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

2022Q3

Date of Issue: 20th October 2022

arm

1 Preamble

1.1 ILP32 Beta

This document includes a beta proposal for ILP32 extensions to ELF for AArch64.

Feedback welcome through your normal channels.

1.2 Abstract

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

1.3 Keywords

ELF, AArch64 ELF, ...

1.4 Latest release and defects report

Please check Application Binary Interface for the Arm® Architecture for the latest release of this document.

Please report defects in this specification to the issue tracker page on GitHub.

1.5 Licence

1.6 About the license

As identified more fully in the Licence section, this project is licensed under CC-BY-SA-4.0 along with an additional patent license. The language in the additional patent license is largely identical to that in Apache-2.0 (specifically, Section 3 of Apache-2.0 as reflected at https://www.apache.org/licenses/LICENSE-2.0) with two exceptions.

First, several changes were made related to the defined terms so as to reflect the fact that such defined terms need to align with the terminology in CC-BY-SA-4.0 rather than Apache-2.0 (e.g., changing “Work” to “Licensed Material”).

Second, the defensive termination clause was changed such that the scope of defensive termination applies to “any licenses granted to You” (rather than “any patent licenses granted to You”). This change is intended to help maintain a healthy ecosystem by providing additional protection to the community against patent litigation claims.

1.7 Contributions

Contributions to this project are licensed under an inbound=outbound model such that any such contributions are licensed by the contributor under the same terms as those in the Licence section.

1.8 Trademark notice

1.9 Copyright

Contents

2 About this document

2.1 Change control

2.1.1 Current status and anticipated changes

The following support level definitions are used by the Arm ABI specifications:

Release

Arm considers this specification to have enough implementations, which have received sufficient testing, to verify that it is correct. The details of these criteria are dependent on the scale and complexity of the change over previous versions: small, simple changes might only require one implementation, but more complex changes require multiple independent implementations, which have been rigorously tested for cross-compatibility. Arm anticipates that future changes to this specification will be limited to typographical corrections, clarifications and compatible extensions.

Beta

Arm considers this specification to be complete, but existing implementations do not meet the requirements for confidence in its release quality. Arm may need to make incompatible changes if issues emerge from its implementation.

Alpha

The content of this specification is a draft, and Arm considers the likelihood of future incompatible changes to be significant.

The ELF32 variant is at "Beta" release quality.

All other content in this document is at the **Release** quality level.

2.1.2 Change history

If there is no entry in the change history table for a release, there are no changes to the content of the document for that release.

Issue	Date	Change
00bet3	20th December 2011	Beta release
1.0	22nd May 2013	First public release
1.1-beta	6th November 2013	ILP32 Beta
2018Q4	31st December 2018	Typographical changes
2019Q1	29th March 2019	Add Program Property for BTI and PAC. Update `MOV[ZK]` related relocations.
2019Q2	30th June 2019	Specify `STO_AARCH64_VARIANT_PCS`. Update `R_<CLS>_TLS_DTPREL` and `R_<CLS>_TLS_DTPMOD`. Clarify `GNU_PROPERTY_AARCH64_FEATURE_1_AND`.
2019Q4	30th January 2020	Minor layout changes.
2020Q2	1st July 2020	Specifiy `R_<CLS>_PLT32`. Correct minus sign not rendering in section Group relocations. Adjust table widths for readability.

Issue	Date	Change
2020Q3	1st October 2020	• document released on Github • new Licence: CC-BY-SA-4.0 • new sections on Contributions, Trademark notice, and Copyright
2021Q1	12th April 2021	• Typo fix in definition of GTPREL expression in section Relocations for thread-local storage • Typo fix of EI_OSABI in ELF Identification • Typo fixes -220 -> -2^20 in section Thread-local storage descriptors
2021Q3	1st November 2021	• Reserved relocation codes for PAuthABIELF64
2022Q1	1st April 2022	• In Program Property, Soft-deprecate GNU_PROPERTY_AARCH64_FEATURE_1_PAC
2022Q3	20th October 2022	• In Dynamic relocations, include the ABS64 and ABS32 relocations in Dynamic relocations. • In Relocation optimization, ADRP + LDR GOT relaxation symbol should not be absolute. • In Program Loading and Dynamic Linking, document new PT_AARCH64_MEMTAG_MTE segment.

2.2 References

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

Ref	External reference or URL	Title
AAELF64	Source for this document	ELF for the Arm 64-bit Architecture (AArch64).
AAPCS64	IHI 0055	Procedure Call Standard for the Arm 64-bit Architecture
Addenda32	IHI 0045	Addenda to, and Errata in, the ABI for the Arm Architecture
PAuthABIELF64	pauthabielf64	PAuth Extension to ELF for the Arm 64-bit Architecture
LSB	http://www.linuxbase.org/	Linux Standards Base
SCO-ELF	http://www.sco.com/developers/gabi/	System V Application Binary Interface – DRAFT
LINUX_ABI	https://github.com/hjl-tools/linux-abi/wiki	Linux Extensions to gABI

Ref	External reference or URL	Title
SYM-VER	http://people.redhat.com/drepper/symbol-versioning	GNU Symbol Versioning
TLSDESC	http://www.fsfla.org/~lxoliva/writeups/TLS/paper-lk2006.pdf	TLS Descriptors for Arm. Original proposal document
MTEEXTENSIONS	https://www.kernel.org/doc/html/latest/arm64/memory-tagging-extension.html#core-dump-support	Linux Kernel MTE core dump format

2.3 Terms and abbreviations

The ABI for the Arm 64-bit Architecture uses the following terms and abbreviations:

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 Addenda32, AAPCS64, ...

Arm-based

... based on the Arm architecture ...

ELF32

An ELF object file with a class of ELFCLASS32

ELF64

An ELF object file with a class of ELFCLASS64

ILP32

SysV-like data model where int, long int and pointer are 32-bit

LP64

SysV-like data model where int is 32-bit, but long int and pointer are 64-bit.

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.

T32

The instruction set named Thumb in the Armv7 architecture; T32 uses 16-bit and 32-bit instructions.

Other terms may be defined when first used.

3 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 Unix System V (SysV) ABI specification where it forms Object Files and Program Loading and Dynamic Linking. However, the ELF specification can be used in isolation as a generic object and executable format. Platform standards (Example Only) covers ELF related matters that are platform specific.

Object Files and Program Loading and Dynamic Linking are structured to correspond to chapters 4 and 5 of the ELF specification. Specifically:

- Object Files covers object files and relocations
- Program Loading and Dynamic Linking covers program loading and dynamic linking.

3.1 ELF Class variants

Two different pointer sizes are supported by this specification, which result in two very similar but different ELF definitions.

3.1.1 64-bit Pointers, ELF64

- Code and data using 64-bit pointers are contained in an ELF object file with a class of **ELFCLASS64**.
- Referred to as **ELF64** in this specification.
- Pointer-size is **64 bits**.
- Suitable for use by the LP64 variant of [AAPCS64]

3.1.2 32-bit Pointers, ELF32 (Beta)

- Code and data using 32-bit pointers is contained in an ELF object file with a class of **ELFCLASS32**.
- Referred to as **ELF32** in this specification.
- Pointer-size is **32 bits**.
- Suitable for use by the ILP32 variant of [AAPCS64]

> ***Note***
>
> Interlinking is not supported between the ELF32 and ELF64 variants.

4 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, Linux Platform ABI (example only) lists some of the issues addressed by the *Linux Standard Base* [LSB] specifications.

4.1 Linux Platform ABI (example only)

4.1.1 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].

4.1.2 Program Linkage Table (PLT) Sequences and Usage Models

4.1.2.1 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.

SysV-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 (Call and Jump relocations). For a Linux/SysV DSO, each `STB_GLOBAL` symbol with `STV_DEFAULT` visibility is a candidate.

4.1.2.2 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 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.

5 Object Files

5.1 Introduction

5.1.1 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 the below table. Tools developers not listed are requested to co-ordinate with Arm to avoid the potential for conflicts.

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
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.

5.2 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 shall contain zero.

5.2.1 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`

For object files (executable, shared and relocatable) the **EI_CLASS** shall be:

- `ELFCLASS64` for an ELF64 object file.
- `ELFCLASS32` for an ELF32 object file **(Beta)**.

`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`).

5.3 Sections

5.3.1 Special Section Indexes

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

5.3.2 Section Types

The defined processor-specific section types are listed in the below table. All other processor-specific values are reserved to future revisions of this specification.

Processor specific section types

Name	Value	Comment
`SHT_AARCH64_ATTRIBUTES`	`0x70000003`	Reserved for Object file compatibility attributes

5.3.3 Section Attribute Flags

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

5.3.3.1 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.

5.3.4 Special Sections

The below table lists the special sections defined by this ABI.

AArch64 special sections

Name	Type	Attributes
`.ARM.attributes`	`SHT_AARCH64_ATTRIBUTES`	none
.note.gnu.property	`SHT_NOTE`	`SHF_ALLOC`

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

`.note.gnu.property` names a section that holds a program property note. See [LINUX_ABI] for more information.

Additional special sections may be required by some platforms standards.

5.3.5 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 minimum page size supported by the platform).

5.3.6 Build Attributes

Build attributes are encoded in a section of type `SHT_AARCH64_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].

5.4 String Table

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

5.5 Symbol Table

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

5.5.1 `st_other` Values

The `st_other` member of a symbol table entry specifies the symbol's visibility in the lowest 2 bits. The top 6 bits are unused in the generic ELF ABI [SCO-ELF], and while there are no values reserved for processor-specific semantics, many other architectures have used these bits.

The defined processor-specific `st_other` flag values are listed below:

Processor specific `st_other` flags

Name	Mask	Comment
`STO_AARCH64_VARIANT_PCS`	0x80	The function associated with the symbol may follow a variant procedure call standard with different register usage convention.

A symbol table entry that is marked with the `STO_AARCH64_VARIANT_PCS` flag set in its `st_other` field may be associated with a function that follows a variant procedure call standard with different register usage convention from the one defined in the base procedure call standard for the list of argument, caller-saved and callee-saved registers [AAPCS64]. The rules in the Call and Jump relocations section still apply to such functions. If a subroutine is called via a symbol reference that is marked with `STO_AARCH64_VARIANT_PCS`, then code that runs between the calling routine and the called subroutine must preserve the contents of all registers except for IP0, IP1, and the condition code flags [AAPCS64].

Static linkers must preserve the marking and propagate it to the dynamic symbol table if any reference or definition of the symbol is marked with `STO_AARCH64_VARIANT_PCS`, and add a `DT_AARCH64_VARIANT_PCS` dynamic tag if required by the Dynamic Section section.

> ***Note***
>
> In particular, when a call is made via the PLT entry of a symbol marked with `STO_AARCH64_VARIANT_PCS`, a dynamic linker cannot assume that the call follows the register usage convention of the base procedure call standard.
>
> An example of a function that follows a variant procedure call standard with different register usage convention is one that takes parameters in scalable vector or predicate registers.

5.6 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`.
  - `st_shndx=SHN_UNDEF, ELF32_ST_BIND()=STB_WEAK` **(Beta)**.
- A weak definition — This is denoted by:
  - `st_shndx!=SHN_UNDEF, ELF64_ST_BIND()=STB_WEAK`.
  - `st_shndx!=SHN_UNDEF, ELF32_ST_BIND()=STB_WEAK` **(Beta)**.

5.6.1 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 symbol 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.

See Relocation for further details.

5.6.2 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.

5.6.3 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`.

5.6.4 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`.
- 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 (Mapping symbols).
- 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 `Lxxx.yyy`, while GNU tools use `.Lxxx`).

5.6.4.1 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***
>
> Global symbols beginning with '__vendor_' (double '_' at start), where vendor is listed in Registered Vendor Names are reserved to the named vendor for the purpose of providing vendor-specific toolchain support functions.

5.6.5 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 the Mapping symbols table. 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.

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)

5.7 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 or dynamic shared object (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.
    - 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 or dynamic shared object has no static relocations after static linking.

5.7.1 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.

5.7.2 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 (Static Data relocations) 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.

5.7.3 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` or `ELF32_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 (Addends and PC-bias) 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.

5.7.3.1 Relocation names and class

A mnemonic name class is used to distinguish between ELF64 and ELF32 relocation names.

- ELF64 relocations have <CLS> = AARCH64, e.g. R_AARCH64_ABS32
- ELF32 relocations have <CLS> = AARCH64_P32, where P32 denotes the pointer size, e.g. R_AARCH64_P32_ABS32 **(Beta)**

> **Note**
>
> Within this document <CLS> is not expanded in instances where only a single relocation name exists.

5.7.3.2 Relocation codes disambiguation

References to relocation codes are disambiguated in the following way:

- ELF64 relocation codes are bounded by parentheses: ( ).
- ELF32 relocation codes are bounded by brackets: [ ].

Static relocation codes for ELF64 object files begin at (257); dynamic ones at (1024). Both (0) and (256) should be accepted as values of R_AARCH64_NONE, the null relocation.

Static relocation codes for ELF32 object files begin at [1]; dynamic ones at [180].

All unallocated type codes are reserved for future allocation.

5.7.3.3 Relocation operations

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).
- X is the result of a relocation operation, before any masking or bit-selection operation is applied
- Page(expr) is the page address of the expression expr, defined as (expr & ~0xFFF). (This applies even if the machine page size supported by the platform has a different value.)
- GOT is the address of the Global Offset Table, the table of code and data addresses to be resolved at dynamic link time. The GOT and each entry in it must be, 64-bit aligned for ELF64 or 32-bit aligned for ELF32.
- GDAT(S+A) represents a pointer-sized entry in the GOT for address S+A. The entry will be relocated at run time with relocation R_<CLS>_GLOB_DAT(S+A).
- G(expr) is the address of the GOT entry for the expression expr.
- Delta(S) if S is a normal symbol, resolves to the difference between the static link address of S and the execution address of S. If S is the null symbol (ELF symbol index 0), resolves to the difference between the static link address of P and the execution address of P.
- Indirect(expr) represents the result of calling expr as a function. The result is the return value from the function that is returned in r0. The arguments passed to the function are defined by the platform ABI.
- [msb:lsb] is a bit-mask operation representing the selection of bits in a value. The bits selected range from lsb up to msb inclusive. For example, ‘bits [3:0]’ represents the bits under the mask 0x0000000F. When range checking is applied to a value, it is applied before the masking operation is performed.

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 (Group relocations). 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. Some non-checking relocations may, however, be expected to check for correct alignment of the result; the notes explain when this is permitted. In ELF32 relocations an overflow check of $-2^{31} <= X < 2^{31}$ or $0 <= X < 2^{31}$ is equivalent to no check (i.e. 'None').

In ELF32 **(Beta)** relocations additional care must be taken when relocating an ADRP instruction which effectively uses a signed 33-bit PC-relative offset to generate a 32-bit address. The following relocations apply to ADRP:

```
R_<CLS>_ADR_PREL_PG_HI21,
R_<CLS>_ADR_GOT_PAGE,
R_<CLS>_TLSGD_ADR_PAGE21,
R_<CLS>_TLSLD_ADR_PAGE21,
R_<CLS>_TLSIE_ADR_GOTTPREL_PAGE21,
R_<CLS>_TLSDESC_ADR_PAGE21
```

5.7.4 Static miscellaneous relocations

R_<CLS>_NONE (null relocation code) 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 a static linker. The effect is to prevent removal of sections that might otherwise appear to be unused.

Null relocation codes

ELF64 Code	ELF32 Code	Name	Operation	Comment
0	0	R_<CLS>_NONE	None	
256	-	withdrawn	None	Treat as R_<CLS>_NONE.

5.7.5 Static Data relocations

See also table GOT-relative data relocations.

Data relocations

ELF64 Code	ELF32 Code	Name	Operation	Comment
257	-	R_<CLS>_ABS64	S + A	No overflow check
258	1	R_<CLS>_ABS32	S + A	Check that $-2^{31} <= X < 2^{32}$
259	2	R_<CLS>_ABS16	S + A	Check that $-2^{15} <= X < 2^{16}$
260	-	R_<CLS>_PREL64	S + A - P	No overflow check
261	3	R_<CLS>_PREL32	S + A - P	Check that $-2^{31} <= X < 2^{32}$
262	4	R_<CLS>_PREL16	S + A - P	Check that $-2^{15} <= X < 2^{16}$
314	29	R_<CLS>_PLT32	S + A - P	Check that $-2^{31} <= X < 2^{31}$ see call and jump relocations

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

`INTn_MAX < X <= UINTn_MAX`, no overflow will be detected but the positive result will be interpreted as a negative value.

5.7.6 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.

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

ELF64 Code	ELF32 Code	Name	Operation	Comment
263	5	R_<CLS>_MOVW_UABS_G0	S + A	Set a MOV[KZ] immediate field to bits [15:0] of X; check that 0 <= X < 2^{16}
264	6	R_<CLS>_MOVW_UABS_G0_NC	S + A	Set a MOV[KZ] immediate field to bits [15:0] of X. No overflow check
265	7	R_<CLS>_MOVW_UABS_G1	S + A	Set a MOV[KZ] immediate field to bits [31:16] of X; check that 0 <= X < 2^{32}
266	-	R_<CLS>_MOVW_UABS_G1_NC	S + A	Set a MOV[KZ] immediate field to bits [31:16] of X. No overflow check
267	-	R_<CLS>_MOVW_UABS_G2	S + A	Set a MOV[KZ] immediate field to bits [47:32] of X; check that 0 <= X < 2^{48}
268	-	R_<CLS>_MOVW_UABS_G2_NC	S + A	Set a MOV[KZ] immediate field to bits [47:32] of X. No overflow check
269	-	R_<CLS>_MOVW_UABS_G3	S + A	Set a MOV[KZ] immediate field to bits [63:48] of X (no overflow check needed)

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

ELF64 Code	ELF32 Code	Name	Operation	Comment
270	8	R_<CLS>_MOVW_SABS_G0	S + A	Set a MOV[NZ] immediate field using bits [15:0] of X (see notes below); check -2^{16} <= X < 2^{16}
271	-	R_<CLS>_MOVW_SABS_G1	S + A	Set a MOV[NZ] immediate field using bits [31:16] of X (see notes below); check -2^{32} <= X < 2^{32}
272	-	R_<CLS>_MOVW_SABS_G2	S + A	Set a MOV[NZ] immediate field using bits [47:32] of X (see notes below); check -2^{48} <= X < 2^{48}

Note

These checking forms relocate `MOVN` or `MOVZ`.

X >= 0: Set the instruction to `MOVZ` and its immediate field to the selected bits of X.

X < 0: Set the instruction to `MOVN` and its immediate field to NOT (selected bits of X).

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

ELF64 Code	ELF32 Code	Name	Operation	Comment
273	9	R_<CLS>_ LD_PREL_LO19	S + A - P	Set a load-literal immediate value to bits [20:2] of X; check that -2^{20} <= X < 2^{20}
274	10	R_<CLS>_ ADR_PREL_LO21	S + A - P	Set an ADR immediate value to bits [20:0] of X; check that -2^{20} <= X < 2^{20}
275	11	R_<CLS>_ ADR_PREL_PG_HI21	Page(S+A)-Page(P)	Set an ADRP immediate value to bits [32:12] of the X; check that -2^{32} <= X < 2^{32}
276	-	R_<CLS>_ ADR_PREL_PG_HI21_NC	Page(S+A)-Page(P)	Set an ADRP immediate value to bits [32:12] of the X. No overflow check
277	12	R_<CLS>_ ADD_ABS_LO12_NC	S + A	Set an ADD immediate value to bits [11:0] of X. No overflow check. Used with relocations ADR_PREL_PG_HI21 and ADR_PREL_PG_HI21_NC
278	13	R_<CLS>_ LDST8_ABS_LO12_NC	S + A	Set an LD/ST immediate value to bits [11:0] of X. No overflow check. Used with relocations ADR_PREL_PG_HI21 and ADR_PREL_PG_HI21_NC
284	14	R_<CLS>_ LDST16_ABS_LO12_NC	S + A	Set an LD/ST immediate value to bits [11:1] of X. No overflow check
285	15	R_<CLS>_ LDST32_ABS_LO12_NC	S + A	Set the LD/ST immediate value to bits [11:2] of X. No overflow check
286	16	R_<CLS>_ LDST64_ABS_LO12_NC	S + A	Set the LD/ST immediate value to bits [11:3] of X. No overflow check
299	17	R_<CLS>_ LDST128_ABS_LO12_NC	S + A	Set the LD/ST immediate value to bits [11:4] of X. No overflow check

Note

Relocations (284, 285, 286 and 299) or [14, 15, 16, 17] are intended to be used with `R_<CLS>_ADR_PREL_PG_HI21` (275) or [11] 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_<CLS>_ADR_PREL_PG_HI21`.

Although overflow must not be checked, a linker should check that the value of X is aligned to a multiple of the datum size.

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

ELF64 Code	ELF32 Code	Name	Operation	Comment
279	18	R_<CLS>_TSTBR14	S+A-P	Set the immediate field of a TBZ/TBNZ instruction to bits [15:2] of X; check -2^{15} <= X < 2^{15}
280	19	R_<CLS>_CONDBR19	S+A-P	Set the immediate field of a conditional branch instruction to bits [20:2] of X; check -2^{20} <= X< 2^{20}
282	20	R_<CLS>_JUMP26	S+A-P	Set a B immediate field to bits [27:2] of X; check that -2^{27} <= X < 2^{27}
283	21	R_<CLS>_CALL26	S+A-P	Set a CALL immediate field to bits [27:2] of X; check that -2^{27} <= X < 2^{27}

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

ELF64 Code	ELF32 Code	Name	Operation	Comment
287	22	R_<CLS>_MOVW_PREL_G0	S+A-P	Set a MOV[NZ]immediate field to bits [15:0] of X (see notes below)
288	23	R_<CLS>_MOVW_PREL_G0_NC	S+A-P	Set a MOVK immediate field to bits [15:0] of X. No overflow check
289	24	R_<CLS>_MOVW_PREL_G1	S+A-P	Set a MOV[NZ]immediate field to bits [31:16] of X (see notes below)
290	-	R_<CLS>_MOVW_PREL_G1_NC	S+A-P	Set a MOVK immediate field to bits [31:16] of X. No overflow check
291	-	R_<CLS>_MOVW_PREL_G2	S+A-P	Set a MOV[NZ]immediate value to bits [47:32] of X (see notes below)
292	-	R_<CLS>_MOVW_PREL_G2_NC	S+A-P	Set a MOVK immediate field to bits [47:32] of X. No overflow check
293	-	R_<CLS>_MOVW_PREL_G3	S+A-P	Set a MOV[NZ]immediate value to bits [63:48] of X (see notes below)

Note

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

`X >= 0`: Set the instruction to `MOVZ` and its immediate value to the selected bits of X; for relocation `R_..._Gn`, check in ELF64 that X < {`G0:` 2^{16}, `G1:` 2^{32}, `G2:` 2^{48}} (no check for `R_..._G3`); in ELF32 only check X < 2^{16} for `R_..._G0`.

`X < 0`: Set the instruction to `MOVN` and its immediate value to NOT (selected bits of X); for relocation `R_..._Gn`, check in ELF64 that -{`G0:` 2^{16}, `G1:` 2^{32}, `G2:` 2^{48}} <= X (no check for `R_..._G3`); in ELF32 only check that -2^{16} <= X for R_..._G0.

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

ELF64 Code	ELF32 Code	Name	Operation	Comment
300	-	R_<CLS>_MOVW_GOTOFF_G0	G(GDAT(S+A)) -GOT	Set a MOV[NZ] immediate field to bits [15:0] of X (see notes below)
301	-	R_<CLS>_MOVW_GOTOFF_G0_NC	G(GDAT(S+A)) -GOT	Set a MOVK immediate field to bits [15:0] of X. No overflow check
302	-	R_<CLS>_MOVW_GOTOFF_G1	G(GDAT(S+A)) -GOT	Set a MOV[NZ] immediate value to bits [31:16] of X (see notes below)
303	-	R_<CLS>_MOVW_GOTOFF_G1_NC	G(GDAT(S+A)) -GOT	Set a MOVK immediate value to bits [31:16] of X. No overflow check
304	-	R_<CLS>_MOVW_GOTOFF_G2	G(GDAT(S+A)) -GOT	Set a MOV[NZ] immediate value to bits [47:32] of X (see notes below)
305	-	R_<CLS>_MOVW_GOTOFF_G2_NC	G(GDAT(S+A)) -GOT	Set a MOVK immediate value to bits [47:32] of X. No overflow check
306	-	R_<CLS>_MOVW_GOTOFF_G3	G(GDAT(S+A)) -GOT	Set a MOV[NZ] immediate value to bits [63:48] of X (see notes below)

Note

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

GOT-relative data relocations

ELF64 Code	ELF32 Code	Name	Operation	Comment
307	-	R_<CLS>_GOTREL64	S+A-GOT	Set the data to a 64-bit offset relative to the GOT.
308	-	R_<CLS>_GOTREL32	S+A-GOT	Set the data to a 32-bit offset relative to GOT, treated as signed; check that -2^{31} <= X < 2^{31}

GOT-relative instruction relocations

ELF64 Code	ELF32 Code	Name	Operation	Comment
309	25	R_<CLS>_GOT_LD_PREL19	G(GDAT(S+A))- P	Set a load-literal immediate field to bits [20:2] of X; check -2^{20} <= X < 2^{20}

ELF64 Code	ELF32 Code	Name	Operation	Comment
310	-	R_<CLS>_LD64_GOTOFF_LO15	G(GDAT(S+A))- GOT	Set a LD/ST immediate field to bits [14:3] of X; check that 0 <= X < 2^{15}, X&7 = 0
311	26	R_<CLS>_ADR_GOT_PAGE	Page(G(GDAT(S+A)))-Page(P)	Set the immediate value of an ADRP to bits [32:12] of X; check that -2^{32} <= X < 2^{32}
312	-	R_<CLS>_LD64_GOT_LO12_NC	G(GDAT(S+A))	Set the LD/ST immediate field to bits [11:3] of X. No overflow check; check that X&7 = 0
-	27	R_<CLS>_LD32_GOT_LO12_NC	G(GDAT(S+A))	Set the LD/ST immediate field to bits [11:2] of X. No overflow check; check that X&3 = 0
313	-	R_<CLS>_LD64_GOTPAGE_LO15	G(GDAT(S+A))-Page(GOT)	Set the LD/ST immediate field to bits [14:3] of X; check that 0 <= X < 2^{15}, X&7 = 0
-	28	R_<CLS>_LD32_GOTPAGE_LO14	G(GDAT(S+A))-Page(GOT)	Set the LD/ST immediate field to bits [13:2] of X; check that 0 <= X < 2^{14}, X&3 = 0

5.7.7 Call and Jump relocations

There is one relocation code (`R_<CLS>_CALL26`) for function call (`BL`) instructions and one (`R_<CLS>_JUMP26`) for jump (`B`) instructions. The (`R_<CLS>_PLT32`) relocation is a data relocation for calculating the offset to a function. This can be used as the target of an indirect jump.

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

`R_<CLS>_CALL26`, `R_<CLS>_JUMP26` or `R_<CLS>_PLT32` 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_<CLS>_CALL26 shall be treated as a jump to the next instruction (the call becomes a no-op). The behaviour of R_<CLS>_JUMP26 and R_<CLS>_PLT32 in these conditions is not specified by this standard.

5.7.8 Group relocations

A relocation code whose name ends in _Gn or _Gn_NC (n = 0, 1, 2, 3) relocates an instruction in a group of instructions that generate a single value or address (see tables unsigned inline group relocations, signed inline group relocations, PC-relative inline relocations, GOT-relative inline relocations). 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 Gn relocation directive starts by examining the residual value Yn 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, Yn = abs(X) & ~((M & -M) - 1).

Overflow checking is performed on Yn 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.

For "MOVW" type relocations it is the assembler's responsibility to encode the hw bits (bits 21 and 22) to indicate the bits in the target value that the immediate field represents.

5.7.9 Relocation optimization

Linkers may optionally optimize instructions affected by relocation. Relocation optimizations improve the efficiency of relocated instructions without changing their visible behaviour. There are several classes of relocation optimizations:

- A single relocation optimization may change an instruction after relocation into an equivalent, more efficient form.
  - Several relocations may result in an addition with zero, which may be optimized as follows:

```
ADD   x0, x1, 0   // eg. R_<CLS>_TLSLE_ADD_TPREL_HI12
ADD   x2, x2, 0   // or R_<CLS>_ADD_ABS_LO12_NC

// after optimization:

MOV   x0, x1
NOP
```

  - The relocation R_<CLS>_ADR_PREL_PG_HI21 may emit a MOV with zero immediate for undefined weak symbols.
  - The following TLS relocations may be optimized if the symbol is not a pre-emptable definition and the TLS offset fits in 16 bits:

```
ADRP  x0, :gottprel: symbol            // R_<CLS>_TLSIE_ADR_GOTTPREL_PAGE21
LDR   x1, [x0, :gottprel_lo12: symbol] // R_<CLS>_TLSIE_LD64_GOTTPREL_LO12_NC
LDR   x2, :gottprel: symbol            // R_<CLS>_TLSIE_LD_GOTTPREL_PREL19

// after optimization:

NOP
MOV   x1, :tprel_g0: symbol            // R_<CLS>_TLSLE_MOVW_TPREL_G0
MOV   x2, :tprel_g0: symbol            // R_<CLS>_TLSLE_MOVW_TPREL_G0
```

    If a linker supports optimizing R_<CLS>_TLSIE_ADR_GOTTPREL_PAGE21, it must also support optimizing R_<CLS>_TLSIE_LD64_GOTTPREL_LO12_NC.

- A sequence of relocated instructions may be optimized if all of the following conditions are true:
  - The relocations apply to consecutive instructions in the order specified.
  - The relocations use the same symbol.
  - The relocated instructions have the same source and destination register.
  - The relocations do not appear separately or in a different order.

  In this case each set of relocations is independent and may be optimized. The following sequences are defined:

  - Large GOT indirection

    A GOT indirection may be optimized into PC-relative addressing:

    ```
    ADRP  x0, :got: symbol             // R_<CLS>_ADR_GOT_PAGE
    LDR   x0, [x0 :got_lo12: symbol]   // R_<CLS>_LD64_GOT_LO12_NC

    // after optimization:

    ADRP  x0, symbol                   // R_<CLS>_ADR_PREL_PG_HI21
    ADD   x0, x0, :lo12: symbol        // R_<CLS>_ADD_ABS_LO12_NC
    ```

    This sequence may be optimized if it meets all of the following conditions:

    - `symbol` is not a pre-emptable definition.
    - `symbol` is not of type `STT_GNU_IFUNC`.
    - `symbol` does not have a `st_shndx` of `SHN_ABS` or the output is not required to be position independent.
    - `symbol` is within range of the `R_<CLS>_ADR_PREL_PG_HI21` relocation.
    - The addend of both relocations is zero.

    The optimized sequence does not require a GOT entry. A linker may avoid creating a GOT entry if no other GOT relocations exist for the symbol.

  - PC-relative addressing

    `ADR` may replace `ADRP/ADD` if `symbol` is within +-1MiB range:

    ```
    ADRP  x0, symbol                   // R_<CLS>_ADR_PREL_PG_HI21
    ADD   x0, x0, :lo12: symbol        // R_<CLS>_ADD_ABS_LO12_NC

    // after optimization:

    NOP
    ADR   x0, symbol                   // R_<CLS>_ADR_PREL_LO21
    ```

5.7.10 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 GOT-relative inline relocations and table 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 (Dynamic relocations).

5.7.11 Relocations for thread-local storage

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

In addition to the terms defined in Relocation types, the tables listing the static relocations relating to thread-local storage use the following terms in the column named Operation.

- `GLDM(S)` represents a consecutive pair of pointer-sized entries in the GOT for the load module index of the symbol `S`. The first pointer-sized entry will be relocated with `R_<CLS>_TLS_DTPMOD(S);` the second pointer-sized entry will contain the constant 0.
- `GTLSIDX(S,A)` represents a consecutive pair of pointer-sized entries in the GOT. The entry contains a `tls_index` structure describing the thread-local variable located at offset `A` from thread-local symbol `S`. The first pointer-sized entry will be relocated with `R_<CLS>_TLS_DTPMOD(S)`, the second pointer-sized entry will be relocated with `R_<CLS>_TLS_DTPREL(S+A)`.
- `GTPREL(S+A)` represents a pointer-sized entry in the GOT for the offset from the current thread pointer (TP) of the thread-local variable located at offset `A` from the symbol `S`. The entry will be relocated with `R_<CLS>_TLS_TPREL(S+A)`.
- `GTLSDESC(S+A)` represents a consecutive pair of pointer-sized entries in the GOT which contain a tlsdesc structure describing the thread-local variable located at offset `A` from thread-local symbol `S`. The first entry holds a pointer to the variable's TLS descriptor resolver function and the second entry holds a platform-specific offset or pointer. The pair of pointer-sized entries will be relocated with `R_<CLS>_TLSDESC(S+A)`.
- `LDM(S)` resolves to the load module index of the symbol `S`.
- `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`.
- `TLSDESC(S+A)` resolves to a contiguous pair of pointer-sized values, as created by `GTLSDESC(S+A)`.

5.7.11.1 General Dynamic thread-local storage model

General Dynamic TLS relocations

ELF64 Code	ELF32 Code	Name	Operation	Comment
512	80	R_<CLS>_TLSGD_ ADR_PREL21	G(GTLSIDX(S,A)) - P	Set an ADR immediate field to bits [20:0] of X; check -2^{20} <= X < 2^{20}
513	81	R_<CLS>_TLSGD_ ADR_PAGE21	Page(G(GTLSIDX(S,A)) - Page(P)	Set an ADRP immediate field to bits [32:12] of X; check -2^{32} <= X < 2^{32}
514	82	R_<CLS>_TLSGD_ ADD_LO12_NC	G(GTLSIDX(S,A))	Set an ADD immediate field to bits [11:0] of X. No overflow check
515	-	R_<CLS>_TLSGD_ MOVW_G1	G(GTLSIDX(S,A)) - GOT	Set a MOV[NZ] immediate field to bits [31:16] of X (see notes below)

ELF64 Code	ELF32 Code	Name	Operation	Comment
516	-	R_<CLS>_TLSGD_ MOVW_G0_NC	G(GTLSIDX(S,A)) - GOT	Set a MOVK immediate field to bits [15:0] of X. No overflow check

Note

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

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

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

5.7.11.2 Local Dynamic thread-local storage model

Local Dynamic TLS relocations

ELF64 Code	ELF32 Code	Name	Operation	Comment
517	83	R_<CLS>_TLSLD_ ADR_PREL21	G(GLDM(S))) - P	Set an ADR immediate field to bits [20:0] of X; check $–2^{20} <= X < 2^{20}$
518	84	R_<CLS>_TLSLD_ ADR_PAGE21	Page(G(GLDM(S)))-Page(P)	Set an ADRP immediate field to bits [32:12] of X; check $–2^{32} <= X < 2^{32}$
519	85	R_<CLS>_TLSLD_ ADD_LO12_NC	G(GLDM(S))	Set an ADD immediate field to bits [11:0] of X. No overflow check
520	-	R_<CLS>_TLSLD_ MOVW_G1	G(GLDM(S)) - GOT	Set a MOV[NZ] immediate field to bits [31:16] of X (see notes below)
521	-	R_<CLS>_TLSLD_ MOVW_G0_NC	G(GLDM(S)) - GOT	Set a MOVK immediate field to bits [15:0] of X. No overflow check
522	86	R_<CLS>_TLSLD_ LD_PREL19	G(GLDM(S)) - P	Set a load-literal immediate field to bits [20:2] of X; check $–2^{20} <= X < 2^{20}$
523	-	R_<CLS>_TLSLD_ MOVW_DTPREL_G2	DTPREL(S+A)	Set a MOV[NZ] immediate field to bits [47:32] of X (see notes below)
524	87	R_<CLS>_TLSLD_ MOVW_DTPREL_G1	DTPREL(S+A)	Set a MOV[NZ] immediate field to bits [31:16] of X (see notes below)
525	-	R_<CLS>_TLSLD_ MOV W_DTPREL_G1_NC	DTPREL(S+A)	Set a MOVK immediate field to bits [31:16] of X. No overflow check

ELF64 Code	ELF32 Code	Name	Operation	Comment
526	88	R_<CLS>_TLSLD_ MOVW_DTPREL_G0	DTPREL(S+A)	Set a MOV[NZ] immediate field to bits [15:0] of X (see notes below)
527	89	R_<CLS>_TLSLD_ MOVW_DTPREL_G0_NC	DTPREL(S+A)	Set a MOVK immediate field to bits [15:0] of X. No overflow check
528	90	R_<CLS>_TLSLD_ ADD_DTPREL_HI12	DTPREL(S+A)	Set an ADD immediate field to bits [23:12] of X; check 0 <= X < 2^{24}
529	91	R_<CLS>_TLSLD_ ADD_DTPREL_LO12	DTPREL(S+A)	Set an ADD immediate field to bits [11:0] of X; check 0 <= X < 2^{12}
530	92	R_<CLS>_TLSLD_ ADD_DTPREL_LO12_NC	DTPREL(S+A)	Set an ADD immediate field to bits [11:0] of X. No overflow check
531	93	R_<CLS>_TLSLD_ LDST8_DTPREL_LO12	DTPREL(S+A)	Set a LD/ST offset field to bits [11:0] of X; check 0 <= X < 2^{12}
532	94	R_<CLS>_TLSLD_ LDST8_DTPREL_LO12_NC	DTPREL(S+A)	Set a LD/ST offset field to bits [11:0] of X. No overflow check
533	95	R_<CLS>_TLSLD_ LDST16_DTPREL_LO12	DTPREL(S+A)	Set a LD/ST offset field to bits [11:1] of X; check 0 <= X < 2^{12}
534	96	R_<CLS>_TLSLD_ LDST16_DTPREL_LO12_NC	DTPREL(S+A)	Set a LD/ST offset field to bits [11:1] of X. No overflow check
535	97	R_<CLS>_TLSLD_ LDST32_DTPREL_LO12	DTPREL(S+A)	Set a LD/ST offset field to bits [11:2] of X; check 0 <= X < 2^{12}
536	98	R_<CLS>_TLSLD_ LDST32_DTPREL_LO12_NC	DTPREL(S+A)	Set a LD/ST offset field to bits [11:2] of X. No overflow check
537	99	R_<CLS>_TLSLD_ LDST64_DTPREL_LO12	DTPREL(S+A)	Set a LD/ST offset field to bits [11:3] of X; check 0 <= X < 2^{12}
538	100	R_<CLS>_TLSLD_ LDST64_DTPREL_LO12_NC	DTPREL(S+A)	Set a LD/ST offset field to bits [11:3] of X. No overflow check
572	101	R_<CLS>_TLSLD_ LDST128_DTPREL_LO12	DTPREL(S+A)	Set a LD/ST offset field to bits [11:4] of X; check 0 <= X < 2^{12}
573	102	R_<CLS>_TLSLD_ LDST128_DTPREL_LO12_NC	DTPREL(S+A)	Set a LD/ST offset field to bits [11:4] of X. No overflow check

Note

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

`X >= 0`: Set the instruction to `MOVZ` and its immediate value to the selected bits S; for relocation `R_..._Gn`, check in ELF64 that X < {`G0:` 2^{16}, `G1:` 2^{32}, `G2:` 2^{48}} (no check for `R_..._G3`); in ELF32 only check that X < 2^{16} for `R_..._G0`.

`X < 0`: Set the instruction to `MOVN` and its immediate value to NOT (selected bits of); for relocation `R_..._Gn`, check in ELF64 that -{`G0:` 2^{16}, `G1:` 2^{32}, `G2:` 2^{48}} <= X (no check for `R_..._G3`); in ELF32 only check that -2^{16} <= X for `R_..._G0`.

For scaled-addressing relocations (533-538, 572 and 573) or [95-102] a linker should check that X is a multiple of the datum size.

5.7.11.3 Initial Exec thread-local storage model

Initial Exec TLS relocations

ELF64 Code	ELF32 Code	Name	Operation	Comment
539	-	R_<CLS>_TLSIE_ MOVW_GOTTPREL_G1	G(GTPREL(S+A)) - GOT	Set a MOV[NZ] immediate field to bits [31:16] of X (see notes below)
540	-	R_<CLS>_TLSIE_ MOVW_GOTTPREL_G0_NC	G(GTPREL(S+A)) - GOT	Set MOVK immediate to bits [15:0] of X. No overflow check
541	103	R_<CLS>_TLSIE_ ADR_GOTTPREL_PAGE21	Page(G(GTPREL(S+A))) - Page(P)	Set an ADRP immediate field to bits [32:12] of X; check –2^{32} <= X < 2^{32}
542	-	R_<CLS>_TLSIE_ LD64_GOTTPREL_LO12_NC	G(GTPREL(S+A))	Set an LD offset field to bits [11:3] of X. No overflow check; check that X&7=0
-	104	R_<CLS>_TLSIE_ LD32_GOTTPREL_LO12_NC	G(GTPREL(S+A))	Set an LD offset field to bits [11:2] of X. No overflow check; check that X&3=0
543	105	R_<CLS>_TLSIE_ LD_GOTTPREL_PREL19	G(GTPREL(S+A)) – P	Set a load-literal immediate to bits [20:2] of X; check –2^{20} <= X < 2^{20}

Note

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

5.7.11.4 Local Exec thread-local storage model

Local Exec TLS relocations

ELF64 Code	ELF32 Code	Name	Operation	Comment
544	-	R_<CLS>_TLSLE_ MOVW_TPREL_G2	TPREL(S+A)	Set a MOV[NZ] immediate field to bits [47:32] of X (see notes below)
545	106	R_<CLS>_TLSLE_ MOVW_TPREL_G1	TPREL(S+A)	Set a MOV[NZ] immediate field to bits [31:16] of X (see notes below)
546	-	R_<CLS>_TLSLE_ MOVW_TPREL_G1_NC	TPREL(S+A)	Set a MOVK immediate field to bits [31:16] of X. No overflow check
547	107	R_<CLS>_TLSLE_ MOVW_TPREL_G0	TPREL(S+A)	Set a MOV[NZ] immediate field to bits [15:0] of X (see notes below)
548	108	R_<CLS>_TLSLE_ MOVW_TPREL_G0_NC	TPREL(S+A)	Set a MOVK immediate field to bits [15:0] of X. No overflow check
549	109	R_<CLS>_TLSLE_ ADD_TPREL_HI12	TPREL(S+A)	Set an ADD immediate field to bits [23:12] of X; check 0 <= X < 2^{24}.
550	110	R_<CLS>_TLSLE_ ADD_TPREL_LO12	TPREL(S+A)	Set an ADD immediate field to bits [11:0] of X; check 0 <= X < 2^{12}.
551	111	R_<CLS>_TLSLE_ ADD_TPREL_LO12_NC	TPREL(S+A)	Set an ADD immediate field to bits [11:0] of X. No overflow check
552	112	R_<CLS>_TLSLE_ LDST8_TPREL_LO12	TPREL(S+A)	Set a LD/ST offset field to bits [11:0] of X; check 0 <= X < 2^{12}.
553	113	R_<CLS>_TLSLE_ LDST8_TPREL_LO12_NC	TPREL(S+A)	Set a LD/ST offset field to bits [11:0] of X. No overflow check
554	114	R_<CLS>_TLSLE_ LDST16_TPREL_LO12	TPREL(S+A)	Set a LD/ST offset field to bits [11:1] of X; check 0 <= X < 2^{12}
555	115	R_<CLS>_TLSLE_ LDST16_TPREL_LO12_NC	TPREL(S+A)	Set a LD/ST offset field to bits [11:1] of X. No overflow check
556	116	R_<CLS>_TLSLE_ LDST32_TPREL_LO12	TPREL(S+A)	Set a LD/ST offset field to bits [11:2] of X; check 0 <= X < 2^{12}

ELF64 Code	ELF32 Code	Name	Operation	Comment
557	117	R_<CLS>_TLSLE_ LDST32_TPREL_LO12_NC	TPREL(S+A)	Set a LD/ST offset field to bits [11:2] of X. No overflow check
558	118	R_<CLS>_TLSLE_ LDST64_TPREL_LO12	TPREL(S+A)	Set a LD/ST offset field to bits [11:3] of X; check 0 <= X < 2^{12}
559	119	R_<CLS>_TLSLE_ LDST64_TPREL_LO12_NC	TPREL(S+A)	Set a LD/ST offset field to bits [11:3] of X. No overflow check
570	120	R_<CLS>_TLSLE_ LDST128_TPREL_LO12	TPREL(S+A)	Set a LD/ST offset field to bits [11:4] of X; check 0 <= X < 2^{12}
571	121	R_<CLS>_TLSLE_ LDST128_TPREL_LO12_NC	TPREL(S+A)	Set a LD/ST offset field to bits [11:4] of X. No overflow check

Note

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

For scaled-addressing relocations (554-559, 570 and 571) or [112-121] a linker should check that X is a multiple of the datum size.

5.7.11.5 Thread-local storage descriptors

TLS descriptor relocations

ELF64 Code	ELF32 Code	Name	Operation	Comment
560	122	R_<CLS>_TLSDESC_ LD_PREL19	G(GTLSDESC(S+A)) - P	Set a load-literal immediate to bits [20:2]; check -2^{20} <= X < 2^{20}; check X & 3 = 0.
561	123	R_<CLS>_TLSDESC_ ADR_PREL21	G(GTLSDESC(S+A)) - P	Set an ADR immediate field to bits [20:0]; check -2^{20} <= X < 2^{20}.
562	124	R_<CLS>_TLSDESC_ ADR_PAGE21	Page(G(GTLSDESC(S+A))) - Page(P)	Set an ADRP immediate field to bits [32:12] of X; check -2^{32} <= X < 2^{32}.
563	-	R_<CLS>_TLSDESC_ LD64_LO12	G(GTLSDESC(S+A))	Set an LD offset field to bits [11:3] of X. No overflow check; check X & 7 = 0.
-	125	R_<CLS>_TLSDESC_ LD32_LO12	G(GTLSDESC(S+A))	Set an LD offset field to bits [11:2] of X. No overflow check; check X & 3 = 0.

ELF64 Code	ELF32 Code	Name	Operation	Comment
564	126	R_<CLS>_TLSDESC_ADD_LO12	G(GTLSDESC(S+A))	Set an ADD immediate field to bits [11:0] of X. No overflow check.
565	-	R_<CLS>_TLSDESC_OFF_G1	G(GTLSDESC(S+A)) - GOT	Set a MOV[NZ] immediate field to bits [31:16] of X; check -2^{32} <= X < 2^{32}. See notes below.
566	-	R_<CLS>_TLSDESC_OFF_G0_NC	G(GTLSDESC(S+A)) - GOT	Set a MOVK immediate field to bits [15:0] of X. No overflow check.
567	-	R_<CLS>_TLSDESC_LDR	None	For relaxation only. Must be used to identify an LDR instruction which loads the TLS descriptor function pointer for S + A if it has no other relocation.
568	-	R_<CLS>_TLSDESC_ADD	None	For relaxation only. Must be used to identify an ADD instruction which computes the address of the TLS Descriptor for S + A if it has no other relocation.
569	127	R_<CLS>_TLSDESC_CALL	None	For relaxation only. Must be used to identify a BLR instruction which performs an indirect call to the TLS descriptor function for S + A.

Note

`X >= 0`: Set the instruction to MOVZ and its immediate value to the selected bits of X.

`X < 0`: Set the instruction to MOVN and its immediate value to NOT (selected bits of X).

Relocation codes `R_<CLS>_TLSDESC_LDR`, `R_<CLS>_TLSDESC_ADD` and `R_<CLS>_TLSDESC_CALL` are needed to permit linker optimization of TLS descriptor code sequences to use Initial-exec or Local-exec TLS sequences; this can only be done if all relevant uses of TLS descriptors are marked to permit accurate relaxation. Object producers that are unable to satisfy this requirement must generate traditional General-dynamic TLS sequences using the relocations described in General Dynamic thread-local storage model. The details of TLS descriptors are beyond the scope of this specification; a general introduction can be found in [TLSDESC].

5.7.12 Dynamic relocations

The dynamic relocations for those execution environments that support only a limited number of run-time relocation types are listed in the below table. The enumeration of dynamic relocations commences at (1024) or [180] and the range is compact.

Dynamic relocations

ELF64 Code	ELF32 Code	Name	Operation	Comment
257	-	R_<CLS>_ABS64	S + A	See note below.
-	1	R_<CLS>_ABS32	S + A	See note below.

ELF64 Code	ELF32 Code	Name	Operation	Comment
1024	180	R_<CLS>_COPY		See note below.
1025	181	R_<CLS>_GLOB_DAT	S + A	See note below
1026	182	R_<CLS>_JUMP_SLOT	S + A	See note below
1027	183	R_<CLS>_RELATIVE	Delta(S) + A	See note below
1028	184	R_<CLS>_TLS_IMPDEF1		See note below
1029	185	R_<CLS>_TLS_IMPDEF2		See note below
		R_<CLS>_TLS_DTPREL	DTPREL(S+A)	See note below
		R_<CLS>_TLS_DTPMOD	LDM(S)	See note below
1030	186	R_<CLS>_TLS_TPREL	TPREL(S+A)	
1031	187	R_<CLS>_TLSDESC	TLSDESC(S+A)	Identifies a TLS descriptor to be filled
1032	188	R_<CLS>_IRELATIVE	Indirect(Delta(S) + A)	See note below.

With the exception of R_<CLS>_COPY all dynamic relocations require that the place being relocated is an 8-byte aligned 64-bit data location in ELF64 or a 4-byte aligned 32-bit data location in ELF32.

R_<CLS>_ABS64 and R_<CLS>_ABS32 may only appear in a well-formed executable or dynamic shared object in ELF64 or ELF32 respectively. Note that for their respective file format these relocations are both static and dynamic relocations.

R_<CLS>_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_<CLS>_COPY is normally only used in SysV 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_<CLS>_GLOB_DAT relocates a GOT entry used to hold the address of a (data) symbol which must be resolved at load time.

R_<CLS>_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 after the relocation has been resolved.
- Because the initial value of the place is not related to the ultimate target of a R_<CLS>_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_<CLS>_RELATIVE represents a relative adjustment to the place based on the load address of the object relative to its original link address. All symbols defined in the same segment will have the same relative adjustment. If S is the null symbol (ELF symbol index 0) then the adjustment is based on the segment defining the place. On systems where all segments are mapped contiguously the adjustment will be the same for each reloction, thus adjustment never needs to resolve the symbol. This relocation represents an optimization; it can be used to replace R_<CLS>_GLOB_DAT when the symbol resolves to the current dynamic shared object.

R_<CLS>_IRELATIVE represents a dynamic selection of the place's resolved value. The means by which this relocation is generated is platform specific, as are the conditions that must hold when resolving takes place.

Relocations R_AARCH64_TLS_DTPREL, R_AARCH64_TLS_DTPMOD and R_AARCH64_TLS_TPREL were previously documented as R_AARCH64_TLS_DTPREL64, R_AARCH64_TLS_DTPMOD64 and R_AARCH64_TLS_TPREL64 respectively. The old names can be supported if needed for backwards compatibility.

It is implementation defined whether R_<CLS>_TLS_IMPDEF1 implements R_<CLS>_TLS_DTPREL and R_<CLS>_TLS_IMPDEF2 implements R_<CLS>_TLS_DTPMOD or whether R_<CLS>_TLS_IMPDEF1 implements R_<CLS>_TLS_DTPMOD and R_<CLS>_TLS_IMPDEF2 implements R_<CLS>_TLS_DTPREL; a platform must document its choice[1].

5.7.13 Private and platform-specific relocations

Private relocations for vendor experiments:

- 0xE000 to 0xEFFF for ELF64
- 0xE0 to 0xEF for ELF32

Platform ABI defined relocations:

- 0xF000 to 0xFFFF for ELF64
- 0xF0 to 0xFF for ELF32

Platform ABI relocations can only be interpreted when the EI_OSABI field is set to indicate the Platform ABI governing the definition.

All of the above codes will not be assigned by any future version of this standard.

5.7.14 Unallocated relocations

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

5.7.15 PAuthABI relocations

The PAuthABIELF64 ELF extension, currently in Alpha state, defines several relocations in the vendor experiment space. Arm reserves codes 580 to 600 for static PAuthABIELF64 relocations and 1040 - 1060 for dynamic PAuthABIELF64 relocations. When the extension moves to Release state the relocations defined in PAuthABIELF64 will be added to this document and all unused codes in the reserved ranges will be released.

5.7.16 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 only be idempotent if the original addend was zero and if subsequent re-linking assumes that REL relocations have zero for all addends.

6 Program Loading and Dynamic Linking

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

6.1 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

The below table lists the processor-specific segment types.

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
PT_AARCH64_MEMTAG_MTE	0x70000002	Reserved for MTE memory tag data dumps in core files

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.

`PT_AARCH64_MEMTAG_MTE` segments (if present) hold MTE memory tags for a particular memory range. At present they are defined for core dump files of type ET_CORE. A description of the program header and contents can be found in [MTEEXTENSIONS].

p_flags

There are no AArch64-specific flags.

6.1.1 Platform architecture compatibility data

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

6.2 Program Property

The following processor-specific program property types [LINUX_ABI] are defined:

Program Property Type

Name	Value
GNU_PROPERTY_AARCH64_FEATURE_1_AND	0xc0000000

`GNU_PROPERTY_AARCH64_FEATURE_1_AND` describes a set of processor features with which an ELF object or executable image is compatible, but does not require in order to execute correctly. It has a single 32-bit value for the `pr_data` field. Each bit represents a separate feature.

Static linkers processing ELF relocatable objects must set the feature bit in the output object or image only if all the input objects have the corresponding feature bit set. For each feature bit set in an ELF executable or shared library, a loader may enable the corresponding processor feature for that ELF file.

The following bits are defined for GNU_PROPERTY_AARCH64_FEATURE_1_AND:

GNU_PROPERTY_AARCH64_FEATURE_1_AND Bit Flags

Name	Value
GNU_PROPERTY_AARCH64_FEATURE_1_BTI	1U << 0
GNU_PROPERTY_AARCH64_FEATURE_1_PAC	1U << 1

`GNU_PROPERTY_AARCH64_FEATURE_1_BTI` This indicates that all executable sections are compatible with Branch Target Identification mechanism. An executable or shared object with this bit set is required to generate Custom PLTs with BTI instruction.

`GNU_PROPERTY_AARCH64_FEATURE_1_PAC` This indicates that all executable sections have been protected with Return Address Signing. Its use is optional, meaning that an ELF file where this feature bit is unset can still have Return Address signing enabled in some or all of its executable sections.

6.3 Program Loading

6.3.1 *Process* GNU_PROPERTY_AARCH64_FEATURE_1_BTI

If Branch Target Identification mechanism is enabled on a processor then the Guard Page (GP) bit must be disabled on the memory image of loaded executable segments of executables and shared objects that do not have `GNU_PROPERTY_AARCH64_FEATURE_1_BTI` set, before execution is transferred to them.

6.4 Dynamic Linking

6.4.1 *Custom PLTs*

- To support Branch Target Identification mechanism, in the presence of a `GNU_PROPERTY_AARCH64_FEATURE_1_BTI` all PLT entries generated by the linker must have a BTI instruction as the first instruction. The linker must add the `DT_AARCH64_BTI_PLT` (AArch64 specific dynamic array tags) tag to the dynamic section.
- To support Pointer Authentication, PLT entries generated by the linker can have an authenticating instruction as the final instruction before branching back. The linker must add the `DT_AARCH64_PAC_PLT` (AArch64 specific dynamic array tags) tag to the dynamic section.
- If the linker generates custom PLT entries with both BTI and PAC instructions, it must add both `DT_AARCH64_BTI_PLT` and `DT_AARCH64_PAC_PLT` tags to the dynamic section.

6.4.2 *Dynamic Section*

AArch64 specifies the following processor-specific dynamic array tags.

AArch64 specific dynamic array tags

Name	Value	d_un	Executable	Shared Object
DT_AARCH64_BTI_PLT	0x70000001	d_val	Platform specific	Platform Specific
DT_AARCH64_PAC_PLT	0x70000003	d_val	Platform specific	Platform Specific
DT_AARCH64_VARIANT_PCS	0x70000005	d_val	Platform specific	Platform Specific

`DT_AARCH64_BTI_PLT` indicates PLTs enabled with Branch Target Identification mechanism.

`DT_AARCH64_PAC_PLT` indicates PLTs enabled with Pointer Authentication.

The presence of both `DT_AARCH64_BTI_PLT` and `DT_AARCH64_PAC_PLT` indicates PLTs enabled with both Branch Target Identification mechanism and Pointer Authentication.

`DT_AARCH64_VARIANT_PCS` must be present if there are `R_<CLS>_JUMP_SLOT` relocations that reference symbols marked with the `STO_AARCH64_VARIANT_PCS` flag set in their `st_other` field.

7 Footnotes

---

1 Earlier versions of this specification required that `R_<CLS>_TLS_IMPDEF1` implement `R_<CLS>_TLS_DTPREL` and `R_<CLS>_TLS_IMPDEF2` implement `R_<CLS>_TLS_DTPMOD`; however the Linux platform ABI has always implemented the alternative specification. It is recommended that new platforms follow the Linux platform specification as this is the most widely adopted.