

# Memory Layout on AArch64 Linux

Author: Catalin Marinas <[catalin.marinas@arm.com](mailto:catalin.marinas@arm.com)>

This document describes the virtual memory layout used by the AArch64 Linux kernel. The architecture allows up to 4 levels of translation tables with a 4KB page size and up to 3 levels with a 64KB page size.

AArch64 Linux uses either 3 levels or 4 levels of translation tables with the 4KB page configuration, allowing 39-bit (512GB) or 48-bit (256TB) virtual addresses, respectively, for both user and kernel. With 64KB pages, only 2 levels of translation tables, allowing 42-bit (4TB) virtual address, are used but the memory layout is the same.

ARMv8.2 adds optional support for Large Virtual Address space. This is only available when running with a 64KB page size and expands the number of descriptors in the first level of translation.

User addresses have bits 63:48 set to 0 while the kernel addresses have the same bits set to 1. TTBRx selection is given by bit 63 of the virtual address. The swapper\_pg\_dir contains only kernel (global) mappings while the user pgd contains only user (non-global) mappings. The swapper\_pg\_dir address is written to TTBR1 and never written to TTBR0.

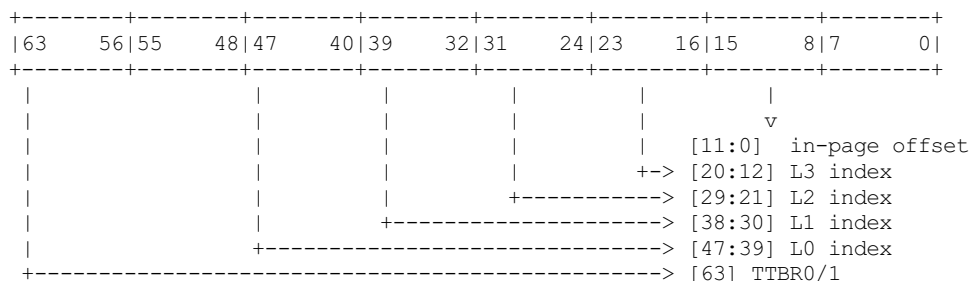
AArch64 Linux memory layout with 4KB pages + 4 levels (48-bit):

Start	End	Size	Use
0000000000000000	0000ffffffffffff	256TB	user
ffff000000000000	ffff7ffffffffffff	128TB	kernel logical memory map
[ffff600000000000	ffff7ffffffffffff]	32TB	[kasan shadow region]
ffff800000000000	ffff800007ffff	128MB	bpf jit region
ffff800008000000	ffff80000ffff	128MB	modules
ffff800010000000	fffffbfffefffff	124TB	vmalloc
fffffbfff0000000	fffffbfffdffff	224MB	fixed mappings (top down)
fffffbfff0000000	fffffbfff07ffff	8MB	[guard region]
fffffbfff0800000	fffffbfff07ffff	16MB	PCI I/O space
fffffbfff0800000	fffffbfff0ffff	8MB	[guard region]
fffffc0000000000	fffffdffffffffff	2TB	vmemmap
fffffe0000000000	fffffdffffffffff	2TB	[guard region]

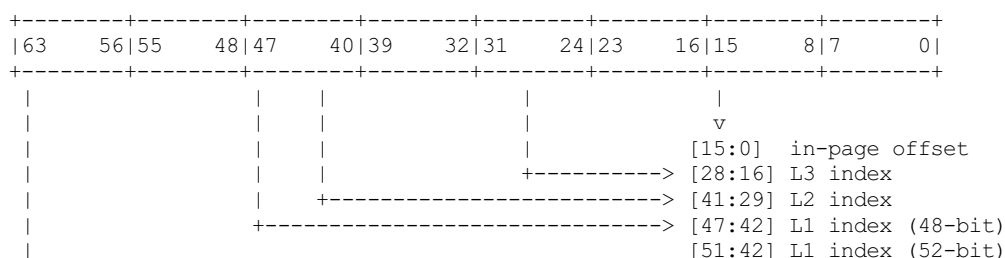
AArch64 Linux memory layout with 64KB pages + 3 levels (52-bit with HW support):

Start	End	Size	Use
0000000000000000	0000ffffffffffff	4PB	user
fff0000000000000	ffff7ffffffffffff	~4PB	kernel logical memory map
[fffd800000000000	ffff7ffffffffffff]	512TB	[kasan shadow region]
ffff800000000000	ffff800007ffff	128MB	bpf jit region
ffff800008000000	ffff80000ffff	128MB	modules
ffff800010000000	fffffbfffefffff	124TB	vmalloc
fffffbfff0000000	fffffbfffdffff	224MB	fixed mappings (top down)
fffffbfff0000000	fffffbfff07ffff	8MB	[guard region]
fffffbfff0800000	fffffbfff07ffff	16MB	PCI I/O space
fffffbfff0800000	fffffbfff0ffff	8MB	[guard region]
fffffc0000000000	fffffdffffffffff	~4TB	vmemmap
fffffe0000000000	fffffdffffffffff	128GB	[guard region]

Translation table lookup with 4KB pages:



Translation table lookup with 64KB pages:



When using KVM without the Virtualization Host Extensions, the hypervisor maps kernel pages in EL2 at a fixed (and potentially random) offset from the linear mapping. See the `kern_hyp_va` macro and `kvm_update_va_mask` function for more details. MMIO devices such as GICv2 gets mapped next to the HYP idmap page, as do vectors when `ARM64_SPECTRE_V3A` is enabled for particular CPUs.

When using KVM with the Virtualization Host Extensions, no additional mappings are created, since the host kernel runs directly in EL2.

## 52-bit VA support in the kernel

If the ARMv8.2-LVA optional feature is present, and we are running with a 64KB page size; then it is possible to use 52-bits of address space for both userspace and kernel addresses. However, any kernel binary that supports 52-bit must also be able to fall back to 48-bit at early boot time if the hardware feature is not present.

This fallback mechanism necessitates the kernel `.text` to be in the higher addresses such that they are invariant to 48/52-bit VAs. Due to the kasan shadow being a fraction of the entire kernel VA space, the end of the kasan shadow must also be in the higher half of the kernel VA space for both 48/52-bit. (Switching from 48-bit to 52-bit, the end of the kasan shadow is invariant and dependent on `~0UL`, whilst the start address will "grow" towards the lower addresses).

In order to optimise `phys_to_virt` and `virt_to_phys`, the `PAGE_OFFSET` is kept constant at `0xFFF0000000000000` (corresponding to 52-bit), this obviates the need for an extra variable read. The `physvirt` offset and `vmemmap` offsets are computed at early boot to enable this logic.

As a single binary will need to support both 48-bit and 52-bit VA spaces, the `VMEMMAP` must be sized large enough for 52-bit VAs and also must be sized large enough to accommodate a fixed `PAGE_OFFSET`.

Most code in the kernel should not need to consider the `VA_BITS`, for code that does need to know the VA size the variables are defined as follows:

`VA_BITS` constant the *maximum* VA space size

`VA_BITS_MIN` constant the *minimum* VA space size

`vabits_actual` variable the *actual* VA space size

Maximum and minimum sizes can be useful to ensure that buffers are sized large enough or that addresses are positioned close enough for the "worst" case.

## 52-bit userspace VAs

To maintain compatibility with software that relies on the ARMv8.0 VA space maximum size of 48-bits, the kernel will, by default, return virtual addresses to userspace from a 48-bit range.

Software can "opt-in" to receiving VAs from a 52-bit space by specifying an `mmap` hint parameter that is larger than 48-bit.

For example:

```
maybe_high_address = mmap(~0UL, size, prot, flags,...);
```

It is also possible to build a debug kernel that returns addresses from a 52-bit space by enabling the following kernel config options:

```
CONFIG_EXPERT=y && CONFIG_ARM64_FORCE_52BIT=y
```

Note that this option is only intended for debugging applications and should not be used in production.