_NC	TPREL(S+A)	Set a LD/ST offset field to bits 00000000 00000FF8 of X, with no overflow check

Note For scaled-addressing relocations 554-559, a linker should check that X is a multiple of the datum size.

4.6.11 Dynamic relocations

The dynamic relocations for those execution environments that support only a limited number of run-time relocation types are listed in *Table 4-19, Dynamic relocations*. The enumeration of dynamic relocations commences at 1024 and the range is compact.

Table 4-19, Dynamic relocations

Code	Name	Operation (X=...)	Comment
1024	R_AARCH64_COPY		See note below.
1025	R_AARCH64_GLOB_DAT	S + A	Resolves to the address of the specified symbol
1026	R_AARCH64_JUMP_SLOT	S + A	Resolves to the address of the specified symbol (see note below)

Code	Name	Operation (X=...)	Comment
1027	R_AARCH64_RELATIVE	ΔS + A ΔP + A	If S ≠ 0 (see notes below) If S = 0 (see notes below)
1028	R_AARCH64_TLS_DTPREL64	DTPREL(S+A)	
1029	R_AARCH64_TLS_DTPMOD64	LDM(S)	
1030	R_AARCH64_TLS_TPREL64	TPREL(S+A)	
1031	R_AARCH64_TLS_DTPREL32	DTPREL(S+A)	1028 with a 32-bit offset
1032	R_AARCH64_TLS_DTPMOD32	LDM(S)	1029 with a 32-bit offset
1033	R_AARCH64_TLS_TPREL32	DTPREL(S+A)	1030 with a 32-bit offset

With the exception of R_AARCH64_COPY and R_...32 (1031-1033) all dynamic relocations require that the place being relocated is an 8-byte aligned 64-bit data location.

R_AARCH64_COPY may only appear in executable ELF files 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 its 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.

R_AARCH64_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, this is difficult to achieve without source-code annotation. A better approach is to avoid defining static global data in shared libraries.

R_AARCH64_GLOB_DAT relocates a GOT entry used to hold the address of a (data) symbol which must be resolved at load time.

R_AARCH64_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 initial value stored in the place is the offset to the entry sequence stub for the dynamic linker. It 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 after has been resolved.
- Because the initial value of the place is not related to the ultimate target of an R_AARCH64_JUMP_SLOT relocation the *addend* A of such a REL-type relocation shall be zero rather than the initial content of the place. A platform ABI shall prescribe whether or not the r_addend field of such a RELA-type relocation is honored. (There may be security-related reasons not to do so).

R_AARCH64_RELATIVE (S ≠ 0) resolves to the difference between the address at which the segment defining the symbol S was loaded and the address at which it was statically linked.

R_AARCH64_RELATIVE (S = 0) resolves to the difference between the address at which the segment being relocated (the one containing the place) was loaded and the address at which it was statically linked.

4.6.12 Private relocations

Relocation codes 0xE000-0xFFFF denote *private relocations*. These codes will not be assigned by any future version of this standard and may be used by a tool chain for its own private purposes.

4.6.13 Unallocated relocations

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

4.6.14 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 cannot be idempotent because applying the relocation destroys the original addend.

5 PROGRAM LOADING AND DYNAMIC LINKING

This section provides details of AArch64-specific definitions and changes relating to executable images.

5.1 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 [SCO-ELF]. The following fields have AArch64-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_AARCH64_ARCHEXT	0x70000000	*Reserved for architecture compatibility information*
PT_AARCH64_UNWIND	0x70000001	*Reserved for exception unwinding tables*

A segment of type PT_AARCH64_ARCHEXT (if present) contains information describing the architecture capabilities required by the executable file. Not all platform ABIs require this segment; the Linux ABI does not. If the segment is present it must appear before segment of type PT_LOAD.

PT_AARCH64_UNWIND (if present) describes the location of a program's exception unwind tables.

p_flags

There are no AArch64-specific flags.

5.1.1 Platform architecture compatibility data

At this time this ABI specifies no generic *platform architecture compatibility* data.

5.2 Program Loading

There are no AArch64-specific definitions relating to program loading.

5.3 Dynamic Linking

5.3.1 Dynamic Section

There are no AArch64-specific dynamic array tags.

5.4 Program headers for SVr4 (Linux) pre-linking

These need to be defined.