Memtag ABI Extension to ELF for the Arm® 64-bit Architecture (AArch64)

2023Q3

Date of Issue: 6th October 2023

arm

1 Preamble

1.1 Abstract

This document describes the Memory Tagging ABI (MemtagABI) Extensions to ELF for the Arm 64-bit architecture (AArch64). A high-level language may use these extensions to apply memory tags to certain regions of memory (globals, stack) that may not otherwise be possible within the base standard.

1.2 Keywords

ELF, AArch64 ELF, Memory Tagging, Memtag, MTE

1.3 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.

1.4 Licence

1.5 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.

1.6 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.

1.7 Trademark notice

1.8 Copyright

Contents

1 Preamble **2**

1.1 Abstract 2

1.2 Keywords 2

1.3 Latest release and defects report 2

1.4 Licence 3

1.5 About the license 3

1.6 Contributions 3

1.7 Trademark notice 3

1.8 Copyright 3

2 About this document **5**

2.1 Change Control 5

2.1.1 Current Status and Anticipated Changes 5

2.1.2 Change history 5

2.2 References 5

2.3 Terms and Abbreviations 5

3 Scope **7**

4 Platform Standards **8**

5 Introduction **9**

5.1 Design Goals 9

5.2 General Principles 9

5.3 General Restrictions 9

6 Dynamic Section **10**

7 Section Types **10**

8 Tagging Global Variables **11**

8.1 Compilation 11

8.1.1 Identifying tagged global variables 11

8.2 Static Linking 11

8.2.1 Resolving Mismatched Declarations 11

8.3 Encoding of `SHT_AARCH64_MEMTAG_GLOBALS_DYNAMIC` 12

8.4 Relocation Operations 14

8.4.1 Extended semantics of `R_AARCH64_RELATIVE` 14

9 Run-time dynamic linking **16**

9.1 Process initialization using dynamic table entries 16

9.2 Global variable tagging 16

9.3 dlopen/dlsym 17

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.

This document is at **Alpha** release quality.

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
0.1	6th March 2023	Alpha draft release.

2.2 References

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

Ref	URL or other reference	Title
AAELF64	IHI 0056	ELF for the Arm 64-bit Architecture

2.3 Terms and Abbreviations

The MemtagABI extension for the Arm 64-bit Architecture uses the following terms and abbreviations.

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 CPPABI64, AAELF64, ...

Asymmetric (MTE mode)

An Armv8.7 extension to MTE, Asymmetric is a mode of operation that provides synchronous checking on memory reads, and asynchronous checking of memory writes.

Asynchronous (MTE mode)

An MTE mode of operation that updates `TFSR_ELx` (or `TFSR0_EL1` for EL0) when mismatched logical and allocation tags are detected during a load/store operation. This allows imprecise detection that a fault has occurred, at the benefit of increased performance over the synchronous mode.

MTE

ARM's Memory Tagging Extension, an Armv8.5 architectural extension. For more information, see the Whitepaper or MTE documentation.

Synchronous (MTE mode)

An MTE mode of operation that provides a synchronous data abort exception when mismatched logical and allocation tags are detected during a load/store operation. Exceptions in this mode are precise, providing the exact instruction where the fault occurred, and the exact faulting address. The performance of this mode is expected to be less than the Asynchronous mode.

3 Scope

This document is a set of extensions to ELF for the Arm 64-bit architecture (AAELF64) describing how the MemtagABI information is encoded in the ELF file. As an alpha document all details in this document are subject to change.

4 Platform Standards

As is the case with the AAELF64, 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. This document will present recommendations for a SysVr4 like operating system such as Linux.

5 Introduction

The Armv8.5-A architecture introduced the Memory Tagging Extension (MTE) feature, which allows each 16-byte region (a "tag granule") of taggable memory to have a 4-bit piece of metadata (an "allocation tag") associated with it. Additionally, the least significant nibble of the top byte of a pointer may contain a similar piece of metadata (a "logical tag"). During load and store instructions, the logical tag and allocation tag are compared, and upon mismatch, the processor aborts with a Synchronous or Asynchronous MTE exception.

5.1 Design Goals

The goals of the MemtagABIELF64 document are to:

- Provide primitives that can be used to support protection of the heap, stack, and global variables with memory tagging.
- Provide rationale for design choices.
- Describe a scheme for tagging of the heap and global variables that is backwards-compatible. In particular, a binary built with MTE heap or globals protection will function correctly when linked with a non-MemtagABI aware linker, loaded with a non-MemtagABI aware dynamic loader, or run on non-MTE capable hardware.

Note

It is expected that stack instrumentation, because MTE instructions are not part of the no-op space, may crash on non-MTE hardware or non-MemtagABI linkers/loaders.

5.2 General Principles

- In the general case, allocation tags are assigned at random at run-time. Some tags may be "mostly random". For example, for global variable or heap protection, it may be useful to ensure that adjacent memory regions have different tags to reliably detect linear buffer overflow. In addition, stack tagging instrumentation may require some other non-fully-random optimisations, such as using a random base tag per-frame, however intra-frame objects have sequential tags.

5.3 General Restrictions

- Global variables can only be tagged in dynamic executables, as assignment of allocation tags happens during load time, and materialization of logical tags happens during relocation processing.
- Global variables can only be tagged on platforms that use RELA relocation types. For more information, see the changed `R_AARCH64_RELATIVE` relocation semantics description in this document.

6 Dynamic Section

MemtagABI adds the following processor-specific dynamic array tags:

MemtagABI specific dynamic array tags

Name	Value	d_un	Executable	Shared Object
`DT_AARCH64_MEMTAG_MODE`	0x70000009	d_val	Platform specific	Platform Specific
`DT_AARCH64_MEMTAG_HEAP`	0x7000000b	d_val	Platform specific	Platform Specific
`DT_AARCH64_MEMTAG_STACK`	0x7000000c	d_ptr	Platform specific	Platform Specific
`DT_AARCH64_MEMTAG_GLOBALS`	0x7000000d	d_val	Platform specific	Platform Specific
`DT_AARCH64_MEMTAG_GLOBALSSZ`	0x7000000f	d_val	Platform specific	Platform Specific

`DT_AARCH64_MEMTAG_MODE` indicates the initial MTE mode that should be set. It has two possible values: `0`, indicating that the desired MTE mode is Synchronous, and `1`, indicating that the desired mode is Asynchronous. This entry is only valid on the main executable, usage in dynamically loaded objects is ignored.

The presence of the `DT_AARCH64_MEMTAG_HEAP` dynamic array entry indicates that heap allocations should be protected with memory tagging. Implementation of this logic is left to the heap allocator. This entry is only valid on the main executable.

The presence of the `DT_AARCH64_MEMTAG_STACK` dynamic array entry indicates that stack allocations should be protected with memory tagging. The implementation of stack tagging is left to the compiler, this dynamic array entry is a signal to the loader that the stack regions should be mapped using tag-capable memory. This entry is only valid on the main executable.

`DT_AARCH64_MEMTAG_GLOBALS` indicates that global variables should be protected with memory tagging. The value is the unrelocated virtual address which points to the start of the `SHT_AARCH64_MEMTAG_GLOBALS_DYNAMIC` section. Each dynamically loaded object (including the main executable) should have its own `DT_AARCH64_MEMTAG_GLOBALS` dynamic table entry, if the object contains tagged global variables.

`DT_AARCH64_MEMTAG_GLOBALSSZ` is the size of the `SHT_AARCH64_MEMTAG_GLOBALS_DYNAMIC` section, in bytes.

These values are in the AArch64 Processor-specific range. The values are subject to change if there is a clash with any section types added by AAELF64.

7 Section Types

MemtagABI adds the following additional Processor-specific section types:

MemtagABI Section Types

Name	Value
`SHT_AARCH64_MEMTAG_GLOBALS_STATIC`	`0x70000007`
`SHT_AARCH64_MEMTAG_GLOBALS_DYNAMIC`	`0x70000008`

`SHT_AARCH64_MEMTAG_GLOBALS_STATIC` is a section type used during static linking. This section is of size `0`. For each tagged global variable in the object file, there exists an `R_AARCH64_NONE` relocation that references both the symbol to be tagged, and this section. This section's flags should not have `SHF_ALLOC` set, and should be discarded by the linker.

`SHT_AARCH64_MEMTAG_GLOBALS_DYNAMIC` is a section type used during dynamic loading. For each linked object, there should only be a single `SHT_AARCH64_MEMTAG_GLOBALS_DYNAMIC` section, which is pointed to by the `DT_AARCH64_MEMTAG_GLOBALS` dynamic array entry. This section contains a compressed series of "global variable descriptors", metadata encoding the offset and size to which a random allocation tag should be applied. This section is defined in detail later in this document.

The value is in the AArch64 Processor-specific range. The value is subject to change if there is a clash with any section types added by AAELF64.

8 Tagging Global Variables

8.1 Compilation

Global variables that you wish to be tag must meet MemtagABI-specific guidelines. TLS variables are not tagged as part of this specification (although extending this document to add this support is encouraged). In addition, when the compiler defines a global variable as tagged, it must also:

1. If necessary, round up the size such that it's a multiple of the tag granule size (in other words, `size % 16 == 0`), and
2. If necessary, round up the alignment such that it's a multiple of the tag granule size (in other words, `alignment % 16 == 0`).

8.1.1 Identifying tagged global variables

If a comdat group contains one or more definitions of tagged global variables, the compiler should produce an `SHT_AARCH64_MEMTAG_GLOBALS_STATIC` section and a relocation section with `sh_link` pointing to the `SHT_AARCH64_MEMTAG_GLOBALS_STATIC` section within that comdat group.

For each tagged global variable, the compiler should produce an `R_AARCH64_NONE` relocation placed in the aforementioned relocation section. This relocation should have its `ELF64_R_SYM` bits of the `r_info` field point to the global variable that needs to be tagged. This communicates to the static linker that the global variable in question is marked for tagging.

In addition, the compiler and static linker should ensure that all references to tagged global variables are via the GOT (including for internal symbols). The relocations described later in this document ensure that the dynamic loader is able to materialize logical tags for global variables into the GOT, which results in correctly tagged global variable accesses.

8.2 Static Linking

The static linker should parse the `SHT_AARCH64_MEMTAG_GLOBALS_STATIC` section to identify all tagged global variables.

8.2.1 Resolving Mismatched Declarations

Ideally, all translation units linked into a program are all built with or without tagged globals. However, this may not always be possible - a common example is when there is hand-written non-MemtagABI-aware assembly. So, the static linker must take special care when resolving the definition of a global variable if there are mismatched declarations that are not all tagged or untagged. In these instances, the linker must:

1. Conservatively mark the resolved definition as untagged, and
2. Use the largest size and alignment for the resolved definition.

Because tagged globals might have a larger size or alignment (in order to align with the tag granularity), the compiler might take advantage of optimisations that are only possible with the larger size or alignment. These optimisations might result in out-of-bounds memory accesses or alignment-related exceptions if the smaller size/alignment definition is selected. Alternatively, hand-written assembly might use PC-relative references to a global whose definition comes from a C source file that's compiled with global tagging. In this instance, the PC-relative reference would be untagged (versus the GOT-relative reference in correctly compiled code), and will cause tag exceptions at runtime.

8.3 Encoding of `SHT_AARCH64_MEMTAG_GLOBALS_DYNAMIC`

Each tagged global variable should have a random allocation tag assigned by the dynamic loader at load time. In order to accomplish this, the loader needs to know the set of regions that need tagging (exactly which symbol corresponds to each region is not necessary). For each global variable, we emit a compressed "descriptor" to the `SHT_AARCH64_MEMTAG_GLOBALS_DYNAMIC` section.

The descriptor is comprised of two parts:

1. The unrelocated virtual address of the global variable, and
2. The tag-granule-aligned size of the global variable.

The descriptors are compressed as follows:

1. Because the virtual address is always tag-granule-aligned, the descriptor holds the virtual address divided by the tag granule size (in other words, `descriptor.address = variable.address / 16`). Similarly, the size is always granule-aligned, and the descriptor holds the size divided by the tag granule size (in other words, `descriptor.size = variable.size / 16`).
2. Most global variables are laid out contiguously, and so the distance between adjacent tagged global variables is short (often the next global is located at `previous_vaddr + previous_size`). Instead of encoding virtual addresses, we sort the entire list of globals, ascending, by the virtual address, and encode the distance (in granules) between the previous tagged global's end address (`previous_vaddr + previous_size`) and itself. The first global is offset by the number of granules from the zero address. For example, if the first 32-byte tagged global is at `0x100` and the second is at `0x120`, then `descriptor[0] = { .addr = 0x10, .size = 0x2 }` and `descriptor[1] = { .addr = 0x0, .size = 0x2 }`. In order to minimise the size of the descriptor section, an ideal linker will lay out all tagged globals within a section contiguously.
3. We encode the distance between globals, and the size of the global, as ULEB128 integers.
4. Often the ULEB128-encoded distance is less than a byte in size (there are many globals separated by a distance of 0, 1, and 2 tags, and not many with a distance larger than that). In addition, most of the globals are 1, 2, and 3 granules in size, but naively cost a full byte for the ULEB128-encoded value. In order to optimise these common cases, the lower three bits of the distance are reserved to encode the size. If the size of the global variable is less than 8 (2^3) tag granules, then the reserved bits of the distance shall carry the size of the global. Otherwise, the reserved bits shall be set to zero, and the size of the global shall be placed as an additional ULEB128-encoded integer after the distance.
5. A tagged global can't be zero tag granules in size. Thus, if an extra ULEB128 integer is used for the size, the encoded value should be `size - 1`.

Pseudocode to generate the `SHT_AARCH64_MEMTAG_GLOBALS_DYNAMIC` section is presented below:

```
distance_reserved_bits = 3
sort(tagged_symbols) # by symbol.address, ascending

last_symbol_end = 0
for symbol in tagged_symbols:
   distance = ((symbol.address - last_symbol_end) / tag_granule_size)
               << distance_reserved_bits
   size = symbol.size / tag_granule_size
```

```
    last_symbol_end = symbol.address + symbol.size

    if size < (1 << distance_reserved_bits):
       write_to_section(uleb128(distance | size))
    else:
       write_to_section(uleb128(distance))
       write_to_section(uleb128(size - 1))
```

8.4 Relocation Operations

In order to support materialization of the logical tag into the relevant GOT entry for a tagged global, some AArch64 relocations have semantics that extend the AAELF64 definition. Because global variable support in the MemtagABI is designed to be backwards-compatible, the result of the relocation on a non-MemtagABI binary or non-MTE device is the same as the non-extended variant.

The relocations reference the following mnemonics:

- `LDG(pointer)` is an instruction for the run-time environment to use the `ldg` instruction on `pointer` to materialize the correct logical tag for a symbol. This operation should also align the pointer down to the closest tag granule before executing the `ldg` instruction.
- For all the relocation types listed below, the loader should use the `ldg` instruction on the target address before writing into the target field, as the target field may be inside of a tagged region.

Relocations with extended semantics

ELF64 Code	Name	Base Operation	MemtagABI Extended Operation
257	R_AARCH64_ABS64	S + A	LDG(S) + A
1025	R_AARCH64_GLOB_DAT	S + A	LDG(S) + A
1027	R_AARCH64_RELATIVE	Delta(S) + A	LDG(Delta(S) + A + *P) - *P

`R_AARCH64_ABS64` and `R_AARCH64_GLOB_DAT` are thus extended to materialize the allocation tag of the symbol `S` for relocation purposes.

8.4.1 *Extended semantics of* `R_AARCH64_RELATIVE`

`R_AARCH64_RELATIVE` is extended with special care to preserve backwards compatibility. This extension handles the case when the logical tag cannot be naively retrieved from the relocation result, as the result ends up being out-of-bounds of the referenced symbol:

```
// Comments show the resulting dynamic relocations if this file were
// compiled with -fPIC.

// GOT_ENTRY_FOR_foo populated with R_AARCH64_RELATIVE, as usual, with
// addend zero and tag-offset zero. This can be compressed to a RELR
// relocation.
int* foo[32];

// GOT_ENTRY_FOR_foo_middle populated with R_AARCH64_RELATIVE, with addend
// 16 and tag-offset -16. Because the linker knows that the tag-offset is
// in-bounds of the symbol, the linker can omit the tag-offset, and
// compress this to a RELR relocation.
int* foo_middle = &foo[16];

// GOT_ENTRY_FOR_foo_end is the example that needs the new R_AARCH64_RELATIVE
// semantics. foo_end must have the same logical tag as foo, but naively
// loading the allocation tag from the resolved address would end up having
// foo_end using an incorrect logical tag. The logical tag of the relocation
// result should be derived from 'foo', not from 'foo + 32', but the non-tag
// bits of the result should be 'foo + 32'. The static linker must encode the
// tag-offset of -32 in the target field so the tag derivation address can be
// calculated correctly, and this relocation may not be compressed to RELR.
```

```
int* foo_end = &foo[32];
```

Effectively, the unused bits in the target field (at `*P`) are used as an offset to tell the relocation where to derive the tag from. In the non-MemtagABI case, and in cases where the relocation operation is utilised for an untagged symbol, the target field will be zero, and so the tag derivation will come directly from `Delta(S) + A`, maintaining backwards compatibility. However, note the following points:

1. MemtagABI binaries must use RELA relocation encoding, as the unused bits in the target field are used as metadata in the relocation operation.
2. Out-of-bounds relocations to tagged symbols must use the `R_AARCH64_RELATIVE` relocation type when `ABS64` or `GLOB_DAT` relocations are not available. The tag derivation offset must be encoded in the target field. In these instances, because the target field is now no longer zero, some global variables may move from the zero-initialized `bss` section to the constant-initialized `data` section.
3. RELR-style relocation compression can only be applied to `R_AARCH64_RELATIVE` relocations that have a zero tag-derivation offset (that is, the tag derivation can be done entirely by `ldg` of the tag-granule-aligned result of the relocation operation).

An alternative considered was to use an `ABS64` or `GLOB_DAT` relocation instead, to avoid having to materialize the tag-derivation offset. Using these relocations would however require additional dynamic symbol table entries for each symbol used, polluting the dynsym table unnecessarily for each tagged internal symbol.

9 Run-time dynamic linking

9.1 Process initialization using dynamic table entries

During process intialization, the dynamic loader should use the MemtagABI dynamic table entries to enable the correct MTE features at runtime.

If `DT_AARCH64_MEMTAG_MODE` is present, the dynamic loader should (in a platform-specific specific way) enable MTE for the process. On Linux, this can be achieved by `prctl(PR_SET_TAGGED_ADDR_CTRL, PR_TAGGED_ADDR_ENABLE | PR_MTE_TCF_[A]SYNC ...)`. When `DT_AARCH64_MEMTAG_MODE` is `0`, the loader should enable MTE in the synchronous mode. When `DT_AARCH64_MEMTAG_MODE` is `1`, the loader should enable MTE in the asynchronous mode.

If `DT_AARCH64_MEMTAG_HEAP` is present, the dynamic loader should inform the heap allocator (in a platform-specific way) that allocation tags should be used for heap allocations in this process.

If `DT_AARCH64_MEMTAG_STACK` is present, the dynamic loader should enable tagging for the main stack and thread stacks. Re-mapping currently alive stack frames as taggable is dangerous. The easiest way for dynamic loader developers to implement this functionality is by:

1. Ensuring that the implementation completes before any threads have spawned, and
2. Creating a new MTE-protected main stack and calling a function that transitions onto the new stack, preserving all existing frames on the untagged stack.

9.2 Global variable tagging

Because allocation tags can only be applied to tag-capable memory, the dynamic loader should ensure that any segments containing global variables are mapped as tag-capable. On Linux, this is done by ensuring that ensuring that the mappings are created as anonymous mappings (`MAP_ANONYMOUS`) with the `PROT_MTE` flag. Note: this requires special handling for when the linker is invoked implicitly (via the value in `PT_INTERP` rather than as an argument to the loader binary), as the kernel will load the process into memory, rather than the loader. In these cases, the loader is required to unmap the already-loaded segments, and remap them as anonymous tag-capable mappings, re-loading content from the file where necessary.

The `DT_AARCH64_MEMTAG_GLOBALS` dynamic table entry points to a `SHT_AARCH64_MEMTAG_GLOBALS_DYNAMIC` section of compressed global variable descriptors. The size of this section, in bytes, is given by the `DT_AARCH64_MEMTAG_GLOBALSSZ` dynamic table entry. When loading a shared object (including the main executable), prior to processing relocations, the dynamic loader must decompress the descriptors, and for each descriptor, perform the following operation:

1. Generate a random tag (probably using the `irg` instruction). Ideally, to ensure linear global buffer overflows are caught determinstally, the tag should be different from adjacent global variables.
2. Apply the random tag, using the `stg` family of instructions, to the number of granules specified in the descriptor.

The decompression and tagging of the section is described in the pseudocode below:

```
# |load_bias| is the offset that must be added to a object file's virtual
# addresses to get the runtime virtual address. In more technical terms, it's
# the difference between the load address and the p_vaddr value of the
# PT_LOAD segment with the lowest p_vaddr.
virtual_address = load_bias

distance_reserved_bits = 3
i = 0
while i < descriptor_section.size:
   value, bytes_consumed = get_uleb128(descriptor_section.bytes_start + i)
   i += bytes_consumed
   distance = (value >> distance_reserved_bits) * tag_granule_size

   virtual_address += distance

   ngranules = value & ((1 << distance_reserved_bits) - 1)
   if ngranules == 0:
      ngranules, bytes_consumed =
            get_uleb128(descriptor_section.bytes_start + i)
      ngranules += 1
      i += bytes_consumed

   write_random_allocation_tag(virtual_address, ngranules * tag_granule_size)
```

9.3 dlopen/dlsym

On many platforms, programs can load shared libraries at run-time via `dlopen` and access symbols in that library via `dlsym` or `dlvsym`. Tagged globals must be assigned allocation tags at `dlopen`-time, and symbol addresses returned by `dlsym` must have the correct logical tag materialized using the `ldg` instruction.