AArch64 Virtualization

Version 1.0

Revision Information

The following revisions have been made to this User Guide.

Date	Issue	Confidentiality	Change
03 March 2017	0100	Non-Confidential	First release

Proprietary Notice

Confidentiality Status

This document is Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Unrestricted Access is an ARM internal classification.

Product Status

The information in this document is final, that is for a developed product.

Web Address

http://www.arm.com

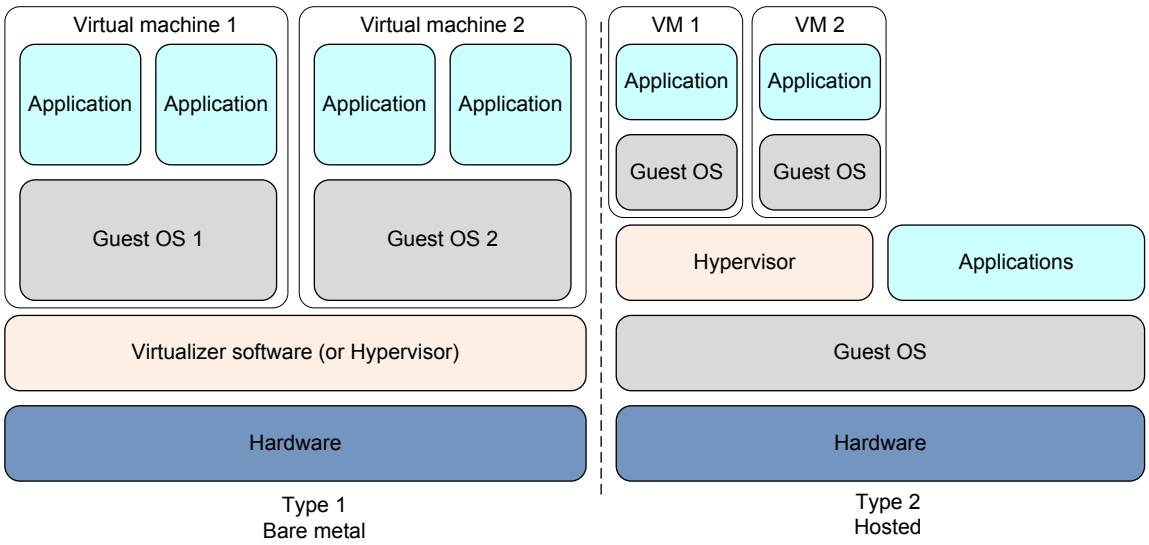
Contents

1 AArch64 virtualization

Most mainstream operating systems are built on the assumption that a system has a single privileged OS running several unprivileged applications. ARM virtualization however, enables more than one OS to co-exist and operate on the same system. Implementing these virtual cores requires both dedicated hardware extensions (to accelerate switching between virtual machines) and hypervisor software.

A hypervisor is a program that allows multiple operating systems to share a single hardware processor. Virtualization hypervisors can be broadly classified as bare metal or hosted according to their design. Each has specific use cases. Regardless of the classification, the functional role of a hypervisor remains the same, namely, arbitration of platform resources, and seamless operation of individual guest operating systems with minimal porting effort and runtime sacrifices.

In the following figure, for Type 1, the bare-metal hypervisor, each Virtual Machine (VM) contains a guest OS. In Type 2, the hosted hypervisor is an extension of the host OS with each subsequent guest OS contained in a separate VM. There are two major Open Source hypervisors, KVM and Xen. Using this scheme, Xen is a Type1 hypervisor and KVM is Type2.

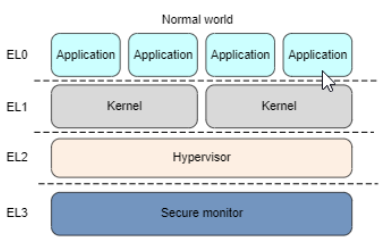


Bare-metal virtualization means that the Type 1 hypervisor has direct access to hardware resources, which results in better performance.

A hosted (Type 2) hypervisor requires an OS to be installed first. The host OS boots first. The hypervisor then behaves like an application that is installed on the OS. This approach provides better hardware compatibility, because the OS is responsible for the hardware drivers instead of the hypervisor. On the other hand, a hosted virtualization hypervisor does not have direct access to hardware and must go through the OS, which can degrade VM performance. Because there can be many services and applications running on the host OS, the hypervisor can often steal resources from the VMs running on it.

Using multiple operating systems on the same system then becomes possible, while providing each OS with an illusion of sole ownership of the system through several architectural features:

- A dedicated Exception level (EL2) for hypervisor code.
- Support for trapping exceptions that change the core context or state.



- Support for routing exceptions and virtual interrupts.
- Two-stage memory translation, where the second stage is for the hypervisor to isolate the guest operating systems.
- A dedicated exception for Hypervisor Call (HVC).

The ARMv8-A architecture permits virtualization using either AArch32 or AArch64 execution states. The hypervisor at EL2 can run in either AArch32 or AArch64 execution state. In AArch32, the execution is similar to the ARMv7-A architecture.

Note

Virtualization support is provided in the Non-secure state only. There is no virtualization support in the Secure state, because TrustZone technology is intended to allow a Secure or trusted environment. This implies a small code base, to enable validation and certification.

When hypervisor code in EL2 is executing in AArch64, there are dedicated registers available, including:

- Exception return state registers: SPSR_EL2 and ELR_EL2.
- Stack pointer: SP_EL2 (and SP_EL0).

2 Hypervisor software

The functions that a hypervisor performs do not depend on the type of hypervisor deployed. They include:

- Memory management.
- Device emulation.
- Device assignment.
- Exception handling.
- Instruction trapping.
- Managing virtual exceptions.
- Interrupt controller management.
- Scheduling.
- Context switching.
- Memory translation.
- Managing multiple virtual address spaces.

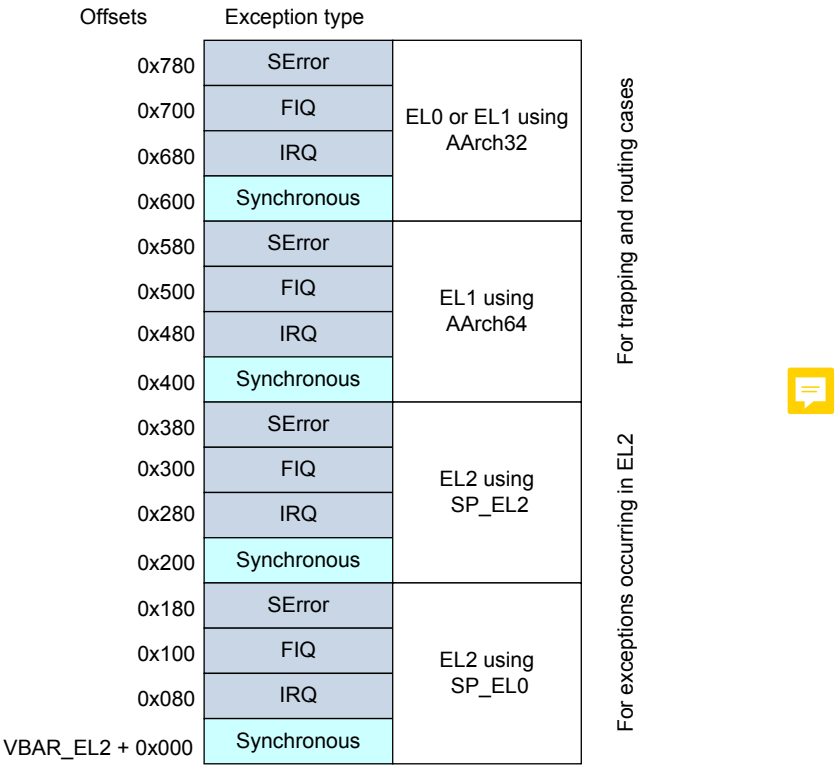
2.1 1.1 Memory management

An ordinary OS manages the memory that the applications run in, and the memory where the OS is located. The hypervisor is responsible for memory management for itself and for the guest operating systems it manages. The entire physical memory is at the direct disposal of the hypervisor. The MMU in EL2 is used by the hypervisor to translate the virtual addresses that the hypervisor uses to address physical memory. EL2 therefore has its own vector table.

The location of the vector table in memory is set using the VBAR_EL2 register.

The following code example shows this:

```
//
// Install vector tables
//
ADR     X0, vector_table
MSR     VBAR_EL2, X0
```



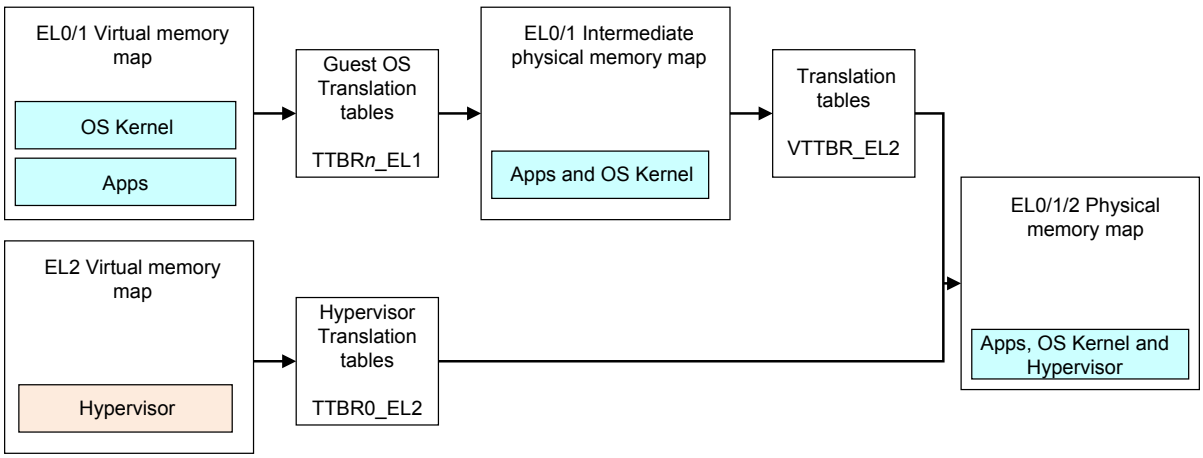
Apart from setting up and managing its own translation tables, a hypervisor must create and manage Stage 2 translation tables for each of its guests.

Example code is shown below:

```
//
// Init VM1
//
.global Image$$VM1_TT_S2$$ZI$$Base
MOV      X0, XZR                              // Reset address: 0x0
MOV      X1, #0                               // Execution state: AArch32
MOV      X2, #0xC0000000                      // Physical memory for VM
LDR      X3, =vm1                             // Data structure of VM context
LDR      X4, =Image$$VM1_TT_S2$$ZI$$Base      // Load address of second stage
                                              // first level table
LDR      X5, =0x80010203                      // Affinity: 0.1.2.3
LDR      X6, =0x411FD073                      // MIDR: Cortex-A57
BL       init_vm


//
// Init VM0
//
.global Image$$VM0_TT_S2$$ZI$$Base
MOV      X0, XZR                              // Reset address: 0x0
MOV      X1, #1                               // Execution state: AArch64
MOV      X2, #0x80000000                      // Physical memory for VM
LDR      X3, =vm0                             // Data structure of VM context
```

```
LDR      X4, =Image$$VM0_TT_S2$$ZI$$Base       // Load address of second stage
first level table
LDR      X5, =0x80000000                       // Affinity: 0.0.0.0
LDR      X6, =0x410FD034                       // MIDR: Cortex-A53
BL       init_vm

. . .

// HCR_EL1
MOV      X8, #1                                // VM==1: Second stage
                                               // translation enable
ORR      X8, X8, #(0x7 << 3)                   // Set FMO/IMO/AMO => physical
                                               // async exceptions routed to
                                               // EL2
ORR      X8, X8, X1, LSL #31                   // RW: based on passed in args
STR      X8, [X3]
MSR      HCR_EL2, X8
. . .
```

Stage 2 translation tables set up by the hypervisor translate intermediate physical memory addresses to physical memory addresses. Any aborts resulting from attempting to translate addresses in Stage 2 are taken at EL2.



The hypervisor is responsible for receiving the aborts and handling them appropriately. For intended and legitimate faults, the hypervisor might take remedial measures such as emulating a device or allocating more memory to a guest operating system. For unexpected faults, the hypervisor can choose to terminate the guest operating system or report aborts to the guest in turn.

2.2 Device emulation

Platform devices are memory-mapped, and guest accesses to devices are subject to at least Stage 2 translation when virtualization is in effect, in this case there are use cases for a hypervisor to emulate a device in software and provide a replacement driver to access either the physical device or a virtual software device. Emulated devices for virtual machines are software implementations of hardware. They exist entirely in software.

Device emulation is necessary where more than one guest is aware of a platform device, using the physical address, and makes attempts to access it. Because of the shared nature, guests cannot be permitted direct accesses to it without arbitration. In such cases, the hypervisor can control access to the device region in question to all guests, with their respective stage 2 translation table descriptors.

Guests can detect a platform device by reading its ID register, or by interrogating registers mentioned in the device tree. By employing the same trapping mechanism as before, the hypervisor can return dummy values for guest reads and ignore writes, effectively giving the guest the impression that the device does not exist on the platform.

When the model has some data to deliver, the hypervisor raises a virtual IRQ (vIRQ). The guest OS responds by attempting to read hardware registers. The hypervisor traps these accesses and provides simulated responses.

In some existing hypervisor solutions, such as KVM or Xen, giving guests access to platform devices is uncommon. Both of these hypervisors provide a virtual platform and emulate all devices, exceptions are only made for network or storage controllers, which might not be platform devices.

2.3 Device assignment

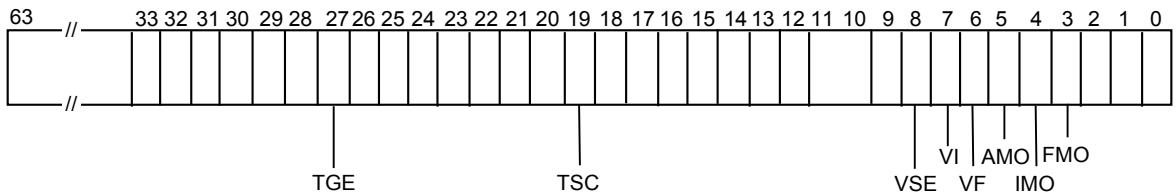
Device emulation is necessary but also expensive, as all accesses to the device by the guest have to be trapped and emulated in software. The hypervisor has the option of assigning individual devices to individual guest operating systems so that the guest can own and operate the device without requiring hypervisor arbitration.

The challenge is to hide from the guest the fact that the device is at a different physical address, and generate a different interrupt ID than that which the guest is expecting. Alternatively, the hypervisor might choose to hide a device from a chosen set of guests either because the device is not present or has already been assigned to a different guest.

Transparent Stage 2 mappings, and interrupt virtualization can circumvent these challenges.

2.4 Exception handling

Exceptions can be trapped, or routed to EL2, to be handled by the hypervisor. This behavior can be selected through control bits in the Hypervisor Control Register (HCR_EL2).



The bit assignments are as follows:

Bit	Name	Function
[3]	FMO	Controls whether the FIQ asynchronous exception is routed to the hypervisor.
[4]	IMO	Controls whether the IRQ asynchronous exception is routed to the

		hypervisor.
[5]	AMO	Controls whether the SError asynchronous exception is routed to the hypervisor.
[6]	VF	Virtual FIQ error. The virtual FIQ is only enabled when the HCR_EL2.FMO bit is set.
[7]	VI	Virtual IRQ error. The virtual IRQ is only enabled when the HCR_EL2.IMO bit is set.
[8]	VSE	Virtual SError/ Asynchronous abort.
[19]	TSC	Controls whether Non-secure EL1 execution of SMC instructions are trapped.
[27]	TGE	Enables or disables trapping for synchronous exceptions.

The AArch64 exception vector table contains four blocks. Which of these four blocks is used depends on whether the core was executing at a lower Exception level (EL0 or EL1) or executing in EL2.

If the core was executing at a lower Exception level, the execution state of EL1 determines which block it uses. If the core was executing in EL2, the currently selected stack pointer (PSTATE.SPSel) determines the block.

2.5 Instruction trapping

In addition to trapping exceptions, the hypervisor can also be configured to trap certain instructions. This is normally because the instructions carry addresses or system address changes which must be translated by the Hypervisor tables. This is also configured through HCR_EL2.

When an instruction has been trapped, hypervisor code can read the *Exception Syndrome Register* (ESR_EL2) to obtain the necessary information about the trapped instruction.

The following instructions can be trapped:

- Accesses to virtual memory control registers, for example TTBR*n* and TTBCR.
- System instructions, for example cache and TLB maintenance instructions.
- Accesses to Auxiliary Control Register (ACTLR_EL1).
- Reads to ID registers.
- WFE and WFI instructions. For example, these could be used to enable the hypervisor to change which guest OS is running when the current guest OS attempts to enter a low-power state.

2.6 Virtual exceptions

ARMv8-A provides support for three virtual exceptions, Virtual SError, Virtual IRQ, and Virtual FIQ. Virtual exceptions are events such as interrupts and aborts that are manufactured by the hypervisor, either in response to an actual physical exception, or manufactured (with no corresponding physical exception) as a result of Device emulation. These are taken in the same

way as the corresponding physical exception. Physical exceptions can be configured to be consumed by the hypervisor. Only virtual exceptions are delivered to guests.

Virtual exceptions can be signaled only when the corresponding physical exception is routed to the hypervisor running at EL2:

These are registered or pended by writing to HCR_EL2 and then taken by the guest OS if not masked.

This means that the hypervisor has control of masking and generating a virtual exception. When a real physical exception occurs, and is routed to the hypervisor software, the hypervisor might run some code, and then signals a virtual exception to the current guest OS. The guest OS handles the exception as it would do for an equivalent physical exception and is not aware that the hypervisor has been involved. The virtual exception is signaled through the virtual CPU interface, or by using dedicated HCR bits. When exceptions are routed to the hypervisor in this fashion, the PSTATE A, I, and F bits no longer mask physical exceptions, instead they mask the handling of virtual exceptions within the guest OS.

Virtual exceptions are always masked when executing in EL2 and EL3. When enabled and pending, the virtual exception is taken when the core returns to Exception level EL1.

In Non-secure EL0 and EL1, each virtual exception is masked by the corresponding PSTATE mask bits. Code running in EL1 or above can write to PSTATE.{A,I,F} bits for current Exception level. If, however, physical exceptions are configured to be routed to EL3, then the hypervisor has no access and cannot set virtual exceptions in response.

2.7 Context switch

When the hypervisor schedules another guest on a core, it must perform a context switch, that is, save the context of the currently running guest to memory, and then restore the context of the new guest from memory. The goal is to recreate the environment for the new guest on the current core before it resumes, creating the illusion of uninterrupted execution. By performing a context switch, the hypervisor ensures that the execution environment follows the guest, and offers the illusion of a virtual core that the guest always occupies.

The following elements of a guest context must be saved and restored:

- The general-purpose registers of the core including the banked registers of all modes.
- System register contents for such things as memory management and access control.
- The pending and active states of private interrupts on the core.
- In the case of guests using core private timers, the timer registers must be saved and restored so that they generate interrupts at the expected intervals.

The physical memory that is assigned to a guest, that the guest sees as RAM, stays in place and does not have to be saved or restored. By using two stages of memory translation, the physical memory that the guest uses stays private and distinct from any others, even for identical guests.

2.8 Memory translation

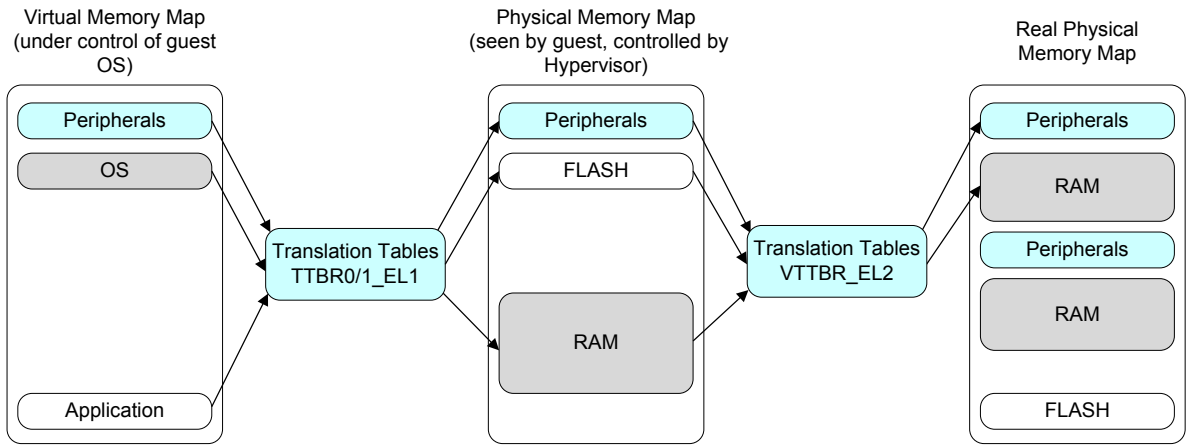
A translation regime is a broad term encompassing the privileges and Exception level of the core and the set of translation tables used. Several memory translation regimes are possible in a virtualized system in the Normal world.

A translation in this case is a memory access into a memory-based table. A stage of translation comprises a set of translation tables that converts an input address to an output address. The input, or virtual addresses are those used by you, and the compiler and linker, when placing code in memory. The output, or physical addresses are those used by the actual system hardware.

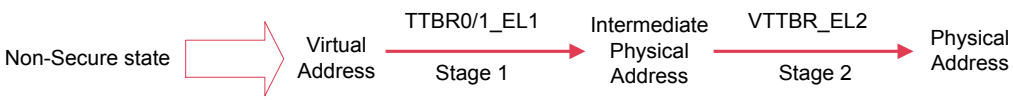
These translations are carried out using the MMUs and the translation table structures that are created by the software control the translation. The input and output addresses take different names depending on the stage of translation.

The hypervisor controls two types of translation. When the core is executing at EL1 or EL0, its translation regime is set up and controlled by the OS. In a system without virtualization, this regime is used to translate virtual addresses to physical addresses. In a virtualized system, however, this physical address is treated as an Intermediate Physical Address (IPA), because it is then subjected to another stage, Stage 2, of translation.

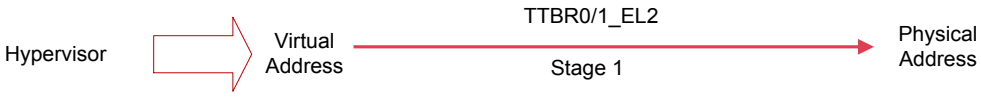
In the presence of an oncoming Stage 2, this first translation regime qualifies as Stage 1. The IPAs cannot be used to address system memory. Although called an intermediate physical address, from the perspective of a guest, the output of this translation regime is what it sees and uses as a physical address.



The second stage of translation uses the Virtualization Translation Table Base Registers VTTBR_EL2 and VTCR_EL2, controlled by the hypervisor.



For the virtual address space of the hypervisor, a single stage translation is used, controlled by the registers TTBR0_EL2 and TCR_EL2.



ARMv8-A virtualization also introduces the concept of a Virtual Machine ID (VMID). Each virtual machine is assigned a VMID, which is an 8-bit value stored inVTTBR_EL2.

These are used to tag a translation as belonging to a particular virtual machine. For guest accesses, the Translation Lookaside Buffers (TLB) within the processor MMU can store a complete VA [illegible] The VMID ensures that only the correct virtual machine can hit on a TLB entry and this can remove the need to invalidate TLBs when a context switch between guest operating systems is performed.

The architecture does not define what value VTTBR_EL2.VMID field is reset to in hardware. This means that the boot code software that runs on each active core must initialize the VTTBR_EL2.VMID field to a known value, for example 0, even if the second stage of translation is not being used.