Project ACRN Documentation

Release v 0.1-rc4

Project ACRN

May 09, 2018

CONTENTS

Welcome to the Project ACRN (version latest) documentation.

For information about the changes and additions for releases, please consult the published *Release Notes* documentation.

Source code for Project ACRN is maintained in the Project ACRN GitHub repo, and is provided under the BSD 3-clause license.

CHAPTER

ONE

SECTIONS

1.1 Introduction to Project ACRN

The open source project ACRN defines a device hypervisor reference stack and an architecture for running multiple software subsystems, managed securely, on a consolidated system by means of a virtual machine manager. It also defines a reference framework implementation for virtual device emulation, called the "ACRN Device Model".

The ACRN Hypervisor is a Type 1 reference hypervisor stack, running directly on the bare-metal hardware, and is suitable for a variety of IoT and embedded device solutions. The ACRN hypervisor addresses the gap that currently exists between datacenter hypervisors, and hard partitioning hypervisors. The ACRN hypervisor architecture partitions the system into different functional domains, with carefully selected guest OS sharing optimizations for IoT and embedded devices.

1.1.1 Automotive Use Case Example

An interesting use case example for the ACRN Hypervisor is in an automotive scenario. The ACRN hypervisor can be used for building a Software Defined Cockpit (SDC) or an In-Vehicle Experience (IVE) solution. As a reference implementation, ACRN provides the basis for embedded hypervisor vendors to build solutions with a reference I/O mediation solution.

In this scenario, an automotive SDC system consists of the Instrument Cluster (IC) system, the In-Vehicle Infotainment (IVI) system, and one or more Rear Seat Entertainment (RSE) systems. Each system is running as an isolated Virtual Machine (VM) for overall system safety considerations.

An **Instrument Cluster (IC)** system is used to show the driver operational information about the vehicle, such as:

- the speed, the fuel level, trip mile and other driving information of the car;
- projecting heads-up images on the windshield, with alerts for low fuel or tire pressure;
- showing rear-view camera, and surround-view for parking assistance.

An **In-Vehicle Infotainment (IVI)** system's capabilities can include:

- navigation systems, radios, and other entertainment systems;
- connection to mobile devices for phone calls, music, and applications via voice recognition;
- control interaction by gesture recognition or touch.

A **Rear Seat Entertainment (RSE)** system could run:

- entertainment system;
- virtual office;
- connection to the front-seat IVI system and mobile devices (cloud connectivity).

- connection to mobile devices for phone calls, music, and applications via voice recognition;
- control interaction by gesture recognition or touch

The ACRN hypervisor can support both Linux* VM and Android* VM as a User OS, with the User OS managed by the ACRN hypervisor. Developers and OEMs can use this reference stack to run their own VMs, together with IC, IVI, and RSE VMs. The Service OS runs as VM0 (also known as Dom0 in other hypervisors) and the User OS runs as VM1, (also known as DomU).

Figure 1.1 shows an example block diagram of using the ACRN hypervisor.

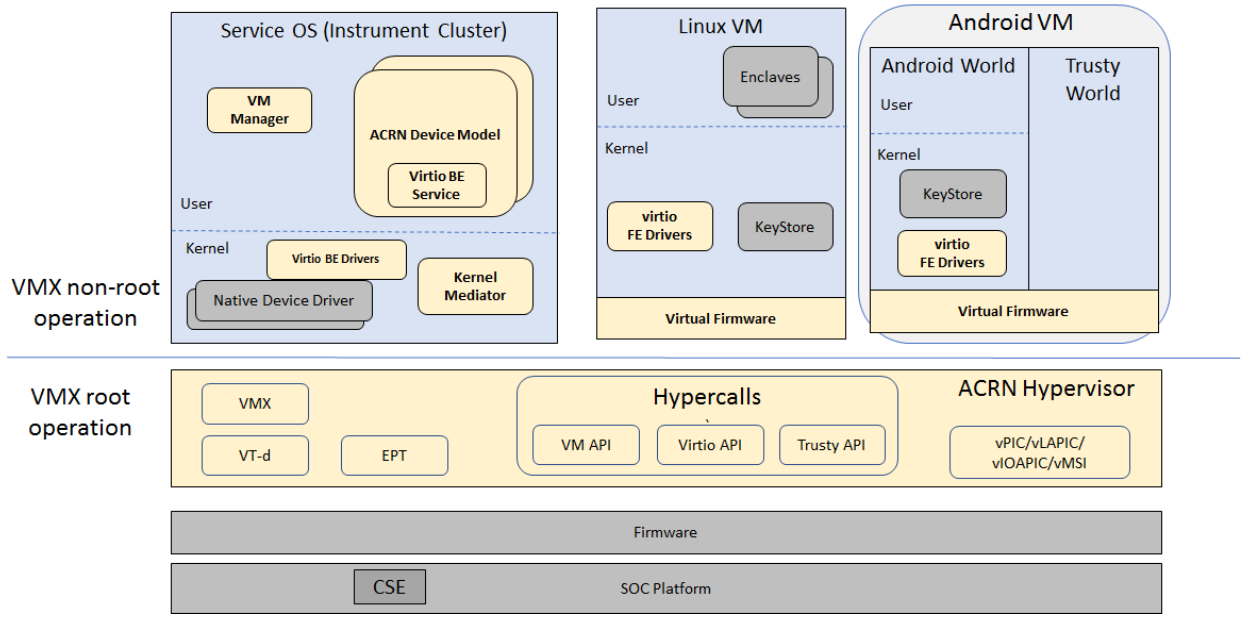


Figure 1.1: Service OS and User OS on top of ACRN hypervisor

This ACRN hypervisor block diagram shows:

- The ACRN hypervisor sits right on top of the bootloader for fast booting capabilities.
- Partitioning of resources to ensure safety-critical and non-safety critical domains are able to coexist on one platform.
- Rich I/O mediators allows various I/O devices shared across VMs, and thus delivers a comprehensive user experience
- Multiple operating systems are supported by one SoC through efficient virtualization.

Note: The yellow color parts in Figure 1.1 are part of the project ACRN software stack. This is a reference architecture diagram and not all features mentioned are fully functional. Other blocks will come from other (open source) projects and are listed here for reference only.

For example: the Service OS and Linux Guest can come from the Clear Linux project at https://clearlinux.org and (in later updates) the Android as a Guest support can come from https://01.org/android-ia.

For the current ACRN-supported feature list, please see *Release Notes*.

1.1.2 Licensing

Both the ACRN hypervisor and ACRN Device model software are provided under the permissive BSD-3-Clause license, which allows *"redistribution and use in source and binary forms, with or without modification"* together with the intact copyright notice and disclaimers noted in the license.

1.1.3 ACRN Device Model, Service OS, and User OS

To keep the hypervisor code base as small and efficient as possible, the bulk of the device model implementation resides in the Service OS to provide sharing and other capabilities. The details of which devices are shared and the mechanism used for their sharing is described in *pass-through* section below.

The Service OS runs with the system's highest virtual machine priority to ensure required device time-sensitive requirements and system quality of service (QoS). Service OS tasks run with mixed priority. Upon a callback servicing a particular User OS request, the corresponding software (or mediator) in the Service OS inherits the User OS priority. There may also be additional low-priority background tasks within the Service OS.

In the automotive example we described above, the User OS is the central hub of vehicle control and in-vehicle entertainment. It provides support for radio and entertainment options, control of the vehicle climate control, and vehicle navigation displays. It also provides connectivity options for using USB, Bluetooth, and WiFi for third-party device interaction with the vehicle, such as Android Auto* or Apple CarPlay*, and many other features.

1.1.4 Boot Sequence

In Figure 1.2 we show a verified Boot Sequence with UEFI on an Intel® Architecture platform NUC (see *Supported Hardware*).

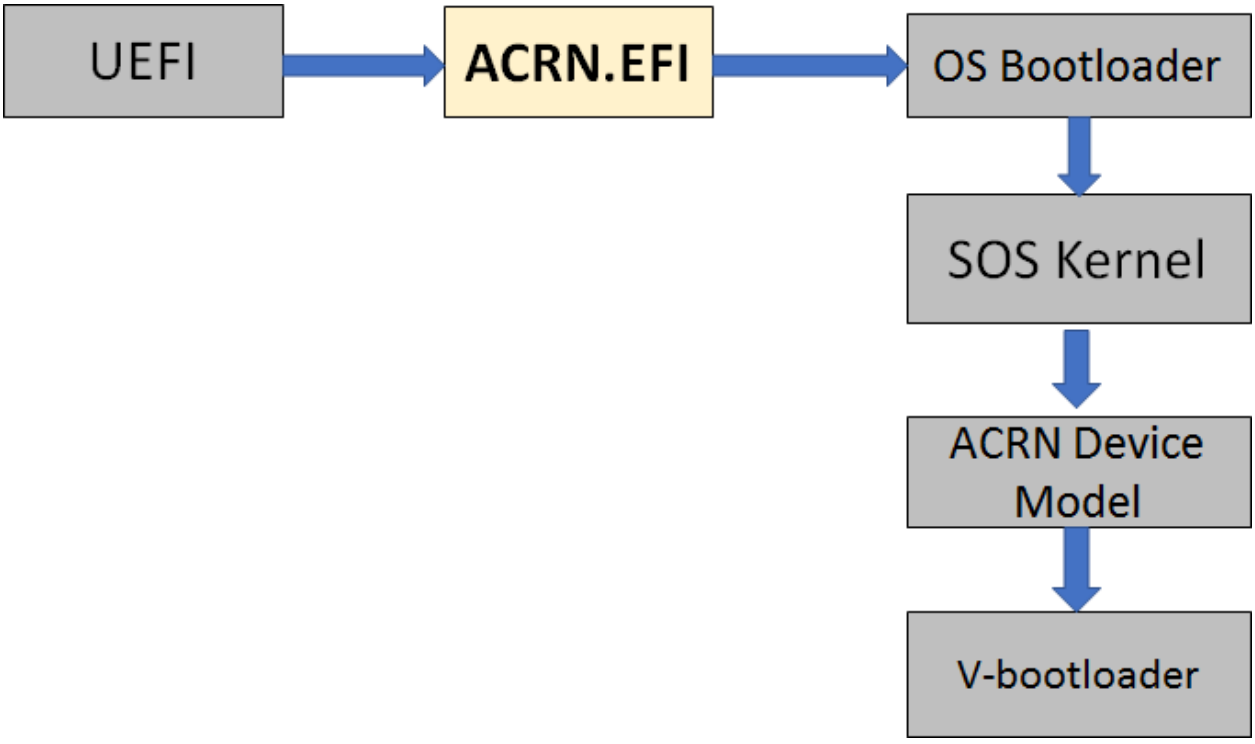


Figure 1.2: ACRN Hypervisor Boot Flow

The Boot process proceeds as follows:

1. UEFI verifies and boots the ACRN hypervisor and Service OS Bootloader
2. UEFI (or Service OS Bootloader) verifies and boots Service OS kernel
3. Service OS kernel verifies and loads ACRN Device Model and Virtual bootloader through dm-verity
4. Virtual bootloader starts the User-side verified boot process

1.1.5 ACRN Hypervisor Architecture

ACRN hypervisor is a Type 1 hypervisor, running directly on bare-metal hardware. It implements a hybrid VMM architecture, using a privileged service VM, running the Service OS that manages the I/O devices and provides I/O mediation. Multiple User VMs are supported, with each of them running Linux* or Android* OS as the User OS .

Running systems in separate VMs provides isolation between other VMs and their applications, reducing potential attack surfaces and minimizing safety interference. However, running the systems in separate VMs may introduce additional latency for applications.

Figure 1.3 shows the ACRN hypervisor architecture, with the automotive example IC VM and service VM together. The Service OS (SOS) owns most of the devices including the platform devices, and provides I/O mediation. Some of the PCIe devices may be passed through to the User OSes via the VM configuration. The SOS runs the IC applications and hypervisor-specific applications together, such as the ACRN device model, and ACRN VM manager.

ACRN hypervisor also runs the ACRN VM manager to collect running information of the User OS, and controls the User VM such as starting, stopping, and pausing a VM, pausing or resuming a virtual CPU.

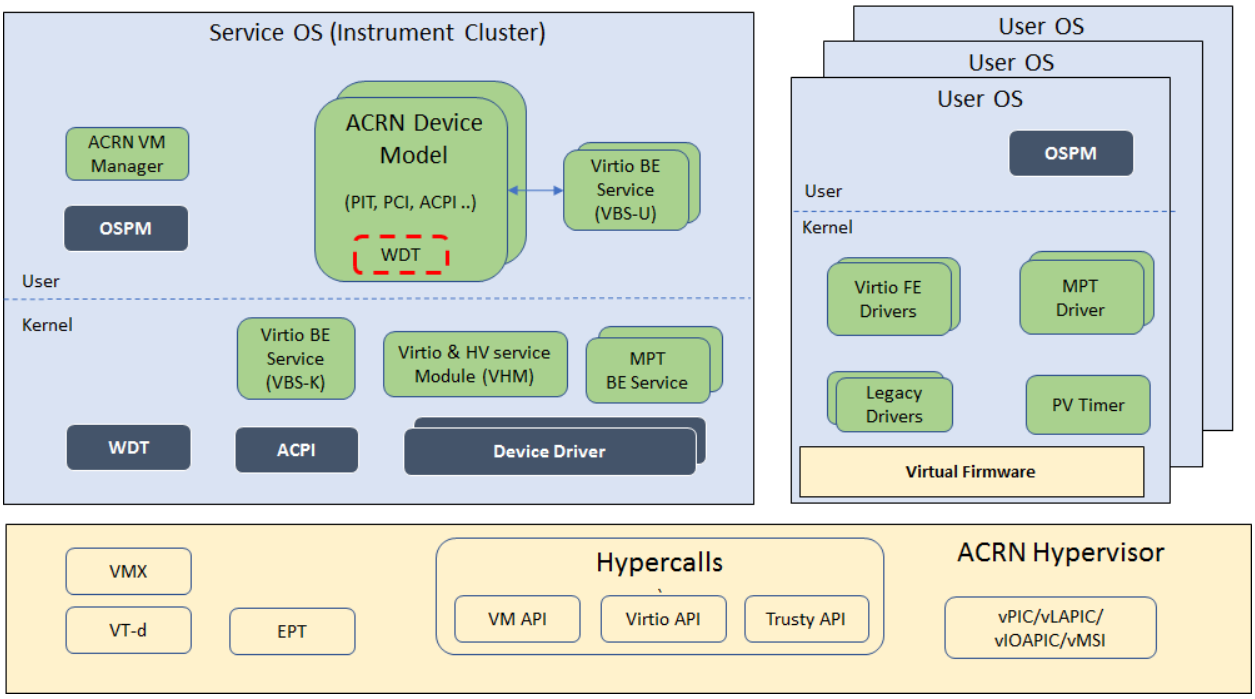


Figure 1.3: ACRN Hypervisor Architecture

ACRN hypervisor takes advantage of Intel Virtualization Technology (Intel VT), and ACRN hypervisor runs in Virtual Machine Extension (VMX) root operation, or host mode, or VMM mode. All the guests, including UOS and SOS, run in VMX non-root operation, or guest mode. (Hereafter, we use the terms VMM mode and Guest mode for simplicity).

The VMM mode has 4 protection rings, but runs the ACRN hypervisor in ring 0 privilege only, leaving rings 1-3 unused. The guest (including SOS & UOS), running in Guest mode, also has its own four protection rings (ring 0 to

3). The User kernel runs in ring 0 of guest mode, and user land applications run in ring 3 of User mode (ring 1 & 2 are usually not used by commercial OSes).

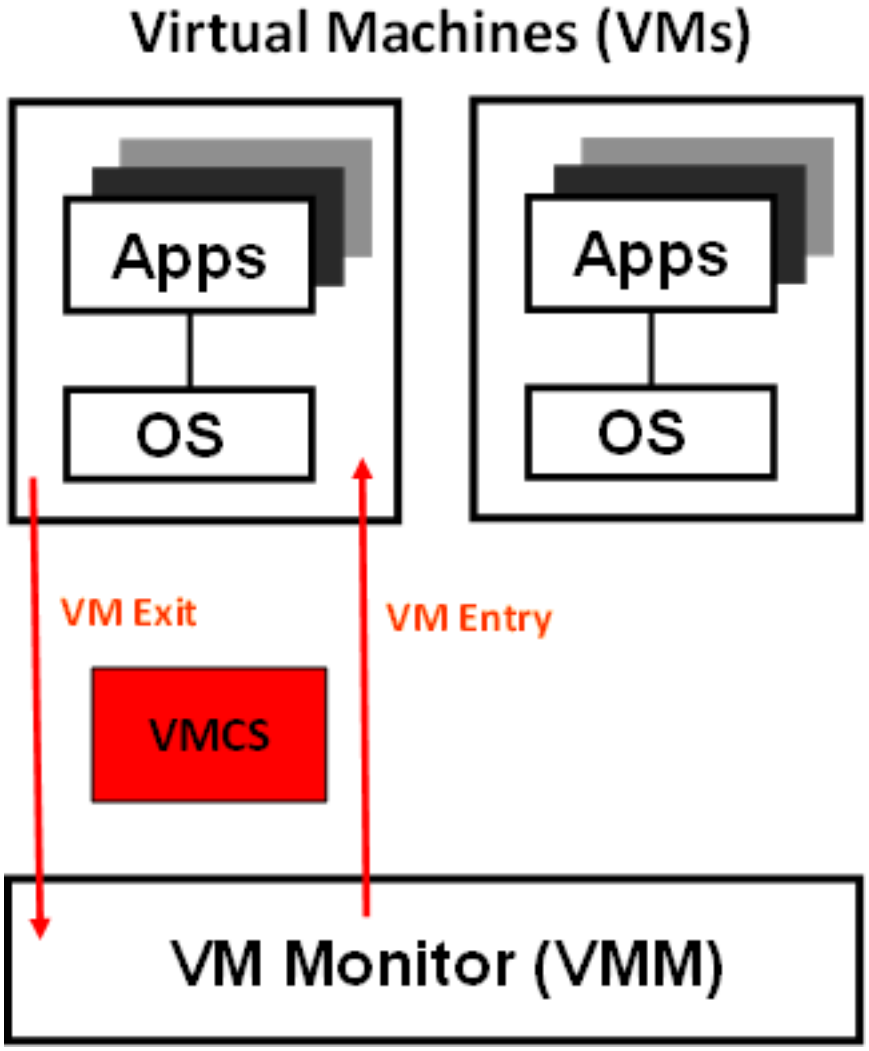


Figure 1.4: VMX Brief

As shown in Figure 1.4, VMM mode and guest mode are switched through VM Exit and VM Entry. When the bootloader hands off control to the ACRN hypervisor, the processor hasn't enabled VMX operation yet. The ACRN hypervisor needs to enable VMX operation thru a VMXON instruction first. Initially, the processor stays in VMM mode when the VMX operation is enabled. It enters guest mode thru a VM resume instruction (or first time VM launch), and returns back to VMM mode thru a VM exit event. VM exit occurs in response to certain instructions and events.

The behavior of processor execution in guest mode is controlled by a virtual machine control structure (VMCS). VMCS contains the guest state (loaded at VM Entry, and saved at VM Exit), the host state, (loaded at the time of VM exit), and the guest execution controls. ACRN hypervisor creates a VMCS data structure for each virtual CPU, and uses the VMCS to configure the behavior of the processor running in guest mode.

When the execution of the guest hits a sensitive instruction, a VM exit event may happen as defined in the VMCS configuration. Control goes back to the ACRN hypervisor when the VM exit happens. The ACRN hypervisor emulates the guest instruction (if the exit was due to privilege issue) and resumes the guest to its next instruction, or fixes the VM

exit reason (for example if a guest memory page is not mapped yet) and resume the guest to re-execute the instruction.

Note that the address space used in VMM mode is different from that in guest mode. The guest mode and VMM mode use different memory mapping tables, and therefore the ACRN hypervisor is protected from guest access. The ACRN hypervisor uses EPT to map the guest address, using the guest page table to map from guest linear address to guest physical address, and using the EPT table to map from guest physical address to machine physical address or host physical address (HPA).

1.1.6 ACRN Device Model Architecture

Because devices may need to be shared between VMs, device emulation is used to give VM applications (and OSes) access to these shared devices. Traditionally there are three architectural approaches to device emulation:

- The first architecture is **device emulation within the hypervisor** which is a common method implemented within the VMware* workstation product (an operating system-based hypervisor). In this method, the hypervisor includes emulations of common devices that the various guest operating systems can share, including virtual disks, virtual network adapters, and other necessary platform elements.
- The second architecture is called **user space device emulation**. As the name implies, rather than the device emulation being embedded within the hypervisor, it is instead implemented in a separate user space application. QEMU, for example, provides this kind of device emulation also used by a large number of independent hypervisors. This model is advantageous, because the device emulation is independent of the hypervisor and can therefore be shared for other hypervisors. It also permits arbitrary device emulation without having to burden the hypervisor (which operates in a privileged state) with this functionality.
- The third variation on hypervisor-based device emulation is **paravirtualized (PV) drivers**. In this model introduced by the XEN project the hypervisor includes the physical drivers, and each guest operating system includes a hypervisor-aware driver that works in concert with the hypervisor drivers.

In the device emulation models discussed above, there's a price to pay for sharing devices. Whether device emulation is performed in the hypervisor, or in user space within an independent VM, overhead exists. This overhead is worthwhile as long as the devices need to be shared by multiple guest operating systems. If sharing is not necessary, then there are more efficient methods for accessing devices, for example "pass-through".

ACRN device model is a placeholder of the UOS. It allocates memory for the User OS, configures and initializes the devices used by the UOS, loads the virtual firmware, initializes the virtual CPU state, and invokes the ACRN hypervisor service to execute the guest instructions. ACRN Device model is an application running in the Service OS that emulates devices based on command line configuration, as shown in the architecture diagram Figure 1.5 below:

ACRN Device model incorporates these three aspects:

Device Emulation: ACRN Device model provides device emulation routines that register their I/O handlers to the I/O dispatcher. When there is an I/O request from the User OS device, the I/O dispatcher sends this request to the corresponding device emulation routine.

I/O Path: see *ACRN-io-mediator* below

VHM: The Virtio and Hypervisor Service Module is a kernel module in the Service OS acting as a middle layer to support the device model. The VHM and its client handling flow is described below:

1. ACRN hypervisor IOREQ is forwarded to the VHM by an upcall notification to the SOS.
2. VHM will mark the IOREQ as "in process" so that the same IOREQ will not pick up again. The IOREQ will be sent to the client for handling. Meanwhile, the VHM is ready for another IOREQ.
3. IOREQ clients are either an SOS Userland application or a Service OS Kernel space module. Once the IOREQ is processed and completed, the Client will issue an IOCTL call to the VHM to notify an IOREQ state change. The VHM then checks and hypercalls to ACRN hypervisor notifying it that the IOREQ has completed.

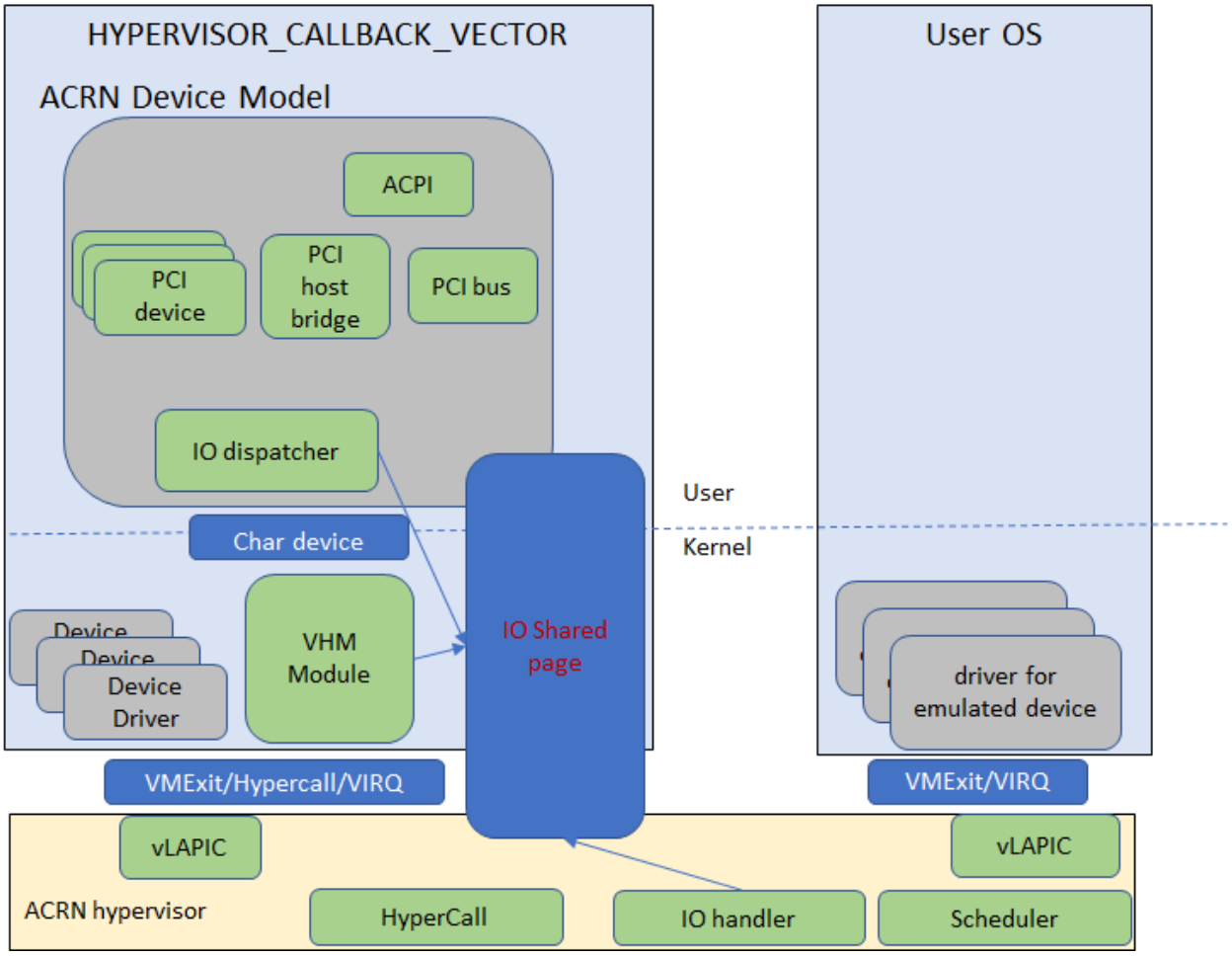


Figure 1.5: ACRN Device Model

Note: Userland: dm as ACRN Device Model.

Kernel space: VBS-K, MPT Service, VHM itself

1.1.7 Device pass through

At the highest level, device pass-through is about providing isolation of a device to a given guest operating system so that the device can be used exclusively by that guest.

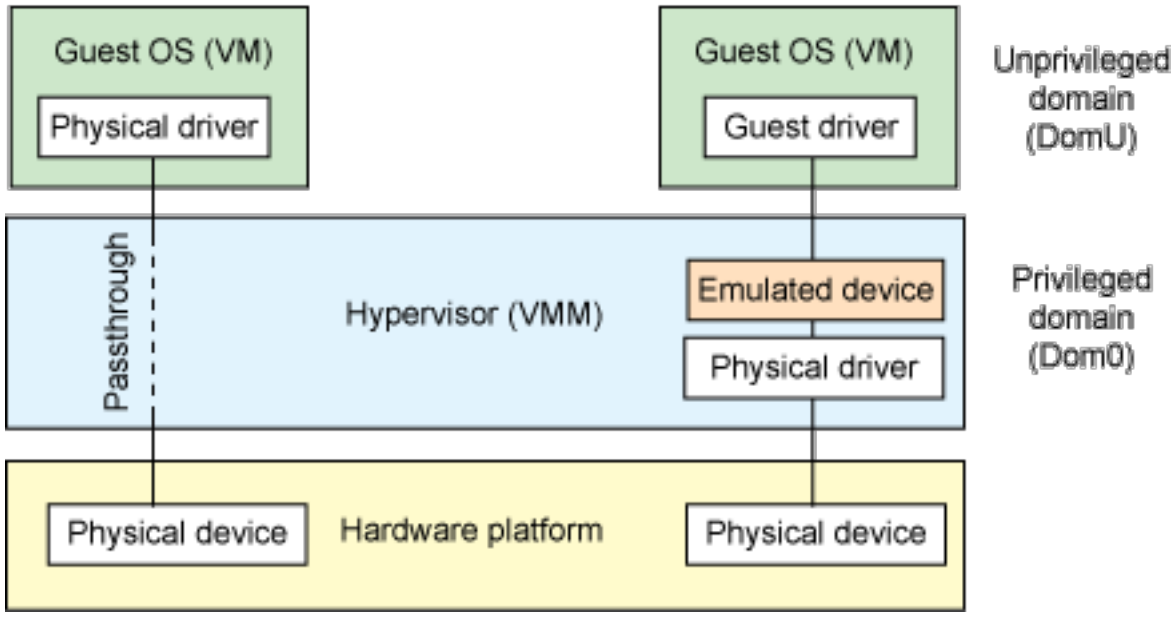


Figure 1.6: Device Passthrough

Near-native performance can be achieved by using device passthrough. This is ideal for networking applications (or those with high disk I/O needs) that have not adopted virtualization because of contention and performance degradation through the hypervisor (using a driver in the hypervisor or through the hypervisor to a user space emulation). Assigning devices to specific guests is also useful when those devices inherently wouldn't be shared. For example, if a system includes multiple video adapters, those adapters could be passed through to unique guest domains.

Finally, there may be specialized PCI devices that only one guest domain uses, so they should be passed through to the guest. Individual USB ports could be isolated to a given domain too, or a serial port (which is itself not shareable) could be isolated to a particular guest. In ACRN hypervisor, we support USB controller Pass through only and we don't support pass through for a legacy serial port, (for example 0x3f8).

Hardware support for device passthrough

Intel's current processor architectures provides support for device pass-through with VT-d. VT-d maps guest physical address to machine physical address, so device can use guest physical address directly. When this mapping occurs, the hardware takes care of access (and protection), and the guest operating system can use the device as if it were a non-virtualized system. In addition to mapping guest to physical memory, isolation prevents this device from accessing memory belonging to other guests or the hypervisor.

Another innovation that helps interrupts scale to large numbers of VMs is called Message Signaled Interrupts (MSI). Rather than relying on physical interrupt pins to be associated with a guest, MSI transforms interrupts into messages

that are more easily virtualized (scaling to thousands of individual interrupts). MSI has been available since PCI version 2.2 but is also available in PCI Express (PCIe), where it allows fabrics to scale to many devices. MSI is ideal for I/O virtualization, as it allows isolation of interrupt sources (as opposed to physical pins that must be multiplexed or routed through software).

Hypervisor support for device passthrough

By using the latest virtualization-enhanced processor architectures, hypervisors and virtualization solutions can support device pass-through (using VT-d), including Xen, KVM, and ACRN hypervisor. In most cases, the guest operating system (User OS) must be compiled to support pass-through, by using kernel build-time options. Hiding the devices from the host VM may also be required (as is done with Xen using pciback). Some restrictions apply in PCI, for example, PCI devices behind a PCIe-to-PCI bridge must be assigned to the same guest OS. PCIe does not have this restriction.

1.1.8 ACRN I/O mediator

Figure 1.7 shows the flow of an example I/O emulation path.

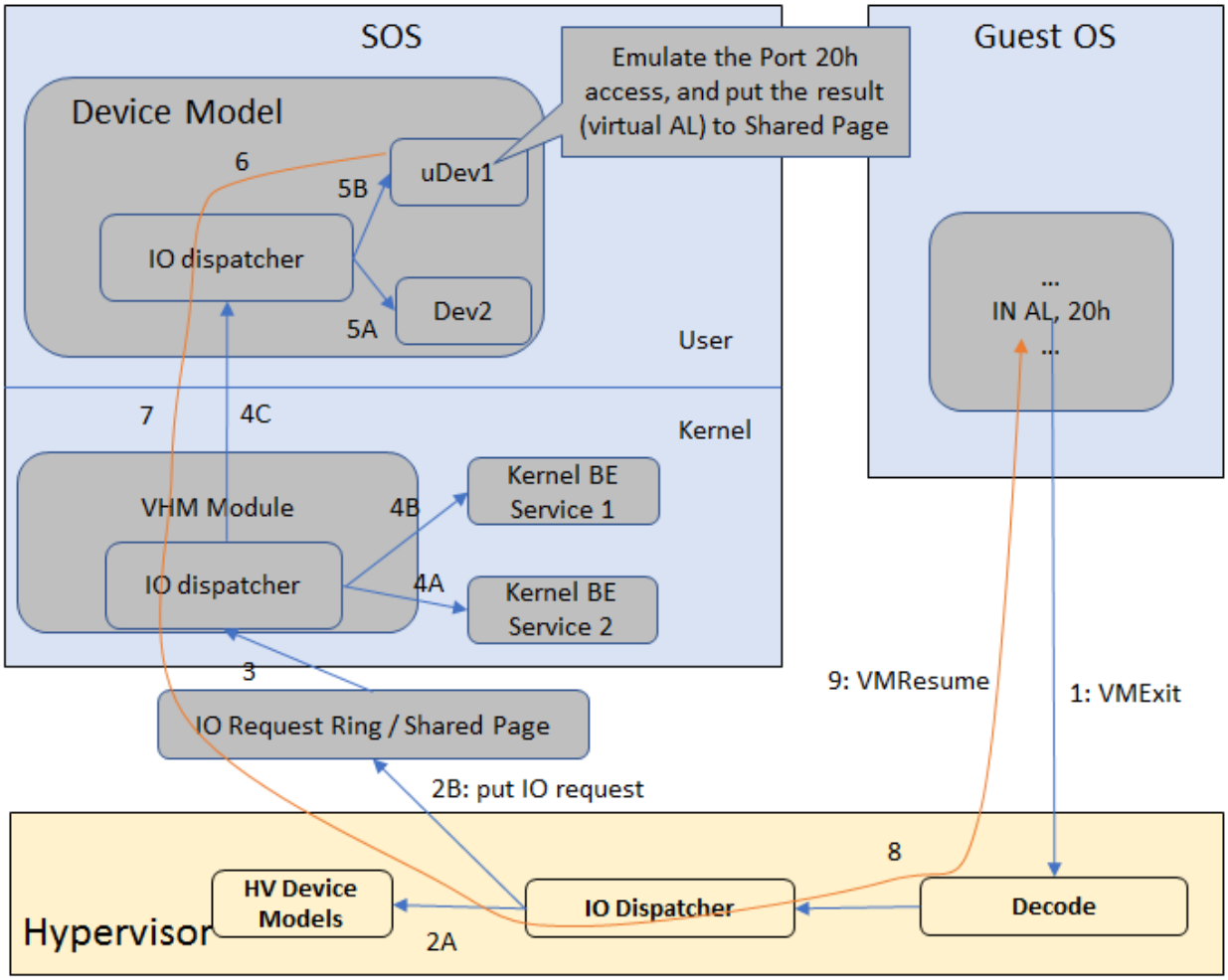


Figure 1.7: I/O Emulation Path

Following along with the numbered items in Figure 1.7:

1. When a guest execute an I/O instruction (PIO or MMIO), a VM exit happens. ACRN hypervisor takes control, and analyzes the the VM exit reason, which is a VMX_EXIT_REASON_IO_INSTRUCTION for PIO access.
2. ACRN hypervisor fetches and analyzes the guest instruction, and notices it is a PIO instruction (`in AL, 20h` in this example), and put the decoded information (including the PIO address, size of access, read/write, and target register) into the shared page, and notify/interrupt the SOS to process.
3. The Virtio and hypervisor service module (VHM) in SOS receives the interrupt, and queries the IO request ring to get the PIO instruction details.
4. It checks to see if any kernel device claims ownership of the IO port: if a kernel module claimed it, the kernel module is activated to execute its processing APIs. Otherwise, the VHM module leaves the IO request in the shared page and wakes up the device model thread to process.
5. The ACRN device model follow the same mechanism as the VHM. The I/O processing thread of device model queries the IO request ring to get the PIO instruction details and checks to see if any (guest) device emulation module claims ownership of the IO port: if a module claimed it, the module is invoked to execute its processing APIs.
6. After the ACRN device module completes the emulation (port IO 20h access in this example), (say uDev1 here), uDev1 puts the result into the shared page (in register AL in this example).
7. ACRN device model then returns control to ACRN hypervisor to indicate the completion of an IO instruction emulation, typically thru VHM/hypercall.
8. The ACRN hypervisor then knows IO emulation is complete, and copies the result to the guest register context.
9. The ACRN hypervisor finally advances the guest IP to indicate completion of instruction execution, and resumes the guest.

The MMIO path is very similar, except the VM exit reason is different. MMIO access usually is trapped thru VMX_EXIT_REASON_EPT_VIOLATION in the hypervisor.

1.1.9 Virtio framework architecture

Virtio is an abstraction for a set of common emulated devices in any type of hypervisor. In the ACRN reference stack, our implementation is compatible with Virtio spec 0.9 and 1.0. By following this spec, virtual environments and guests should have a straightforward, efficient, standard and extensible mechanism for virtual devices, rather than boutique per-environment or per-OS mechanisms.

Virtio provides a common frontend driver framework which not only standardizes device interfaces, but also increases code reuse across different virtualization platforms.

To better understand Virtio, especially its usage in the ACRN project, several key concepts of Virtio are highlighted here:

Front-End Virtio driver (a.k.a. frontend driver, or FE driver in this document) Virtio adopts a frontend-backend architecture, which enables a simple but flexible framework for both frontend and backend Virtio driver. The FE driver provides APIs to configure the interface, pass messages, produce requests, and notify backend Virtio driver. As a result, the FE driver is easy to implement and the performance overhead of emulating device is eliminated.

Back-End Virtio driver (a.k.a. backend driver, or BE driver in this document) Similar to FE driver, the BE driver, runs either in user-land or kernel-land of host OS. The BE driver consumes requests from FE driver and send them to the host's native device driver. Once the requests are done by the host native device driver, the BE driver notifies the FE driver about the completeness of the requests.

Straightforward: Virtio devices as standard devices on existing Buses Instead of creating new device buses from scratch, Virtio devices are built on existing buses. This gives a straightforward way for both FE and BE drivers to interact with each other. For example, FE driver could read/write registers of the device, and the virtual

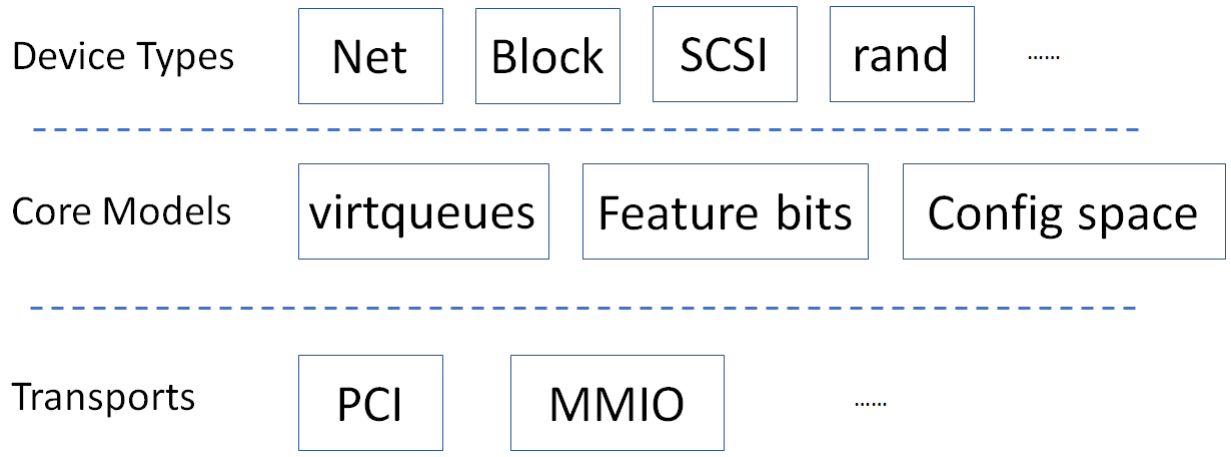


Figure 1.8: Virtio Architecture

device could interrupt FE driver, on behalf of the BE driver, in case of something is happening. Currently Virtio supports PCI/PCIe bus and MMIO bus. In ACRN project, only PCI/PCIe bus is supported, and all the Virtio devices share the same vendor ID 0x1AF4.

Efficient: batching operation is encouraged Batching operation and deferred notification are important to achieve high-performance I/O, since notification between FE and BE driver usually involves an expensive exit of the guest. Therefore batching operating and notification suppression are highly encouraged if possible. This will give an efficient implementation for the performance critical devices.

Standard: virtqueue All the Virtio devices share a standard ring buffer and descriptor mechanism, called a virtqueue, shown in Figure 6. A virtqueue is a queue of scatter-gather buffers. There are three important methods on virtqueues:

- `add_buf` is for adding a request/response buffer in a virtqueue
- `get_buf` is for getting a response/request in a virtqueue, and
- `kick` is for notifying the other side for a virtqueue to consume buffers.

The virtqueues are created in guest physical memory by the FE drivers. The BE drivers only need to parse the virtqueue structures to obtain the requests and get the requests done. How virtqueue is organized is specific to the User OS. In the implementation of Virtio in Linux, the virtqueue is implemented as a ring buffer structure called vring.

In ACRN, the virtqueue APIs can be leveraged directly so users don't need to worry about the details of the virtqueue. Refer to the User OS for more details about the virtqueue implementations.

Extensible: feature bits A simple extensible feature negotiation mechanism exists for each virtual device and its driver. Each virtual device could claim its device specific features while the corresponding driver could respond to the device with the subset of features the driver understands. The feature mechanism enables forward and backward compatibility for the virtual device and driver.

In the ACRN reference stack, we implement user-land and kernel space as shown in Figure 1.9:

In the Virtio user-land framework, the implementation is compatible with Virtio Spec 0.9/1.0. The VBS-U is statically linked with Device Model, and communicates with Device Model through the PCIe interface: PIO/MMIO or MSI/MSIx. VBS-U accesses Virtio APIs through user space vring service API helpers. User space vring service API helpers access shared ring through remote memory map (mmap). VHM maps UOS memory with the help of ACRN Hypervisor.

VBS-U offloads data plane processing to VBS-K. VBS-U initializes VBS-K at the right timings, for example. The

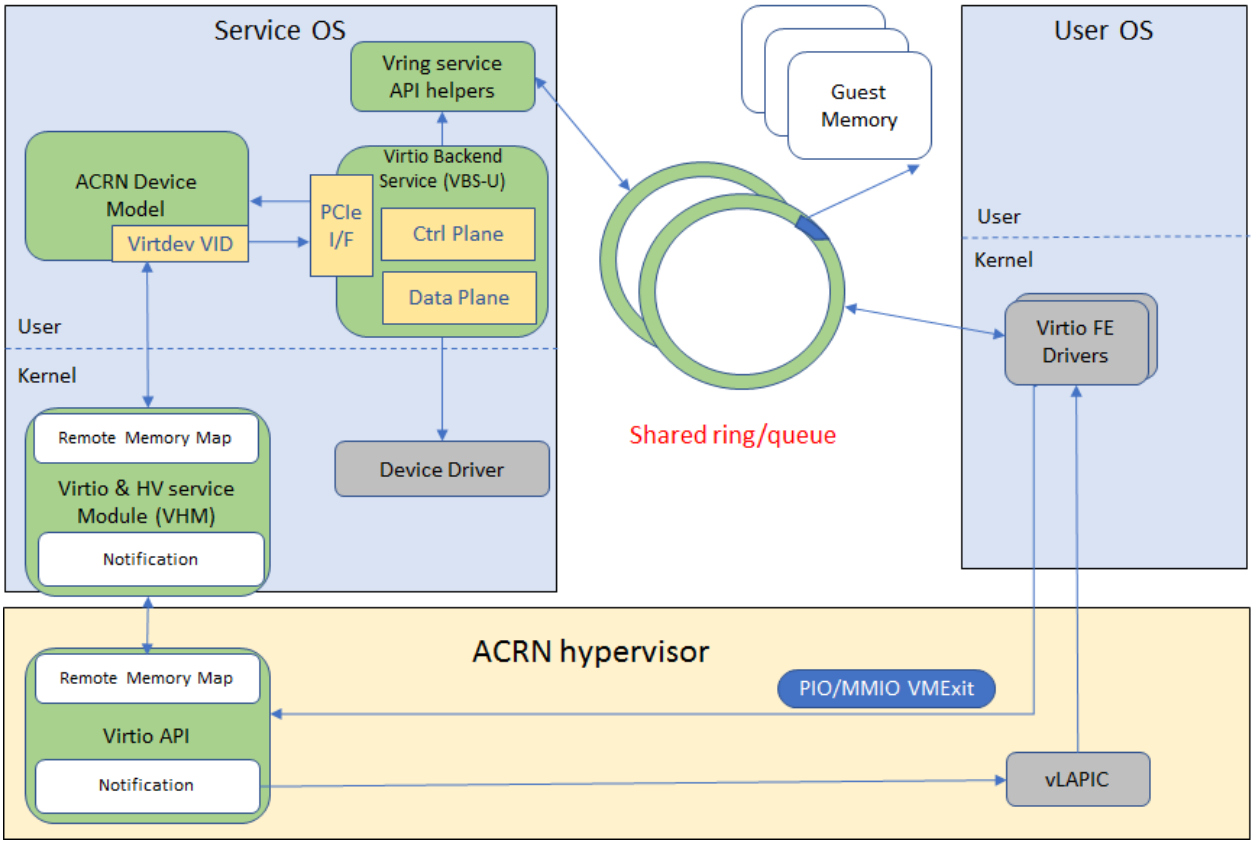


Figure 1.9: Virtio Framework - User Land

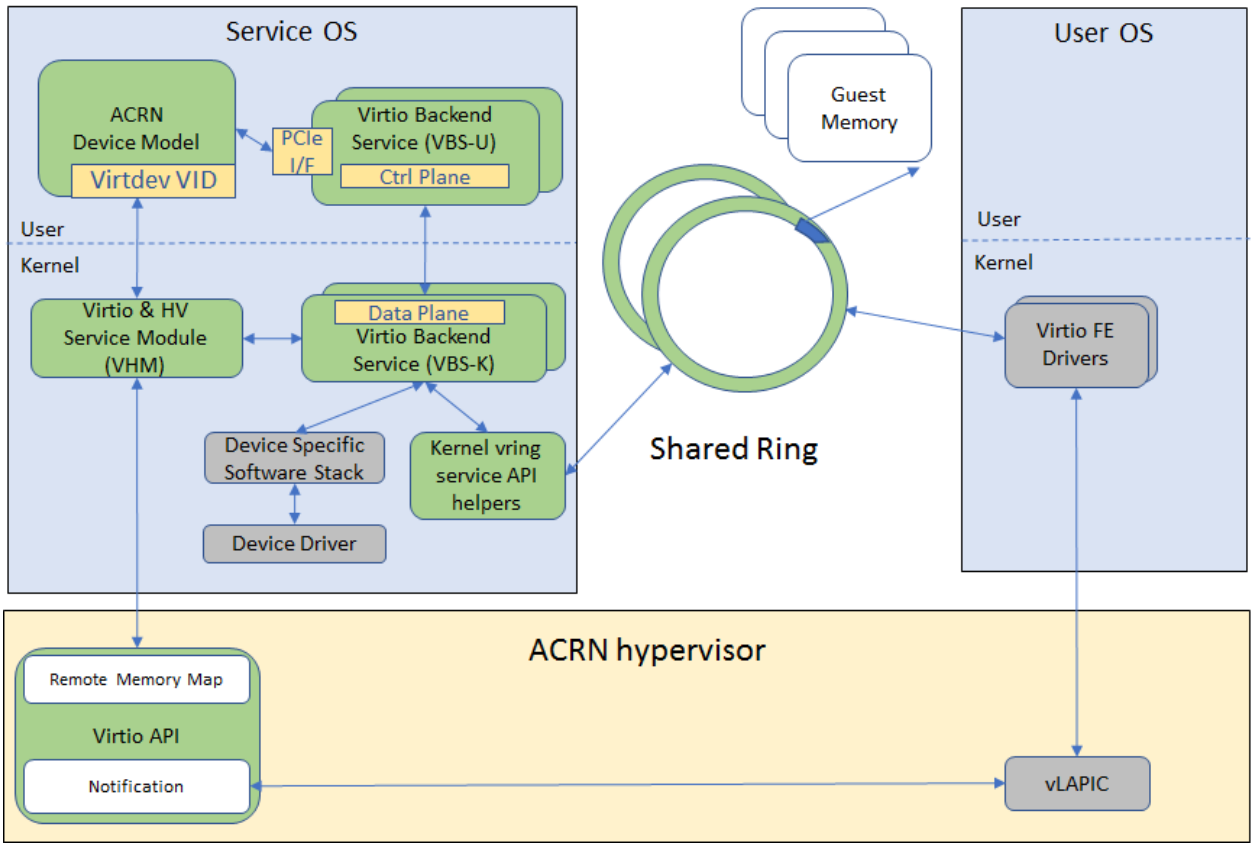


Figure 1.10: Virtio Framework - Kernel Space

FE driver sets VIRTIO_CONFIG_S_DRIVER_OK to avoid unnecessary device configuration changes while running. VBS-K can access shared rings through VBS-K virtqueue APIs. VBS-K virtqueue APIs are similar to VBS-U virtqueue APIs. VBS-K registers as VHM client(s) to handle a continuous range of registers

There may be one or more VHM-clients for each VBS-K, and there can be a single VHM-client for all VBS-Ks as well. VBS-K notifies FE through VHM interrupt APIs.

1.2 Supported Hardware

We welcome community contributions to help build Project ACRN support for a broad collection of architectures and platforms.

This initial release of Project ACRN has been tested on the following hardware platform.

1.2.1 Intel Apollo Lake NUC

- Intel NUC Kit NUC6CAYH Reference
- Intel® Celeron® Processor J3455
- Tested NUC with 8GB of RAM and using an 128GB SSD

You can read a full AnandTech review of the NUC6CAYH NUC Kit and search online for where to purchase this NUC from Amazon, SimplyNUC, and other vendors. Be sure to purchase the NUC with the recommended memory and storage options noted above.

1.3 Getting Started Guide

After reading the *Introduction to Project ACRN*, use this guide to get started using ACRN in a reference setup. We'll show how to set up your development and target hardware, and then how to boot up the ACRN hypervisor and the Clear Linux Service OS and Guest OS on the Intel® NUC.

1.3.1 Hardware setup

The Intel® NUC (NUC6CAYH) is the supported reference target platform for ACRN work, as described in *Supported Hardware*, and is the only platform currently tested with these setup instructions.

The recommended NUC hardware configuration is:

- NUC: NUC Kit NUC6CAYH
- UEFI BIOS (version 0047).
- Memory: 8G DDR3
- SSD: 120G SATA

1.3.2 Software setup

Firmware update on the NUC

You may need to update to the latest UEFI firmware for the NUC hardware. Follow these BIOS Update Instructions for downloading and flashing an updated BIOS for the NUC.

Set up a Clear Linux Operating System

Currently, an installable version of ARCN does not exist. Therefore, you need to setup a base Clear Linux OS to bootstrap ACRN on the NUC. You'll need a network connection for your NUC to complete this setup.

Note: ACRN requires Clear Linux version 22140 or newer. The instructions below have been validated with version 22140 and need some adjustment to work with newer versions. You will see a note when the instruction needs to be adjusted.

---

1. Download the compressed Clear installer image from https://download.clearlinux.org/releases/22140/clear/clear-22140-installer.img.xz and follow the Clear Linux installation guide as a starting point for installing Clear Linux onto your NUC. Follow the recommended options for choosing an **Automatic** installation type, and using the NUC's storage as the target device for installation (overwriting the existing data and creating three partitions on the NUC's SSD drive).
2. After installation is complete, boot into Clear Linux, login as **root**, and set a password.
3. Clear Linux is set to automatically update itself. We recommend that you disable this feature to have more control over when the updates happen. Use this command to disable the autoupdate feature:

```
# swupd autoupdate --disable
```

4. Use the `swupd bundle-add` command and add these Clear Linux bundles:

```
# swupd bundle-add vim network-basic service-os kernel-pk
```

Table 1.1: Clear Linux bundles

Bundle	Description
vim	vim text editor
network-basic	Run network utilities and modify network settings
service-os	Add the acrn hypervisor, the acrn devicemodel and Service OS kernel
kernel-pk	Run the Intel "PK" kernel(product kernel source) and enterprise-style kernel with backports

Add the ACRN hypervisor to the EFI Partition

In order to boot the ACRN SOS on the NUC, you'll need to add it to the EFI partition. Follow these steps:

1. Mount the EFI partition and verify you have the following files:

```
# mount /dev/sda1 /mnt

# ls -1 /mnt/EFI/org.clearlinux
bootloaderx64.efi
kernel-org.clearlinux.native.4.16.6-563
kernel-org.clearlinux.pk414-sos.4.14.34-28
kernel-org.clearlinux.pk414-standard.4.14.34-28
loaderx64.efi
```

Note: The Clear Linux project releases updates often, sometimes twice a day, so make note of the specific kernel versions (`*-sos` and `*-standard`) listed on your system, as you will need them later.

---

2. Put the `acrn.efi` hypervisor application (included in the Clear Linux release) on the EFI partition with:

```
# mkdir /mnt/EFI/acrn
# cp /usr/share/acrn/acrn.efi /mnt/EFI/acrn/
```

3. Configure the EFI firmware to boot the ACRN hypervisor by default

   The ACRN hypervisor (`acrn.efi`) is an EFI executable loaded directly by the platform EFI firmware. It then in turns loads the Service OS bootloader. Use the `efibootmgr` utility to configure the EFI firmware and add a new entry that loads the ACRN hypervisor.

```
# efibootmgr -c -l "\EFI\acrn\acrn.efi" -d /dev/sda1 -p 1 -L ACRN
# cd /mnt/EFI/org.clearlinux/
# cp bootloaderx64.efi bootloaderx64_origin.efi
```

Note: Be aware that a Clearlinux update that includes a kernel upgrade will reset the boot option changes you just made.. A Clearlinux update could happen automatically (if you have not disabled it as described above), if you later install a new bundle to your system, or simply if you decide to trigger an update manually. Whenever that happens, double-check the platform boot order using `efibootmgr -v` and modify it if needed.

---

4. Create a boot entry for the ACRN Service OS by copying a provided `acrn.conf` and editing it to account for the kernel versions noted in a previous step.

   It must contain these settings:

Setting	Description
title	Text to show in the boot menu
linux	Linux kernel for the Service OS (*-sos)
options	Options to pass to the Service OS kernel (kernel parameters)

A starter acrn.conf configuration file is included in the Clear Linux release and is also available in the acrn-hypervisor GitHub repo as acrn.conf as shown here:

Code Block 1.1: acrn-hypervisor/bsp/uefi/clearlinux/acrn.conf

```
title The ACRN Service OS
linux   /EFI/org.clearlinux/kernel-org.clearlinux.pk414-sos.4.14.23-19
options pci_devices_ignore=(0:18:2) maxcpus=1 console=tty0 console=ttyS0 i915.
↪nuclear_pageflip=1 root=PARTUUID=<UUID of rootfs partition> rw rootwait ignore_
↪loglevel no_timer_check consoleblank=0 i915.tsd_init=7 i915.tsd_delay=2000 i915.
↪avail_planes_per_pipe=0x00000F i915.domain_plane_owners=0x011111110000 i915.
↪enable_guc_loading=0 i915.enable_guc_submission=0 i915.enable_preemption=1 i915.
↪context_priority_mode=2 i915.enable_gvt=1 hvlog=2M@0x1FE00000␣
↪cma=2560M@0x100000000-0
```

On the NUC, copy the `acrn.conf` file to the EFI partition we mounted earlier:

```
# cp /usr/share/acrn/demo/acrn.conf /mnt/loader/entries/
```

You will need to edit this file to adjust the kernel version (`linux` section) and also insert the `PARTUUID` of your `/dev/sda3` partition (`root=PARTUUID=<><UUID of rootfs partition>`) in the `options` section.

Use `blkid` to find out what your `/dev/sda3` `PARTUUID` value is.

5. Add a timeout period for Systemd-Boot to wait, otherwise it will not present the boot menu and will always boot the base Clear Linux

```
# clr-boot-manager set-timeout 20
# clr-boot-manager update
```

6. Reboot and select "The ACRN Service OS" to boot, as shown in Figure 1.11:

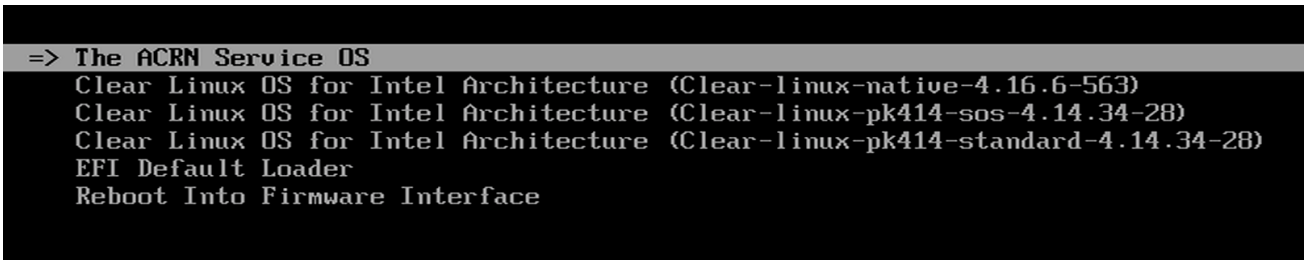


Figure 1.11: ACRN Service OS Boot menu

7. After booting up the ACRN hypervisor, the Service OS will be launched automatically by default, as shown in Figure 1.12:

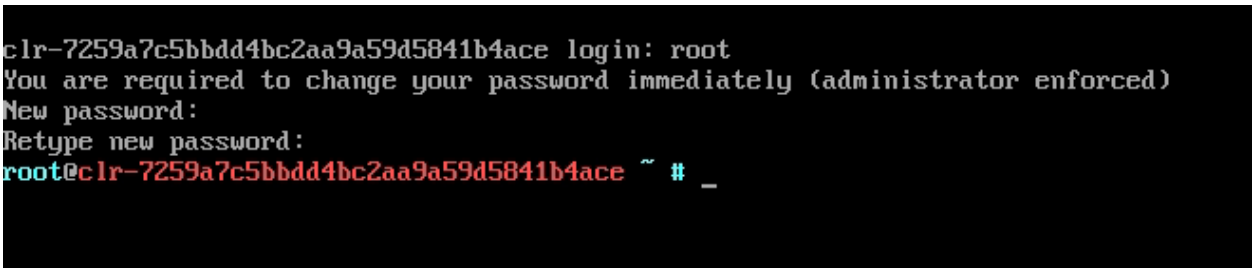


Figure 1.12: Service OS Console

> **Note:** You may need to hit `Enter` to get a clean login prompt

8. From here you can login as root using the password you set previously when you installed Clear Linux.

Create a Network Bridge

Without a network bridge, the SOS and UOS are not able to talk to each other.

A sample bridge.sh is included in the Clear Linux release, and is also available in the acrn-devicemodel GitHub repo (in the samples folder) as shown here:

Code Block 1.2: acrn-devicemodel/samples/bridge.sh

```
#!/bin/bash

# Instructions to create systemd-networkd configuration files:

if [ ! -e /etc/systemd/network ]; then
    mkdir -p /etc/systemd/network

    # /etc/systemd/network/acrn.network
```

```
    cat <<EOF>/etc/systemd/network/acrn.network
[Match]
Name=e*

[Network]
Bridge=acrn-br0
EOF


    # /etc/systemd/network/acrn.netdev
    cat <<EOF>/etc/systemd/network/acrn.netdev
[NetDev]
Name=acrn-br0
Kind=bridge
EOF

    # /etc/systemd/network/eth.network
    cat <<EOF>/etc/systemd/network/eth.network
[Match]
Name=acrn-br0

[Network]
DHCP=ipv4
EOF

    # need to mask 80-dhcp.network and 80-virtual.network
    ln -s /dev/null /etc/systemd/network/80-dhcp.network
    ln -s /dev/null /etc/systemd/network/80-virtual.network

    # should get specifc list of taps
    # /etc/systemd/network/acrn_tap0.netdev
    cat <<EOF>/etc/systemd/network/acrn_tap0.netdev
[NetDev]
Name=acrn_tap0
Kind=tap
EOF

    # restart systemd-network to create the devices
    # and bind network address to the bridge
    systemctl restart systemd-networkd
fi

# add tap device under the bridge
ifconfig acrn_tap0 up
brctl addif acrn-br0 acrn_tap0
```

By default, the script is located in the `/usr/share/acrn/demo/` directory. Run it to create a network bridge:

```
# cd /usr/share/acrn/demo/
# ./bridge.sh
# cd
```

Set up Reference UOS

1. On your NUC, download the pre-built reference Clear Linux UOS image into your (root) home directory:

```
# cd ~
# curl -O https://download.clearlinux.org/releases/22140/clear/clear-22140-kvm.
↪img.xz
```

Note: In case you want to use or try out a newer version of Clear Linux as the UOS, you can download the latest from http://download.clearlinux.org/image. Make sure to adjust the steps described below accordingly (image file name and kernel modules version).

2. Uncompress it:

```
# unxz clear-22140-kvm.img.xz
```

3. Deploy the UOS kernel modules to UOS virtual disk image (note: you'll need to use the same **standard** image version number noted in step 1 above):

```
# losetup -f -P --show /root/clear-22140-kvm.img
# mount /dev/loop0p3 /mnt
# cp -r /usr/lib/modules/4.14.34-28.pk414-standard /mnt/lib/modules/
# umount /mnt
# sync
```

4. Edit and Run the launch_uos.sh script to launch the UOS.

   A sample launch_uos.sh is included in the Clear Linux release, and is also available in the acrn-devicemodel GitHub repo (in the samples folder) as shown here:

Code Block 1.3: acrn-devicemodel/samples/launch_uos.sh

```
#!/bin/bash

function launch_clear()
{
vm_name=vm$1


#check if the vm is running or not
vm_ps=$(pgrep -a -f acrn-dm)
result=$(echo $vm_ps | grep "${vm_name}")
if [[ "$result" != "" ]]; then
  echo "$vm_name is running, can't create twice!"
  exit
fi

#for memsize setting
mem_size=1000M

acrn-dm -A -m $mem_size -c $2 -s 0:0,hostbridge -s 1:0,lpc -l com1,stdio \
  -s 5,virtio-console,@pty:pty_port \
  -s 6,virtio-hyper_dmabuf \
  -s 3,virtio-blk,/root/clear-21260-kvm.img \
  -s 4,virtio-net,tap0 \
  -k /usr/lib/kernel/org.clearlinux.pk414-standard.4.14.23-19 \
  -B "root=/dev/vda3 rw rootwait maxcpus=$2 nohpet console=tty0 console=hvc0 \
  console=ttyS0 no_timer_check ignore_loglevel log_buf_len=16M \
  consoleblank=0 tsc=reliable i915.avail_planes_per_pipe=$4 \
  i915.enable_hangcheck=0 i915.nuclear_pageflip=1" $vm_name
```

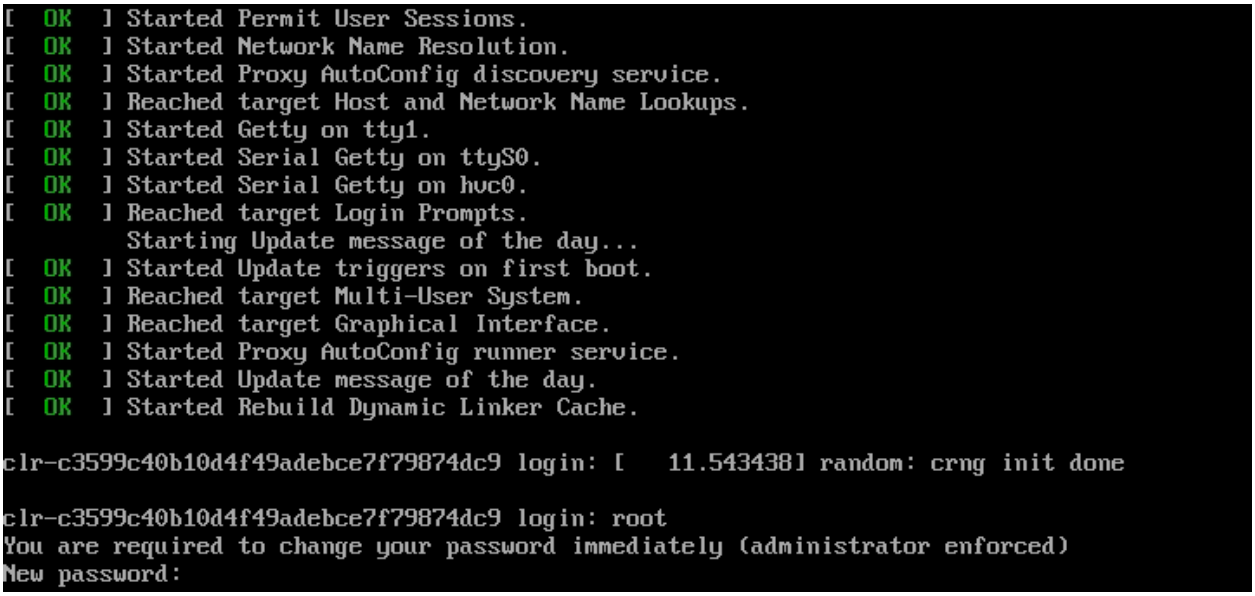
```
}

launch_clear 2 1 "64 448 8" 0x00000C clear
```

Note: In case you have downloaded a different Clear Linux image than the one above (`clear-22140-kvm.img.xz`), you will need to modify the Clear Linux file name and version number highlighted above (the `-s 3,virtio-blk` argument) to match what you have downloaded above. Likewise, you may need to adjust the kernel file name on the second line highlighted (check the exact name to be used using: `ls /usr/lib/kernel/org.clearlinux*-standard*`).

By default, the script is located in the `/usr/share/acrn/demo/` directory. You can edit it there, and then run it to launch the User OS:

```
# cd /usr/share/acrn/demo/
# ./launch_uos.sh
```

5. At this point, you've successfully booted the ACRN hypervisor, SOS, and UOS:



1.3.3 Build ACRN from Source

If you would like to build ACRN hypervisor and device model from source, follow these steps.

Install build tools and dependencies

ARCN development is supported on popular Linux distributions, each with their own way to install development tools:

- On a Clear Linux development system, install the `os-clr-on-clr` bundle to get the necessary tools:

```
$ sudo swupd bundle-add os-clr-on-clr
```

- On a Ubuntu/Debian development system:

```
$ sudo apt install git \
     gnu-efi \
     libssl-dev \
     libpciaccess-dev \
     uuid-dev
```

- On a Fedora/Redhat development system:

```
$ sudo dnf install gcc \
     gnu-efi-devel \
     libuuid-devel \
     openssl-devel \
     libpciaccess-devel
```

- On a CentOS development system:

```
$ sudo yum install gcc \
        gnu-efi-devel \
        libuuid-devel \
        openssl-devel \
        libpciaccess-devel
```

Build the hypervisor and device model

1. Download the ACRN hypervisor and build it.

```
$ git clone https://github.com/projectacrn/acrn-hypervisor
$ cd acrn-hypervisor
$ make PLATFORM=uefi
```

   The build results are found in the `build` directory.

2. Download the ACRN device model and build it.

```
$ git clone https://github.com/projectacrn/acrn-devicemodel
$ cd acrn-devicemodel
$ make
```

   The build results are found in the `build` directory.

Follow the same instructions to boot and test the images you created from your build.

1.4 Developer Primer

This Developer Primer introduces the fundamental components of ACRN and the virtualization technology used by this open source reference stack. Code level documentation and additional details can be found by consulting the *API Documentation* documentation and the source code in GitHub.

The ACRN Hypervisor acts as a host with full control of the processor(s) and the hardware (physical memory, interrupt management and I/O). It provides the User OS with an abstraction of a virtual platform, allowing the guest to behave as if were executing directly on a logical processor.

1.4.1 Source Tree Structure

Understanding the ACRN hypervisor and the ACRN device model source tree structure is helpful for locating the code associated with a particular hypervisor and device emulation feature. The ACRN hypervisor and the ACRN device model source tree provides the following top-level directories:

ACRN hypervisor source tree

arch/x86/ hypervisor architecture, which includes arch x86 related source files to run the hypervisor, such as CPU, memory, interrupt, and VMX.

boot/ boot stuff mainly including ACPI related

bsp/ board support package, used to support NUC with UEFI

common/ common source files for hypervisor, which including VM hypercall definition, VM main loop, and VM software loader

debug/ all debug related source files, which will not be compiled for release version, mainly including console, uart, logmsg and shell

include/ include files for all public APIs (doxygen comments in these source files are used to generate the *API Documentation* documentation)

lib/ runtime service libraries

ACRN Device Model source tree

core/ ACRN Device model core logic (main loop, SOS interface, etc.)

hw/ Hardware emulation code, with the following subdirectories:

acpi/ ACPI table generator.

pci/ PCI devices, including VBS-Us (Virtio backend drivers in user-space).

platform/ platform devices such as uart, and keyboard.

include/ include files for all public APIs (doxygen comments in these source files are used to generate the *API Documentation* documentation)

samples/ include files for all public APIs (doxygen comments in these source

ACRN documentation source tree

Project ACRN documentation is written using the reStructuredText markup language (.rst file extension) with Sphinx extensions, and processed using Sphinx to create a formatted stand-alone website, (the one you're reading now.) Developers can view this content either in its raw form as .rst markup files in the acrn-documentation repo, or you can generate the HTML content and view it with a web browser directly on your workstation, useful if you're contributing documentation to the project.

api/ ReST files for API document generation

custom-doxygen/ Customization files for doxygen-generated html output (while generated, we currently don't include the doxygen html output but do use the XML output to feed into the Sphinx-generation process)

getting_started/ ReST files and images for the Getting Started Guide

primer/ ReST files and images for the Developer Primer

images/ Image files not specific to a document (logos, and such)

introduction/ ReST files and images for the Introduction to Project ACRN

scripts/ Files used to assist building the documentation set

static/ Sphinx folder for extras added to the generated output (such as custom CSS additions)

1.4.2 CPU virtualization

The ACRN hypervisor uses static partitioning of the physical CPU cores, providing each User OS a virtualized environment containing at least one statically assigned physical CPU core. The CPUID features for a partitioned physical core is the same as the native CPU features. CPU power management (Cx/Px) is managed by the User OS.

The supported Intel® NUC platform (see *Supported Hardware*) has a CPU with four cores. The Service OS is assigned one core and the other three cores are assigned to the User OS. `XSAVE` and `XRSTOR` instructions (used to perform a full save/restore of the extended state in the processor to/from memory) are currently not supported in the User OS. (The kernel boot parameters must specify `noxsave`). Processor core sharing among User OSes is planned for a future release.

The following sections introduce CPU virtualization related concepts and technologies.

Host GDT

The ACRN hypervisor initializes the host Global Descriptor Table (GDT), used to define the characteristics of the various memory areas during program execution. Code Segment `CS: 0x8` and Data Segment `DS: 0x10` are configured as Hypervisor selectors, with their settings in host the GDT as shown in Figure 1.13:

```
.org    HOST_GDT + HOST_GDT_RING0_CODE_SEL     /* Ring 0 Code Sel Descriptor */
    /* Axxx ==> Granularity==1 (4k to 4G)
                64 bit code segment
       xx9x ==> Segment Present
                Ring 0
                Code or Data Segment
       xxxB ==> 64-bit TSS Busy */
    .quad    0x00Af9b000000ffff

.org    HOST_GDT + HOST_GDT_RING0_DATA_SEL     /* Ring 0 Data Sel Descriptor */
    /* Cxxx ==> Granularity == 1
                32-bit Segment
       xx9x ==> Segment Present
                Ring 0
                Code or Data Segment
       xxx3 ==> 16-bit TSS (Busy) */
    .quad    0x00cf93000000ffff
```

Figure 1.13: Host GDT

Host IDT

The ACRN hypervisor installs interrupt gates for both Exceptions and Vectors. That means exceptions and interrupts will automatically disable interrupts. The `HOST_GDT_RING0_CODE_SEL` is used in the Host IDT table.

Guest SMP Booting

The Bootstrap Processor (BSP) vCPU for the User OS boots into x64 long mode directly, while the Application Processors (AP) vCPU boots into real mode. The virtualized Local Advanced Programmable Interrupt Controller (vLAPIC) for the User OS in the hypervisor emulates the INIT/STARTUP signals.

The AP vCPU belonging to the User OS begins in an infinite loop, waiting for an INIT signal. Once the User OS issues a Startup IPI (SIPI) signal to another vCPU, the vLAPIC traps the request, resets the target vCPU, and then enters the `INIT->STARTUP#1->STARTUP#2` cycle to boot the vCPUs for the User OS.

VMX configuration

ACRN hypervisor has the Virtual Machine configuration (VMX) shown in Table 1.2 below. (These configuration settings may change in the future, according to virtualization policies.)

Table 1.2: VMX Configuration

VMX MSR	Bits	Description
MSR_IA32_VMX_PINBASED_CTLS	Bit0 set	Enable External IRQ VM Exit
	Bit6 set	Enable HV pre-40ms Preemption timer
	Bit7 clr	Post interrupt did not support
MSR_IA32_VMX_PROCBASED_CTLS	Bit25 set	Enable I/O bitmap
	Bit28 set	Enable MSR bitmap
	Bit19,20 set	Enable CR8 store/load
MSR_IA32_VMX_PROCBASED_CTLS2	Bit1 set	Enable EPT
	Bit7 set	Allow guest real mode
MSR_IA32_VMX_EXIT_CTLS	Bit15	VMX Exit auto ack vector
	Bit18,19	MSR IA32_PAT save/load
	Bit20,21	MSR IA32_EFER save/load
	Bit9	64-bit mode after VM Exit

CPUID and Guest TSC calibration

User OS access to CPUID will be trapped by ACRN hypervisor, however the ACRN hypervisor will pass through most of the native CPUID information to the guest, except the virtualized CPUID 0x1 (to provide fake x86_model).

The Time Stamp Counter (TSC) is a 64-bit register present on all x86 processors that counts the number of cycles since reset. ACRN hypervisor also virtualizes `MSR_PLATFORM_INFO` and `MSR_ATOM_FSB_FREQ`.

RDTSC/RDTSCP

User OS vCPU reads of `RDTSC`, `RDTSCP`, or `MSR_IA32_TSC_AUX` will not make the VM Exit to the hypervisor. Thus the vCPUID provided by `MSR_IA32_TSC_AUX` can be changed via the User OS.

The `RDTSCP` instruction is widely used by the ACRN hypervisor to identify the current CPU (and read the current value of the processor's time-stamp counter). Because there is no VM Exit for `MSR_IA32_TSC_AUX` msr register, the hypervisor will save and restore the `MSR_IA32_TSC_AUX` value on every VM Exit and Enter. Before the hypervisor restores the host CPU ID, we must not use a `RDTSCP` instruction because it would return the vCPU ID instead of host CPU ID.

CR Register virtualization

Guest CR8 access will make the VM Exit, and is emulated in the hypervisor for vLAPIC to update its PPR register. Guest access to CR3 will not make the VM Exit.

MSR BITMAP

In the ACRN hypervisor, only these module-specific registers (MSR) are supported:

MSR_IA32_TSC_DEADLINE emulates Guest TSC timer program

MSR_PLATFORM_INFO emulates a fake X86 module

MSR_ATOM_FSB_FREQ provides the CPU frequency directly via this MSR to avoid TSC calibration

I/O BITMAP

All User OS I/O port accesses are trapped into the ACRN hypervisor by default. Most of the Service OS I/O port accesses are not trapped into the ACRN hypervisor, allowing the Service OS direct access to the hardware port.

The Service OS I/O trap policy is:

0x3F8/0x3FC for emulated vUART inside hypervisor for SOS only, will be trapped

0x20/0xA0/0x460 for vPIC emulation in hypervisor, will be trapped

0xCF8/0xCFC for hypervisor PCI device interception, will be trapped

Exceptions

The User OS handles its exceptions inside the VM, including page fault, GP, etc. A #MC and #DB exception causes a VM Exit to the ACRN hypervisor console.

1.4.3 Memory virtualization

ACRN hypervisor provides memory virtualization by using a static partition of system memory. Each virtual machine owns its own contiguous partition of memory, with the Service OS staying in lower memory and the User OS instances in high memory. (High memory is memory which is not permanently mapped in the kernel address space, while Low Memory is always mapped, so you can access it in the kernel simply by dereferencing a pointer.) In future implementations, this will evolve to utilize EPT/VT-d.

ACRN hypervisor memory is not visible to any User OS. In the ACRN hypervisor, there are a few memory accesses that need to work efficiently:

- ACRN hypervisor to access host memory
- vCPU per VM to access guest memory
- vCPU per VM to access host memory
- vCPU per VM to access MMIO memory

The rest of this section introduces how these kinds of memory accesses are managed. It gives an overview of physical memory layout, Paravirtualization (MMU) memory mapping in the hypervisor and VMs, and Host-Guest Extended Page Table (EPT) memory mapping for each VM.

Physical Memory Layout

The Physical Memory Layout Example for Service OS & User OS is shown in Figure 1.14 below:

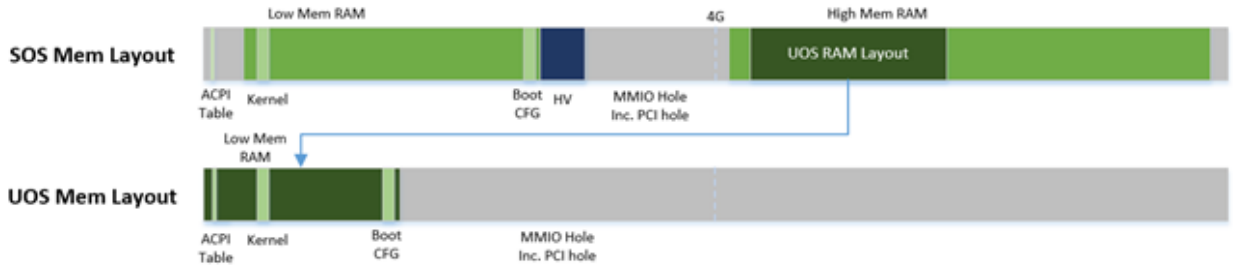


Figure 1.14: Memory Layout

Figure 1.14 shows an example of physical memory layout of the Service and User OS. The Service OS accepts the whole e820 table (all usable memory address ranges not reserved for use by the BIOS) after filtering out the Hypervisor memory too. From the SOS's point of view, it takes control of all available physical memory, including User OS memory, not used by the hypervisor (or BIOS). Each User OSes memory is allocated from (High) SOS memory and the User OS only owns this section of memory control.

Some of the physical memory of a 32-bit machine, needs to be sacrificed by making it hidden so memory-mapped I/O (MMIO) devices have room to communicate. This creates an MMIO hole for VMs to access some range of MMIO addresses directly for communicating to devices; or they may need the hypervisor to trap some range of MMIO to do device emulation. This access control is done through EPT mapping.

PV (MMU) Memory Mapping in the Hypervisor

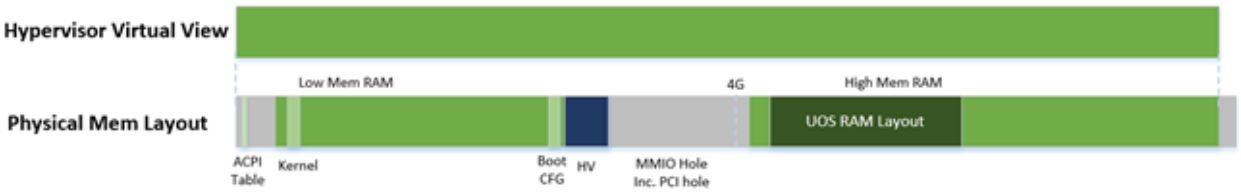


Figure 1.15: ACRN Hypervisor PV Mapping Example

The ACRN hypervisor is trusted and can access and control all system memory, as shown in Figure 1.15. Because the hypervisor is running in protected mode, an MMU page table must be prepared for its PV translation. To simplify things, the PV translation page table is set as a 1:1 mapping. Some MMIO range mappings could be removed if they are not needed. This PV page table is created when the hypervisor memory is first initialized.

PV (MMU) Memory Mapping in VMs

As mentioned earlier, the Primary vCPU starts to run in protected mode when its VM is started. But before it begins, a temporary PV (MMU) page table must be prepared..

This page table is a 1:1 mapping for 4 Gb, and only lives for a short time when the vCPU first runs. After the vCPU starts to run its kernel image (for example Linux*), the kernel will create its own PV page tables, after which, the temporary page table will be obsoleted.

Host-Guest (EPT) Memory Mapping

The VMs (both SOS and UOS) need to create an Extended Page Table (EPT) to access the host physical memory based on its guest physical memory. The guest VMs also need to set an MMIO trap to trigger EPT violations for device emulation (such as IOAPIC, and LAPIC). This memory layout is shown in Figure 1.16:

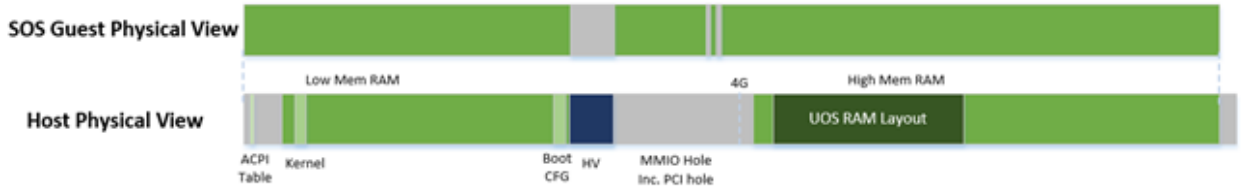


Figure 1.16: SOS EPT Mapping Example

The SOS takes control of all the host physical memory space: its EPT mapping covers almost all of the host memory except that reserved for the hypervisor (HV) and a few MMIO trap ranges for IOAPIC & LAPIC emulation. The guest to host mapping for SOS is 1:1.

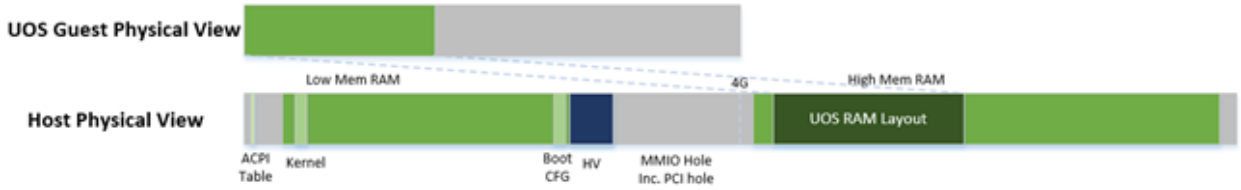


Figure 1.17: UOS EPT Mapping Example

However, for the UOS, its memory EPT mapping is linear but with an offset (as shown in Figure 1.17). The MMIO hole is not mapped to trap all MMIO accesses from the UOS (and do emulating in the device model). To support pass through devices in the future, some MMIO range mapping may be added.

1.4.4 Graphic mediation

Intel® Graphics Virtualization Technology –g (Intel® GVT-g) provides GPU sharing capability to multiple VMs by using a mediated pass-through technique. This allows a VM to access performance critical I/O resources (usually partitioned) directly, without intervention from the hypervisor in most cases.

Privileged operations from this VM are trap-and-emulated to provide secure isolation among VMs. The Hypervisor must ensure that no vulnerability is exposed when assigning performance-critical resource to each VM. When a performance-critical resource cannot be partitioned, a scheduler must be implemented (either in software or hardware) to allow time-based sharing among multiple VMs. In this case, the device must allow the hypervisor to save and restore the hardware state associated with the shared resource, either through direct I/O register read/write (when there is no software invisible state) or through a device-specific context save/restore mechanism (where there is a software invisible state).

In the initial release of Project ACRN, graphic mediation is not enabled, and is planned for a future release.

1.4.5 I/O emulation

The I/O path is explained in the *ACRN I/O mediator* section of the *Introduction to Project ACRN*. The following sections, provide additional device assignment management and PIO/MMIO trap flow introduction.

Device Assignment Management

ACRN hypervisor provides major device assignment management. Since the hypervisor owns all native vectors and IRQs, there must be a mapping table to handle the Guest IRQ/Vector to Host IRQ/Vector. Currently we assign all devices to VM0 except the UART.

If a PCI device (with MSI/MSI-x) is assigned to Guest, the User OS will program the PCI config space and set the guest vector to this device. A Hypercall `CWP_VM_PCI_MSIX_FIXUP` is provided. Once the guest programs the guest vector, the User OS may call this hypercall to notify the ACRN hypervisor. The hypervisor allocates a host vector, creates a guest-host mapping relation, and replaces the guest vector with a real native vector for the device:

PCI MSI/MSI-X PCI Message Signaled Interrupts (MSI/MSX-x) from devices can be triggered from a hypercall when a guest program vectors. All PCI devices are programed with real vectors allocated by the Hypervisor.

PCI/INTx Device assignment is triggered when the guest programs the virtual Advanced I/O Programmable Interrupt Controller (vIOAPC) Redirection Table Entries (RTE).

Legacy Legacy devices are assigned to VM0.

User OS device assignment is similar to the above, except the User OS doesn't call hypercall. Instead, the Guest program PCI configuration space will be trapped into the Device Module, and Device Module may issue hypercall to notify hypervisor the guest vector is changing.

Currently, there are two types of I/O Emulation supported: MMIO and PORTIO trap handling. MMIO emulation is triggered by an EPT violation VMExit only. If there is an EPT misconfiguration and VMExit occurs, the hypervisor will halt the system. (Because the hypervisor set up all EPT page table mapping at the beginning of the Guest boot, there should not be an EPT misconfiguration.)

There are multiple places where I/O emulation can happen - in ACRN hypervisor, Service OS Kernel VHM module, or in the Service OS Userland ACRN Device Module.

PIO/MMIO trap Flow

Here is a description of the PIO/MMIO trap flow:

1. Instruction decoder: get the Guest Physical Address (GPA) from VM Exit, go through gla2gpa() page walker if necessary.
2. Emulate the instruction. Here the hypervisor will have an address range check to see if the hypervisor is interested in this IO port or MMIO GPA access.
3. Hypervisor emulates vLAPIC, vIOAPIC, vPIC, and vUART only (for Service OS only). Any other emulation request are forwarded to the SOS for handling. The vCPU raising the I/O request will halt until this I/O request is processed successfully. An IPI will send to vCPU0 of SOS to notify there is an I/O request waiting for service.
4. Service OS VHM module takes the I/O request and dispatches the request to multiple clients. These clients could be SOS kernel space VBS-K, MPT, or User-land Device model. VHM I/O request server selects a default fallback client responsible to handle any I/O request not handled by other clients. (The Device Manager is the default fallback client.) Each client needs to register its I/O range or specific PCI bus/device/function (BDF) numbers. If an I/O request falls into the client range, the I/O request server will send the request to that client.
5. Multiple clients - fallback client (Device Model in user-land), VBS-K client, MPT client. Once the I/O request emulation completes, the client updates the request status and notifies the hypervisor by a hypercall. Hypervisor picks up that request, do any necessary cleanup, and resume the Guest vCPU.

Most I/O emulation tasks are done by the SOS CPU, and requests come from UOS vCPUs.

1.4.6 Virtual interrupt

All interrupts received by the User OS comes from a virtual interrupt injected by a virtual vLAPIC, vIOAPIC, or vPIC. All device emulation is done inside the SOS Userspace device model. However for performance consideration, vLAPIC, vIOAPIC, and vPIC devices are emulated inside the ACRN hypervisor directly. From the guest point of view, vPIC uses Virtual Wire Mode via vIOAPIC.

The symmetric I/O Mode is shown in Figure 1.18:

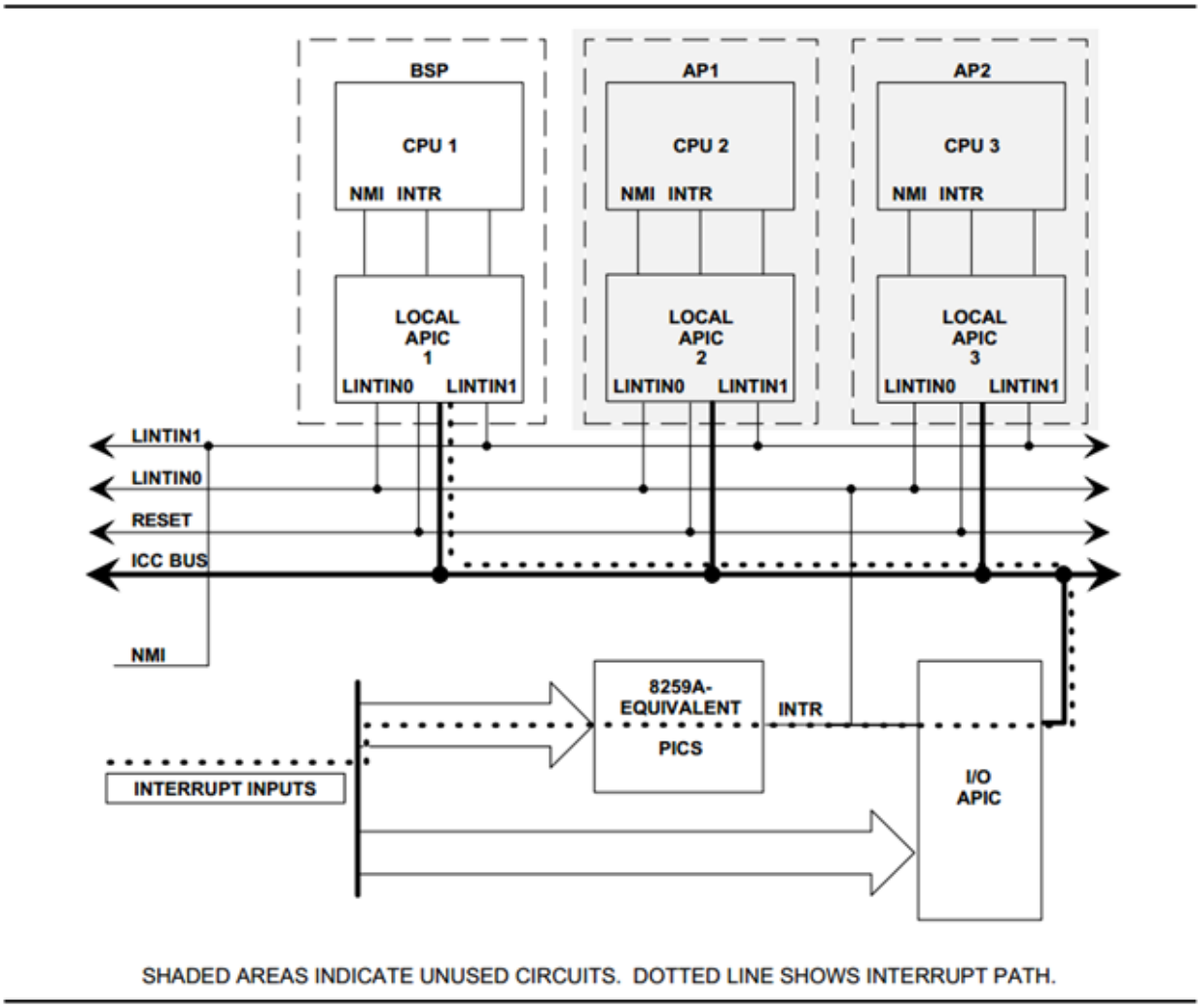


Figure 1.18: Symmetric I/O Mode

Kernel boot param with vPIC add "maxcpu=0" to User OS to use PIC

Kernel boot param with vIOAPIC add "maxcpu=1" (as long as not "0") User OS will use IOAPIC. Keep IOAPIC pin2 as source of PIC.

Virtual LAPIC

The LAPIC (Local Advanced Programmable interrupt Controller) is virtualized for SOS or UOS. The vLAPIC is currently emulated by a Guest MMIO trap to GPA address range: 0xFEE00000 - 0xFEE100000 (1MB). ACRN hypervisor will support APIC-v and Post interrupts in a future release.

vLAPIC provides the same feature as a native LAPIC:

- Mask/Unmask vectors
- Inject virtual vectors (Level or Edge trigger mode) to vCPU
- Notify vIOAPIC of EOI processing
- Provide TSC Timer service
- vLAPIC support CR8 to update TPR
- INIT/STARTUP handling

Virtual IOAPIC

A vIOAPIC is emulated by the hypervisor when the Guest accesses MMIO GPA Range: 0xFEC00000 - 0xFEC01000. The vIOAPIC for the SOS will match the same pin numbers as the native HW IOAPIC. The vIOAPIC for UOS only provides 24 Pins. When a vIOAPIC PIN is asserted, the vIOAPIC calls vLAPIC APIs to inject the vector to the Guest.

Virtual PIC

A vPIC is required for TSC calculation. Normally the UOS boots with a vIOAPIC. A vPIC is a source of external interrupts to the Guest. On every VMExit, the hypervisor checks if there are pending external PIC interrupts.

Virtual Interrupt Injection

The source of virtual interrupts comes from either the Device Module or from assigned devices:

SOS assigned devices As we assigned all devices to SOS directly whenever a devices' physical interrupts come, we inject the corresponding virtual interrupts to SOS via the vLAPIC/vIOAPIC. In this case, the SOS doesn't use the vPIC and does not have emulated devices.

UOS assigned devices Only PCI devices are assigned to UOS, and virtual interrupt injection follows the same way as the SOS. A virtual interrupt injection operation is triggered when a device's physical interrupt is triggered.

UOS emulated devices Device Module (user-land Device Model) is responsible for UOS emulated devices' interrupt lifecycle management. The Device Model knows when an emulated device needs to assert a virtual IOPAIC/PIC Pin or needs to send a virtual MSI vector to the Guest. This logic is entirely handled by the Device Model.

Figure 1.19 shows how the hypervisor handles interrupt processing and pending interrupts (acrn_do_intr_process):

There are many cases where the Guest RFLAG.IF is cleared and interrupts are disabled. The hypervisor will check if the Guest IRQ window is available before injection. NMI is unmasked interrupt injection regardless of existing guest IRQ window status. If the current IRQ windows is not available, hypervisor enables `MSR_IA32_VMX_PROCBASED_CTLS_IRQ_WIN` (PROCBASED_CTRL.bit[2]) and VMEnter directly. The injection will be done on next VMExit once the Guest issues STI (GuestRFLAG.IF=1).

1.4.7 VT-x and VT-d

Since 2006, Intel CPUs have supported hardware assist - VT-x instructions, where the CPU itself traps specific guest instructions and register accesses directly into the VMM without need for binary translation (and modification) of the guest operating system. Guest operating systems can be run natively without modification, although it is common to still install virtualization-aware para-virtualized drivers into the guests to improve functionality. One common example is access to storage via emulated SCSI devices.

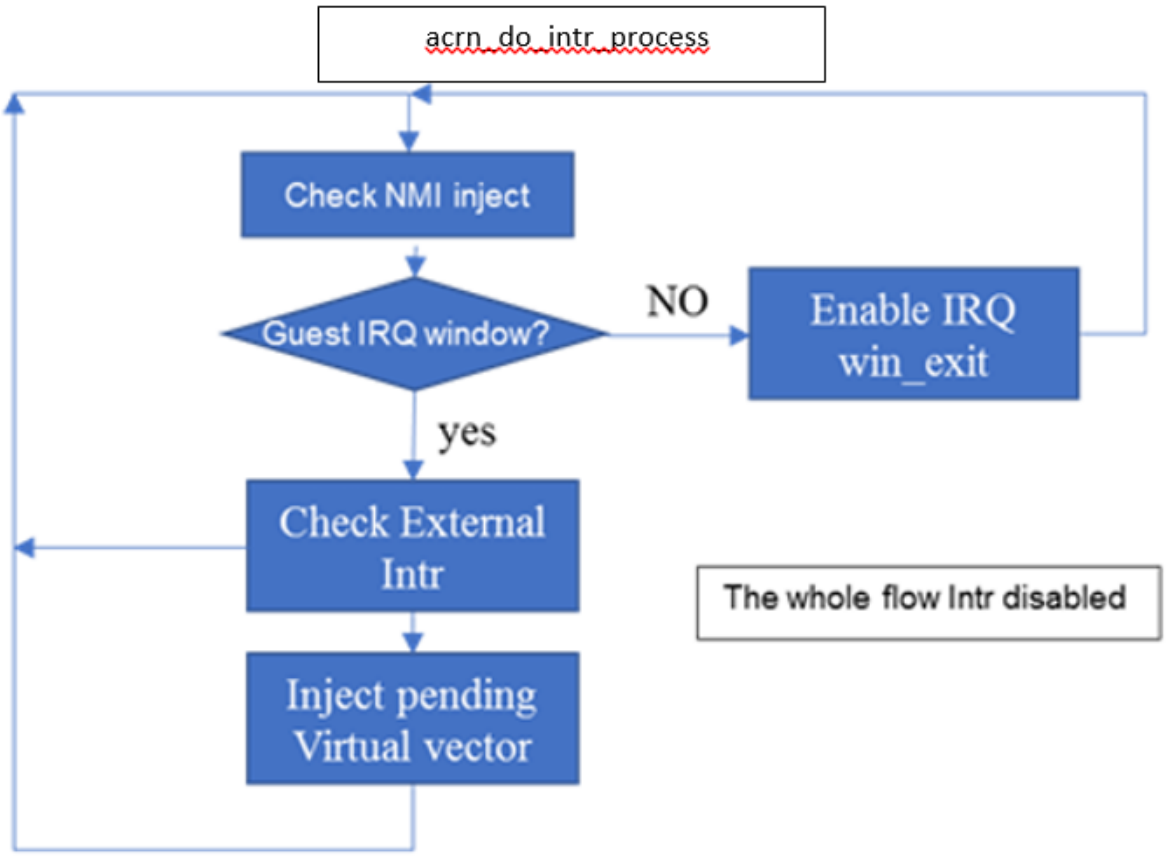


Figure 1.19: Hypervisor Interrupt handler

Intel CPUs and chipsets support various Virtualization Technology (VT) features - such as VT-x and VT-d. Physical events on the platform trigger CPU **VM Exits** (a trap into the VMM) to handle physical events such as physical device interrupts,

In the ACRN hypervisor design, VT-d can be used to do DMA Remapping, such as Address translation and Isolation. Figure 1.20 is an example of address translation:

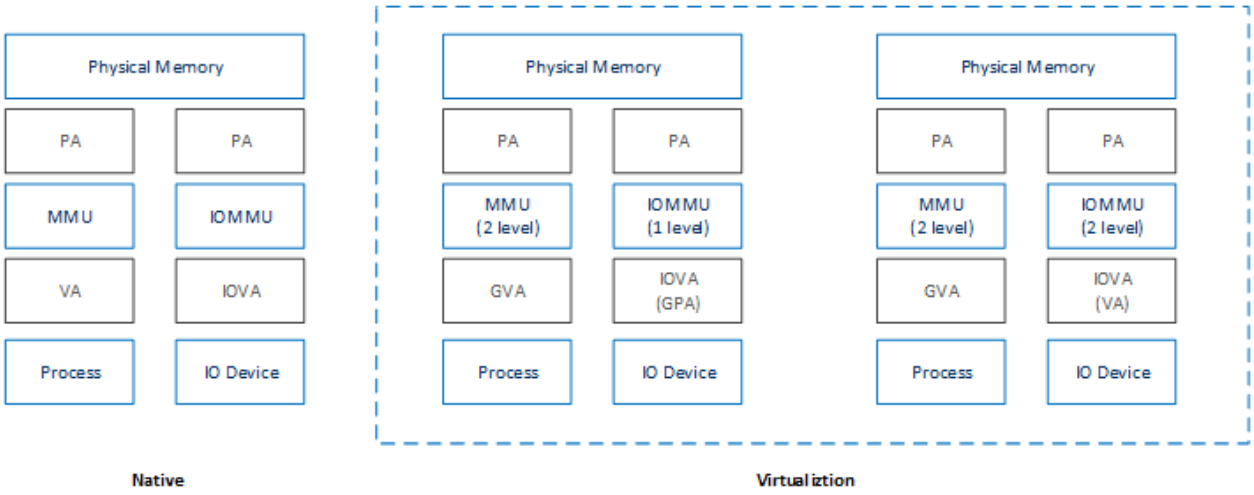


Figure 1.20: DMA address mapping

1.4.8 Hypercall

ACRN hypervisor currently supports less than a dozen *Hypercall APIs* and VHM upcall APIs to support the necessary VM management, IO request distribution and guest memory mappings. The hypervisor and Service OS (SOS) reserve vector 0xF4 for hypervisor notification to the SOS. This upcall is necessary whenever device emulation is required by the SOS. The upcall vector 0xF4 is injected to SOS vCPU0.

Refer to the *API Documentation* documentation for details.

1.4.9 Device emulation

The ACRN Device Model emulates different kinds of platform devices, such as RTC, LPC, UART, PCI device, and Virtio block device. The most important thing about device emulation is to handle the I/O request from different devices. The I/O request could be PIO, MMIO, or PCI CFG SPACE access. For example:

- a CMOS RTC device may access 0x70/0x71 PIO to get the CMOS time,
- a GPU PCI device may access its MMIO or PIO BAR space to complete its frame buffer rendering, or
- the bootloader may access PCI devices' CFG SPACE for BAR reprogramming.

ACRN Device Model injects interrupts/MSIs to its frontend devices when necessary as well, for example, a RTC device needs to get its ALARM interrupt or a PCI device with MSI capability needs to get its MSI. The Data Model also provides a PIRQ routing mechanism for platform devices.

1.4.10 Virtio Devices

This section introduces the Virtio devices supported by ACRN. Currently all the Back-end Virtio drivers are implemented using the Virtio APIs and the FE drivers are re-using Linux standard Front-end Virtio drivers.

Virtio-rnd

The Virtio-rnd entropy device supplies high-quality randomness for guest use. The Virtio device ID of the Virtio-rnd device is 4, and supports one virtqueue of 64 entries (configurable in the source code). No feature bits are defined.

When the FE driver requires random bytes, the BE device places bytes of random data onto the virtqueue.

To launch the Virtio-rnd device, you can use the following command:

```
./acrn-dm -A -m 1168M \
   -s 0:0,hostbridge \
   -s 1,virtio-blk,./uos.img \
   -s 2,virtio-rnd \
   -k bzImage \
   -B "root=/dev/vda rw rootwait noxsave maxcpus=0 nohpet \
       console=hvc0 no_timer_check ignore_loglevel \
       log_buf_len=16M consoleblank=0 tsc=reliable" vm1
```

To verify the result in user OS side, you can use the following command:

```
od /dev/random
```

Virtio-blk

The Virtio-blk device is a simple virtual block device. The FE driver will place read, write, and other requests onto the virtqueue, so that the BE driver can process them accordingly.

The Virtio device ID of the Virtio-blk is 2, and it supports one virtqueue with 64 entries, configurable in the source code. The feature bits supported by the BE device are as follows:

VTBLK_F_SEG_MAX(bit 2) Maximum number of segments in a request is in seg_max.

VTBLK_F_BLK_SIZE(bit 6) block size of disk is in blk_size.

VTBLK_F_FLUSH(bit 9) cache flush command support.

VTBLK_F_TOPOLOGY(bit 10) device exports information on optimal I/O alignment.

To use the Virtio-blk device, use the following command:

```
./acrn-dm -A -m 1168M \
   -s 0:0,hostbridge \
   -s 1,virtio-blk,./uos.img** \
   -k bzImage -B "root=/dev/vda rw rootwait noxsave maxcpus=0 \
      nohpet console=hvc0 no_timer_check ignore_loglevel \
      log_buf_len=16M consoleblank=0 tsc=reliable" vm1
```

To verify the result, you should expect the user OS to boot successfully.

Virtio-net

The Virtio-net device is a virtual Ethernet device. The Virtio device ID of the Virtio-net is 1. The Virtio-net device supports two virtqueues, one for transmitting packets and the other for receiving packets. The FE driver will place empty buffers onto one virtqueue for receiving packets, and enqueue outgoing packets onto the other virtqueue for transmission. Currently the size of each virtqueue is 1000, configurable in the source code.

To access the external network from user OS, a L2 virtual switch should be created in the service OS, and the BE driver is bonded to a tap/tun device linking under the L2 virtual switch. See Figure 1.21:

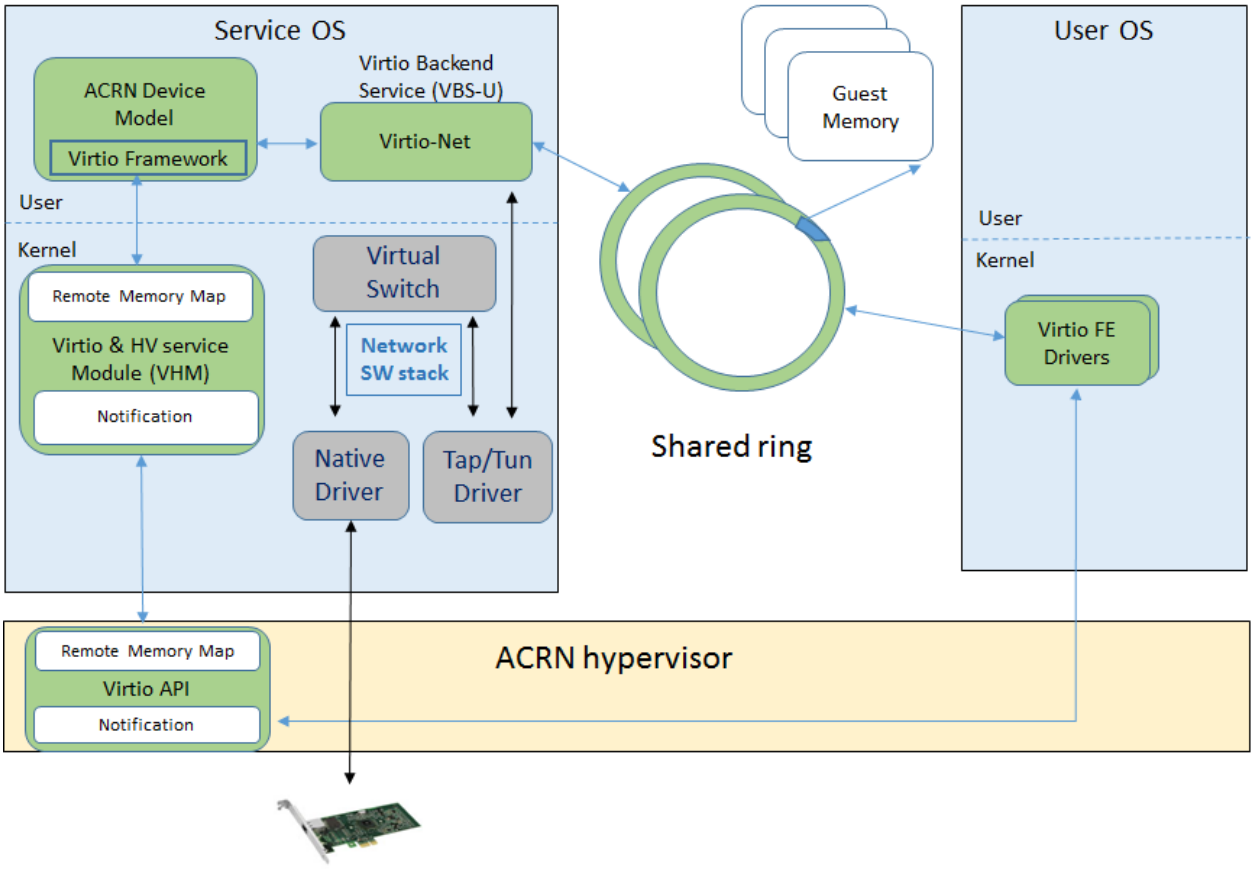


Figure 1.21: Accessing external network from User OS

Currently the feature bits supported by the BE device are:

VIRTIO_NET_F_MAC(bit 5) device has given MAC address.

VIRTIO_NET_F_MRG_RXBUF(bit 15) BE driver can merge receive buffers.

VIRTIO_NET_F_STATUS(bit 16) configuration status field is available.

VIRTIO_F_NOTIFY_ON_EMPTY(bit 24) device will issue an interrupt if it runs out of available descriptors on a virtqueue.

To enable the Virtio-net device, use the following command:

```
./acrn-dm -A -m 1168M \
   -s 0:0,hostbridge \
   -s 1,virtio-blk,./uos.img \
   -s 2,virtio-net,tap0 \
   -k bzImage -B "root=/dev/vda rw rootwait noxsave maxcpus=0 \
      nohpet console=hvc0 no_timer_check ignore_loglevel \
      log_buf_len=16M consoleblank=0 tsc=reliable" vm1
```

To verify the correctness of the device, the external network should be accessible from the user OS.

Virtio-console

The Virtio-console device is a simple device for data input and output. The Virtio device ID of the Virtio-console device is 3. A device could have from one to 16 ports. Each port has a pair of input and output virtqueues used to communicate information between the FE and BE drivers. Currently the size of each virtqueue is 64, configurable in the source code.

Similar to Virtio-net device, the two virtqueues specific to a port are for transmitting virtqueue and receiving virtqueue. The FE driver will place empty buffers onto the receiving virtqueue for incoming data, and enqueue outgoing characters onto transmitting virtqueue.

Currently the feature bits supported by the BE device are:

VTCON_F_SIZE(bit 0) configuration columns and rows are valid.

VTCON_F_MULTIPORT(bit 1) device supports multiple ports, and control virtqueues will be used.

VTCON_F_EMERG_WRITE(bit 2) device supports emergency write.

Virtio-console supports redirecting guest output to various backend devices, including stdio/pty/tty. Users could follow the syntax below to specify which backend to use:

```
virtio-console,[@]stdio\|tty\|pty:portname[=portpath][,
↪[@]stdio\|tty\|pty:portname[=portpath]]
```

For example, to use stdio as a Virtio-console backend, use the following command:

```
./acrn-dm -A -m 1168M \
   -s 0:0,hostbridge \
   -s 1,virtio-blk,./uos.img \
   -s 3,virtio-console,@stdio:stdio\_port \
   -k bzImage -B "root=/dev/vda rw rootwait noxsave maxcpus=0 \
      nohpet console=hvc0 no_timer_check ignore_loglevel \
      log_buf_len=16M consoleblank=0 tsc=reliable" vm1
```

Then user could login into user OS:

```
Ubuntu 17.04 xubuntu hvc0
xubuntu login: root
Password:
```

To use pty as a virtio-console backend, use the following command:

```
./acrn-dm -A -m 1168M \
   -s 0:0,hostbridge \
   -s 1,virtio-blk,./uos.img \
   -s 2,virtio-net,tap0 \
   -s 3,virtio-console,@pty:pty\_port \
   -k ./bzImage -B "root=/dev/vda rw rootwait noxsave maxcpus=0 \
      nohpet console=hvc0 no_timer_check ignore_loglevel \
      log_buf_len=16M consoleblank=0 tsc=reliable" vm1 &
```

When ACRN-DM boots User OS successfully, a similar log will be shown as below:

```
**************************************************************
virt-console backend redirected to /dev/pts/0
**************************************************************
```

You can then use the following command to login the User OS:

```
minicom -D /dev/pts/0
```

or

```
screen /dev/pts/0
```

1.5 How-Tos

Our technical documentation for Project ACRN is being developed right along with the features. Here are some how-to technical notes that help explain how you can use ACRN capabilities.

1.5.1 Technical Notes

Work in Progress

This is a placeholder doc for technical how-to articles in-progress.

1.5.2 Process Notes

ACRN documentation generation

These instructions will walk you through generating the Project ACRN's documentation and publishing it to https://projectacrn.github.io. You can also use these instructions to generate the ARCN documentation on your local system.

Documentation overview

Project ACRN content is written using the reStructuredText markup language (.rst file extension) with Sphinx extensions, and processed using Sphinx to create a formatted stand-alone website. Developers can view this content either

in its raw form as .rst markup files, or you can generate the HTML content and view it with a web browser directly on your workstation.

You can read details about reStructuredText, and Sphinx from their respective websites.

The project's documentation contains the following items:

- ReStructuredText source files used to generate documentation found at the http://projectacrn.github.io website. All of the reStructuredText sources are found in this acrn-documentation repo.
- Doxygen-generated material used to create all API-specific documents found at http://projectacrn.github.io/api/. Clones of the acrn-hypervisor and acrn-devicemodel repos are also used to access the header files for the the public APIs, but more about that later.

The reStructuredText files are processed by the Sphinx documentation system, and make use of the breathe extension for including the doxygen-generated API material.

Set up the documentation working folders

You'll need git installed to get the working folders set up:

- For an Ubuntu development system use:

```
sudo apt-get install git
```

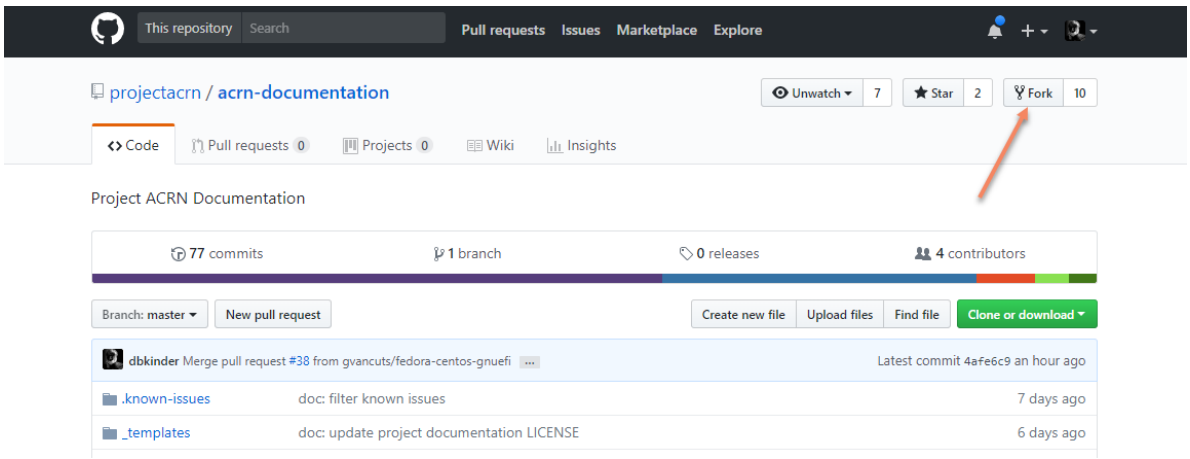
- For a Fedora development system use

```
sudo dnf install git
```

Because we need up-to-date header files to generate API docs, doc generation assumes the three repos are cloned as sibling folders. Here's the recommended folder setup for documentation contributions and generation:

It's best if these folders are ssh clones of your personal forks of the upstream project ACRN repos (though https clones work too):

1. Use your browser to visit https://github.com/projectacrn. One at a time, visit the **acrn-documentation**, **acrn-hypervisor**, and **acrn-devicemodel** repos and do a fork to your personal GitHub account.



2. At a command prompt, create the working folder and clone the three repositories to your local computer:

```
cd ~
mkdir projectacrn && cd projectacrn
git clone  git@github.com:<github-username>/acrn-documentation.git
```

```
git clone  git@github.com:<github-username>/acrn-hypervisor.git
git clone  git@github.com:<github-username>/acrn-devicemodel.git
```

3. For each of the cloned local repos, tell git about the upstream repos:

```
cd acrn-documentation
git remote add upstream git@github.com:projectacrn/acrn-documentation.git
cd ../acrn-hypervisor
git remote add upstream git@github.com:projectacrn/acrn-hypervisor.git
cd ../acrn-devicemodel
git remote add upstream git@github.com:projectacrn/acrn-devicemodel.git
cd ..
```

4. If you haven't do so already, be sure to configure git with your name and email address for the signed-off-by line in your commit messages:

```
git config --global user.name "David Developer"
git config --global user.email "david.developer@company.com"
```

Installing the documentation tools

Our documentation processing has been tested to run with:

- Python 3.6.3
- Doxygen version 1.8.13
- Sphinx version 1.6.7
- Breathe version 4.7.3
- docutils version 0.14
- sphinx_rtd_theme version 0.2.4

Depending on your Linux version, install the needed tools:

- For Ubuntu use:

```
sudo apt-get install doxygen python3-pip python3-wheel make
```

- For Fedora use:

```
sudo dnf install doxygen python3-pip python3-wheel make
```

And for either Linux environment, install the remaining python-based tools:

```
cd ~/projectacrn/acrn-documentation
pip3 install --user -r scripts/requirements.txt
```

And with that you're ready to generate the documentation.

Documentation presentation theme

Sphinx supports easy customization of the generated documentation appearance through the use of themes. Replace the theme files and do another `make htmldocs` and the output layout and style is changed. The `read-the-docs` theme is installed as part of the `requirements.txt` list above.

Running the documentation processors

The acrn-documentation directory has all the .rst source files, extra tools, and Makefile for generating a local copy of the ACRN technical documentation.

```
cd ~/projectacrn/acrn-documentation
make html
```

Depending on your development system, it will take about 15 seconds to collect and generate the HTML content. When done, you can view the HTML output with your browser started at `~/projectacrn/acrn-documentation/_build/html/index.html`

Publishing content

If you have push rights to the projectacrn repo called projectacrn.github.io, you can update the public project documentation found at https://projectacrn.github.io.

You'll need to do a one-time clone the upstream repo:

```
git clone git@github.com:projectacrn/projectacrn.github.io.git
```

Then, after you've verified the generated HTML from `make html` looks good, you can push directly to the publishing site with:

```
make publish
```

This will delete everything in the publishing repo (in case the new version has deleted files) and push a copy of the newly-generated HTML content directly to the GitHub pages publishing repo. The public site at https://projectacrn.github.io will be updated (nearly) immediately so it's best to verify the locally generated html before publishing.

Filtering expected warnings

Alas, there are some known issues with the doxygen/Sphinx/Breathe processing that generates warnings for some constructs, in particular around unnamed structures in nested unions or structs. While these issues are being considered for fixing in Sphinx/Breathe, we've added a post-processing filter on the output of the documentation build process to check for "expected" messages from the generation process output.

The output from the Sphinx build is processed by the python script `scripts/filter-known-issues.py` together with a set of filter configuration files in the `.known-issues/doc` folder. (This filtering is done as part of the `Makefile`.)

If you're contributing components included in the ACRN API documentation and run across these warnings, you can include filtering them out as "expected" warnings by adding a conf file to the `.known-issues/doc` folder, following the example of other conf files found there.

1.6 Release Notes

1.6.1 Version 0.1 release (March 2018)

In March 2018, Intel announced the open source Project ACRN at the Embedded Linux Conference and OpenIoT Summit North America 2018.

ACRN is a flexible, lightweight reference hypervisor, built with real-time and safety-criticality in mind, optimized to streamline embedded development through an open source platform. Check out the *Introduction to Project ACRN* for more information.

The project ACRN reference code can be found on GitHub in https://github.com/projectacrn. It includes the ACRN hypervisor, the ACRN device model, and documentation.

ACRN's key features include:

- Small footprint
- Built with real-time in mind
- Virtualization of embedded IoT device functions
- Safety-critical workload considerations
- Adaptable
- Open Source

This version 0.1 release has the following software components:

- The ACRN Hypervisor
- The ACRN Device Model
- The ACRN Virtio framework
- The ACRN Block & NIC & console Virtio drivers
- The ACRN Virtio Backend Service(VBS) & Virtio and Hypervisor Service Module(VHM).

Version 0.1 features include:

- ACRN hypervisor (Type 1 hypervisor)
- A hybrid VMM architecture implementation
- Multiple User OS supported
- VM management - such as VM start/stop/pause, virtual CPU pause/resume
- CPU virtualization
- Memory virtualization
- I/O emulation
- Virtual interrupt
- VT-x and VT-d support
- Hypercall for guest
- Device emulation
- Device pass-through mechanism
- Device Emulation mechanism
- Virtio console
- Virt-network

1.7 Contribution Guidelines

As an open-source project, we welcome and encourage the community to submit patches directly to project ACRN. In our collaborative open source environment, standards and methods for submitting changes help reduce the chaos that can result from an active development community.

This document explains how to participate in project conversations, log bugs and enhancement requests, and submit patches to the project so your patch will be accepted quickly in the codebase.

1.7.1 Licensing

Licensing is very important to open source projects. It helps ensure the software continues to be available under the terms that the author desired.

Project ACRN uses a BSD-3-Clause license, as found in the license_header in the project's GitHub repo.

A license tells you what rights you have as a developer, as provided by the copyright holder. It is important that the contributor fully understands the licensing rights and agrees to them. Sometimes the copyright holder isn't the contributor, such as when the contributor is doing work on behalf of a company.

1.7.2 Developer Certification of Origin (DCO)

To make a good faith effort to ensure licensing criteria are met, project ACRN requires the Developer Certificate of Origin (DCO) process to be followed.

The DCO is an attestation attached to every contribution made by every developer. In the commit message of the contribution, (described more fully later in this document), the developer simply adds a `Signed-off-by` statement and thereby agrees to the DCO.

When a developer submits a patch, it is a commitment that the contributor has the right to submit the patch per the license. The DCO agreement is shown below and at http://developercertificate.org/.

```
Developer's Certificate of Origin 1.1

By making a contribution to this project, I certify that:

(a) The contribution was created in whole or in part by me and I
    have the right to submit it under the open source license
    indicated in the file; or

(b) The contribution is based upon previous work that, to the
    best of my knowledge, is covered under an appropriate open
    source license and I have the right under that license to
    submit that work with modifications, whether created in whole
    or in part by me, under the same open source license (unless
    I am permitted to submit under a different license), as
    Indicated in the file; or

(c) The contribution was provided directly to me by some other
    person who certified (a), (b) or (c) and I have not modified
    it.

(d) I understand and agree that this project and the contribution
    are public and that a record of the contribution (including
    all personal information I submit with it, including my
    sign-off) is maintained indefinitely and may be redistributed
```

```
   consistent with this project or the open source license(s)
   involved.
```

DCO Sign-Off Methods

The DCO requires that a sign-off message, in the following format, appears on each commit in the pull request:

```
Signed-off-by: Acrnus Jones <acrnusj@gmail.com>
```

The DCO text can either be manually added to your commit body, or you can add either `-s` or `--signoff` to your usual Git commit commands. If you forget to add the sign-off you can also amend a previous commit with the sign-off by running `git commit --amend -s`. If you've pushed your changes to GitHub already you'll need to force push your branch after this with `git push -f`.

1.7.3 Prerequisites

As a contributor, you'll want to be familiar with project ACRN, how to configure, install, and use it as explained on the project ACRN website, and how to set up your development environment as introduced in the project ACRN Getting Started Guide.

You should be familiar with common developer tools such as Git and CMake, and platforms such as GitHub.

If you haven't already done so, you'll need to create a (free) GitHub account on https://github.com and have Git tools available on your development system.

1.7.4 Repository layout

To clone the ACRN hypervisor repository use:

```
git clone https://github.com/projectacrn/acrn-hypervisor
```

To clone the ACRN device model repository use:

```
git clone https://github.com/projectacrn/acrn-devicemodel
```

To clone the ACRN documentation repository use:

```
git clone https://github.com/projectacrn/acrn-documentation
```

The project ACRN directory structure is described in the Hypervisor Primer document. In addition to the ACRN hypervisor and device model itself, you'll also find the sources for technical documentation available from the ACRN documentation site. All of these are available for developers to contribute to and enhance.

1.7.5 Submitting Issues

Issue tracking for ACRN is done using the ACRN-dev mailing list. Before starting on a patch, first read through discussions in the ACRN-dev mailing list to see what's been reported on the issue you'd like to address. Have a conversation on the ACRN-dev mailing list to see what others think of your issue (and proposed solution). You may find others that have encountered the issue you're finding, or that have similar ideas for changes or additions. Send a message to the ACRN-dev mailing list to introduce and discuss your idea with the development community.

It's always a good practice to search for existing or related issues before submitting your own. When you submit an issue (bug or feature request), the triage team will review and comment on the submission, typically within a few business days.

1.7.6 Contribution Tools and Git Setup

Signed-off-by

The name in the commit message `Signed-off-by:` line and your email must match the change authorship information. Make sure your `.gitconfig` is set up correctly by using:

```
git config --global user.name "David Developer"
git config --global user.email "david.developer@company.com"
```

1.7.7 Coding Style

Use these coding guidelines to ensure that your development complies with the project's style and naming conventions.

In general, follow the Linux kernel coding style, with the following exceptions:

- Add braces to every `if` and `else` body, even for single-line code blocks. Use the `--ignore BRACES` flag to make *checkpatch* stop complaining.
- Use spaces instead of tabs to align comments after declarations, as needed.
- Use C89-style single line comments, `/* */`. The C99-style single line comment, `//`, is not allowed.
- Use `/** */` for doxygen comments that need to appear in the documentation.

1.7.8 Contribution Workflow

One general practice we encourage, is to make small, controlled changes. This practice simplifies review, makes merging and rebasing easier, and keeps the change history clear and clean.

When contributing to project ACRN, it is also important you provide as much information as you can about your change, update appropriate documentation, and test your changes thoroughly before submitting.

The general GitHub workflow used by project ACRN developers uses a combination of command line Git commands and browser interaction with GitHub. As it is with Git, there are multiple ways of getting a task done. We'll describe a typical workflow here for the acrn-hypervisor repo that can also be used for the acrn-devicemodel and acrn-documentation repos:

1. Create a Fork of acrn-hypervisor to your personal account on GitHub. (Click on the fork button in the top right corner of the project acrn-hypervisor repo page in GitHub.)
2. On your development computer, clone the fork you just made:

   ```
   git clone https://github.com/<your github id>/acrn-hypervisor
   ```

   This would be a good time to let Git know about the upstream repo too:

   ```
   git remote add upstream https://github.com/projectacrn/acrn-hypervisor.git
   ```

   and verify the remote repos:

```
git remote -v
```

3. Create a topic branch (off of master) for your work (if you're addressing an issue, we suggest including the issue number in the branch name):

```
git checkout master
git checkout -b fix_comment_typo
```

4. Make changes, test locally, change, test, test again, . . .
5. When things look good, start the pull request process by checking which files have not been staged:

```
git status
```

Then add the changed files:

```
git add [file(s) that changed, add -p if you want to be more specific]
```

(or to have all changed files added use):

```
git add -A
```

6. Verify changes to be committed look as you expected:

```
git diff --cached
```

7. Commit your changes to your local repo:

```
git commit -s
```

The `-s` option automatically adds your `Signed-off-by:` to your commit message. Your commit will be rejected without this line that indicates your agreement with the *DCO*. See the *Commit Guidelines* section below for specific guidelines for writing your commit messages.

8. Push your topic branch with your changes to your fork in your personal GitHub account:

```
git push origin fix_comment_typo
```

9. In your web browser, go to your personal forked repo and click on the Compare & pull request button for the branch you just worked on and you want to submit to the upstream repo.
10. Review the pull request changes, and verify that you are opening a pull request for the appropriate branch. The title and message from your commit message should appear as well.
11. GitHub will assign one or more suggested reviewers (based on the CODEOWNERS file in the repo). If you are a project member, you can select additional reviewers now too.
12. Click on the submit button and your pull request is sent and awaits review. Email will be sent as review comments are made, or you can check on your pull request at https://github.com/projectacrn/acrn-hypervisor/pulls.
13. While you're waiting for your pull request to be accepted and merged, you can create another branch to work on another issue. (Be sure to make your new branch off of master and not the previous branch.):

```
git checkout master
git checkout -b fix_another_issue
```

and use the same process described above to work on this new topic branch.

14. If reviewers do request changes to your patch, you can interactively rebase commit(s) to fix review issues. In your development repo, make the needed changes on the branch you made the initial submission:

```
git checkout fix-comment-typo
```

then:

```
git fetch --all
git rebase --ignore-whitespace upstream/master
```

The `--ignore-whitespace` option stops git apply (called by rebase) from changing any whitespace. Continuing:

```
git rebase -i <offending-commit-id>
```

In the interactive rebase editor, replace pick with edit to select a specific commit (if there's more than one in your pull request), or remove the line to delete a commit entirely. Then edit files to fix the issues in the review.

As before, inspect and test your changes. When ready, continue the patch submission:

```
git add [file(s)]
git rebase --continue
```

Update commit comment if needed, and continue:

```
git push --force origin fix_comment_typo
```

By force pushing your update, your original pull request will be updated with your changes so you won't need to resubmit the pull request.

You can follow the same workflow for contributing to acrn-devicemodel or acrn-documentation repos.

1.7.9 Commit Guidelines

Changes are submitted as Git commits. Each commit message must contain:

- A short and descriptive subject line that is less than 72 characters, followed by a blank line. The subject line must include a prefix that identifies the subsystem being changed, followed by a colon, and a short title, for example: `doc: update commit guidelines instructions`. (If you're updating an existing file, you can use `git log <filename>` to see what developers used as the prefix for previous patches of this file.)
- A change description with your logic or reasoning for the changes, followed by a blank line.
- A Signed-off-by line, `Signed-off-by: <name> <email>` typically added automatically by using `git commit -s`
- If the change addresses an issue, include a line of the form:

```
Fixes #<brief description about the reported issue>.
```

All changes and topics sent to GitHub must be well-formed, as described above.

Commit Message Body

When editing the commit message, please briefly explain what your change does and why it's needed. A change summary of `"Fixes stuff"` will be rejected.

Warning: An empty change summary body is not permitted. Even for trivial changes, please include a summary body in the commmit message.

The description body of the commit message must include:

- **what** the change does,
- **why** you chose that approach,
- **what** assumptions were made, and
- **how** you know it works – for example, which tests you ran.

For examples of accepted commit messages, you can refer to the acrn-hypervisor GitHub changelog.

Other Commit Expectations

- Commits must build cleanly when applied on top of each other, thus avoiding breaking bisectability.
- Each commit must address a single identifiable issue and must be logically self-contained. Unrelated changes should be submitted as separate commits.
- You may submit pull request RFCs (requests for comments) to send work proposals, progress snapshots of your work, or to get early feedback on features or changes that will affect multiple areas in the code base.

Identifying Contribution Origin

When adding a new file to the tree, it is important to detail the source of origin on the file, provide attributions, and detail the intended usage. In cases where the file is an original to acrn-hypervisor, the commit message should include the following ("Original" is the assumption if no Origin tag is present):

```
Origin: Original
```

In cases where the file is imported from an external project, the commit message shall contain details regarding the original project, the location of the project, the SHA-id of the origin commit for the file, the intended purpose, and if the file will be maintained by the acrn-hypervisor project, (whether or not project ACRN will contain a localized branch or if it is a downstream copy).

For example, a copy of a locally maintained import:

```
Origin: Contiki OS
License: BSD 3-Clause
URL: http://www.contiki-os.org/
commit: 853207acfdc6549b10eb3e44504b1a75ae1ad63a
Purpose: Introduction of networking stack.
Maintained-by: acrn-hypervisor
```

For example, a copy of an externally maintained import:

```
Origin: Tiny Crypt
License: BSD 3-Clause
URL: https://github.com/01org/tinycrypt
commit: 08ded7f21529c39e5133688ffb93a9d0c94e5c6e
Purpose: Introduction of TinyCrypt
Maintained-by: External
```

1.8 API Documentation

Welcome to Project ACRN API (Application Programing Interface) documentation.

This section contains the API documentation automatically extracted from the code. If you are looking for a specific API, enter it on the search box. The search results display all sections containing information about that API.

1.8.1 Hypercall APIs

This section contains APIs for the hypercall services. Sources for the Device Model are found in the ACRN Hypervisor GitHub repo

group **acrn_hypercall**
Hypercall.

Defines

ACRN_INTR_TYPE_ISA
Interrupt type for *acrn_irqline*: inject interrupt to IOAPIC

ACRN_INTR_TYPE_IOAPIC
Interrupt type for *acrn_irqline*: inject interrupt to both PIC and IOAPIC

GUEST_CFG_OFFSET
The guest config pointer offset.

It's designed to support passing DM config data pointer, based on it, hypervisor would parse then pass DM defined configuration to GUEST VCPU when booting guest VM. the address 0xd0000 here is designed by DM, as it arranged all memory layout below 1M, DM should make sure there is no overlap for the address 0xd0000 usage.

SPACE_SYSTEM_MEMORY
Info The power state data of a VCPU.

SPACE_SYSTEM_IO

SPACE_PCI_CONFIG

SPACE_Embedded_Control

SPACE_SMBUS

SPACE_PLATFORM_COMM

SPACE_FFixedHW

PMCMD_VMID_MASK
Info PM command from DM/VHM.

The command would specify request type(e.g. get px count or data) for specific VM and specific VCPU with specific state number. For Px, PMCMD_STATE_NUM means Px number from 0 to (MAX_PSTATE - 1), For Cx, PMCMD_STATE_NUM means Cx entry index from 1 to MAX_CX_ENTRY.

PMCMD_VCPUID_MASK

PMCMD_STATE_NUM_MASK

PMCMD_TYPE_MASK

PMCMD_VMID_SHIFT

PMCMD_VCPUID_SHIFT

PMCMD_STATE_NUM_SHIFT

Enums

enum pm_cmd_type

Values:

PMCMD_GET_PX_CNT

PMCMD_GET_PX_DATA

PMCMD_GET_CX_CNT

PMCMD_GET_CX_DATA

Functions

int64_t **hcall_get_api_version**(**struct** vm *vm, uint64_t *param*)

Get hypervisor api version.

The function only return api version information when VM is VM0.

Return 0 on success, non-zero on error.

Parameters

- vm: Pointer to VM data structure
- param: guest physical memory address. The api version returned will be copied to this gpa

int64_t **hcall_create_vm**(**struct** vm *vm, uint64_t *param*)

create virtual machine

Create a virtual machine based on parameter, currently there is no limitation for calling times of this function, will add MAX_VM_NUM support later.

Return 0 on success, non-zero on error.

Parameters

- vm: Pointer to VM data structure
- param: guest physical memory address. This gpa points to struct *acrn_create_vm*

int64_t **hcall_destroy_vm**(uint64_t *vmid*)

destroy virtual machine

Destroy a virtual machine, it will pause target VM then shutdown it. The function will return -1 if the target VM does not exist.

Return 0 on success, non-zero on error.

Parameters

- vmid: ID of the VM

int64_t **hcall_resume_vm**(uint64_t *vmid*)
resume virtual machine

Resume a virtual machine, it will schedule target VM's vcpu to run. The function will return -1 if the target VM does not exist or the IOReq buffer page for the VM is not ready.

Return 0 on success, non-zero on error.

Parameters

- `vmid`: ID of the VM

int64_t **hcall_pause_vm**(uint64_t *vmid*)
pause virtual machine

Pause a virtual machine, if the VM is already paused, the function will return 0 directly for success. The function will return -1 if the target VM does not exist.

Return 0 on success, non-zero on error.

Parameters

- `vmid`: ID of the VM

int64_t **hcall_create_vcpu**(**struct** vm **vm*, uint64_t *vmid*, uint64_t *param*)
create vcpu

Create a vcpu based on parameter for a VM, it will allocate vcpu from freed physical cpus, if there is no available pcpu, the function will return -1.

Return 0 on success, non-zero on error.

Parameters

- `vm`: Pointer to VM data structure
- `vmid`: ID of the VM
- `param`: guest physical addressx. This gpa points to struct *acrn_create_vcpu*

int64_t **hcall_assert_irqline**(**struct** vm **vm*, uint64_t *vmid*, uint64_t *param*)
assert IRQ line

Assert a virtual IRQ line for a VM, which could be from ISA or IOAPIC, normally it will active a level IRQ. The function will return -1 if the target VM does not exist.

Return 0 on success, non-zero on error.

Parameters

- `vm`: Pointer to VM data structure
- `vmid`: ID of the VM
- `param`: guest physical address. This gpa points to struct *acrn_irqline*

int64_t **hcall_deassert_irqline**(**struct** vm **vm*, uint64_t *vmid*, uint64_t *param*)
deassert IRQ line

Deassert a virtual IRQ line for a VM, which could be from ISA or IOAPIC, normally it will deactive a level IRQ. The function will return -1 if the target VM does not exist.

Return 0 on success, non-zero on error.

Parameters

- `vm`: Pointer to VM data structure
- `vmid`: ID of the VM
- `param`: guest physical address. This gpa points to struct *acrn_irqline*

int64_t **hcall_pulse_irqline**(**struct** vm *vm, uint64_t *vmid*, uint64_t *param*)
trigger a pulse on IRQ line

Trigger a pulse on a virtual IRQ line for a VM, which could be from ISA or IOAPIC, normally it triggers an edge IRQ. The function will return -1 if the target VM does not exist.

Return 0 on success, non-zero on error.

Parameters

- `vm`: Pointer to VM data structure
- `vmid`: ID of the VM
- `param`: guest physical address. This gpa points to struct *acrn_irqline*

int64_t **hcall_inject_msi**(**struct** vm *vm, uint64_t *vmid*, uint64_t *param*)
inject MSI interrupt

Inject a MSI interrupt for a VM. The function will return -1 if the target VM does not exist.

Return 0 on success, non-zero on error.

Parameters

- `vm`: Pointer to VM data structure
- `vmid`: ID of the VM
- `param`: guest physical address. This gpa points to struct *acrn_msi_entry*

int64_t **hcall_set_ioreq_buffer**(**struct** vm *vm, uint64_t *vmid*, uint64_t *param*)
set ioreq shared buffer

Set the ioreq share buffer for a VM. The function will return -1 if the target VM does not exist.

Return 0 on success, non-zero on error.

Parameters

- `vm`: Pointer to VM data structure
- `vmid`: ID of the VM
- `param`: guest physical address. This gpa points to struct *acrn_set_ioreq_buffer*

int64_t **hcall_notify_req_finish**(uint64_t *vmid*, uint64_t *param*)
notify request done

Notify the requestor VCPU for the completion of an ioreq. The function will return -1 if the target VM does not exist.

Return 0 on success, non-zero on error.

Parameters

- `vmid`: ID of the VM
- `param`: vcpu ID of the requestor

int64_t **hcall_set_vm_memmap**(**struct** vm *vm, uint64_t vmid, uint64_t param)
setup ept memory mapping

Set the ept memory mapping for a VM. The function will return -1 if the target VM does not exist.

Return 0 on success, non-zero on error.

Parameters

- `vm`: Pointer to VM data structure
- `vmid`: ID of the VM
- `param`: guest physical address. This gpa points to struct *vm_set_memmap*

int64_t **hcall_remap_pci_msix**(**struct** vm *vm, uint64_t vmid, uint64_t param)
remap PCI MSI interrupt

Remap a PCI MSI interrupt from a VM's virtual vector to native vector. The function will return -1 if the target VM does not exist.

Return 0 on success, non-zero on error.

Parameters

- `vm`: Pointer to VM data structure
- `vmid`: ID of the VM
- `param`: guest physical address. This gpa points to struct *acrn_vm_pci_msix_remap*

int64_t **hcall_gpa_to_hpa**(**struct** vm *vm, uint64_t vmid, uint64_t param)
translate guest physical address ot host physical address

Translate guest physical address to host physical address for a VM. The function will return -1 if the target VM does not exist.

Return 0 on success, non-zero on error.

Parameters

- `vm`: Pointer to VM data structure
- `vmid`: ID of the VM
- `param`: guest physical address. This gpa points to struct *vm_gpa2hpa*

int64_t **hcall_assign_ptdev**(**struct** vm *vm, uint64_t vmid, uint64_t param)
Assign one passthrough dev to VM.

Return 0 on success, non-zero on error.

Parameters

- `vm`: Pointer to VM data structure
- `vmid`: ID of the VM

- `param`: guest physical address. This gpa points to physical BDF of the assigning ptdev

int64_t **hcall_deassign_ptdev**(**struct** vm *vm, uint64_t *vmid*, uint64_t *param*)
Deassign one passthrough dev from VM.

Return 0 on success, non-zero on error.

Parameters

- `vm`: Pointer to VM data structure
- `vmid`: ID of the VM
- `param`: guest physical address. This gpa points to physical BDF of the deassigning ptdev

int64_t **hcall_set_ptdev_intr_info**(**struct** vm *vm, uint64_t *vmid*, uint64_t *param*)
Set interrupt mapping info of ptdev.

Return 0 on success, non-zero on error.

Parameters

- `vm`: Pointer to VM data structure
- `vmid`: ID of the VM
- `param`: guest physical address. This gpa points to data structure of *hc_ptdev_irq* including intr remapping info

int64_t **hcall_reset_ptdev_intr_info**(**struct** vm *vm, uint64_t *vmid*, uint64_t *param*)
Clear interrupt mapping info of ptdev.

Return 0 on success, non-zero on error.

Parameters

- `vm`: Pointer to VM data structure
- `vmid`: ID of the VM
- `param`: guest physical address. This gpa points to data structure of *hc_ptdev_irq* including intr remapping info

int64_t **hcall_setup_sbuf**(**struct** vm *vm, uint64_t *param*)
Setup a share buffer for a VM.

Return 0 on success, non-zero on error.

Parameters

- `vm`: Pointer to VM data structure
- `param`: guest physical address. This gpa points to struct *sbuf_setup_param*

int64_t **hcall_get_cpu_pm_state**(**struct** vm *vm, uint64_t *cmd*, uint64_t *param*)
Switch VCPU state between Normal/Secure World.

Return 0 on success, non-zero on error.

Parameters

- `vcpu`: Pointer to VCPU data structure

int64_t **hcall_world_switch**(**struct** vcpu *_vcpu_)
Get VCPU Power state.

Return 0 on success, non-zero on error.

Parameters

- `VCPU`: power state data

int64_t **hcall_initialize_trusty**(**struct** vcpu *_vcpu_, uint64_t _param_)
Initialize environment for Trusty-OS on a VCPU.

Return 0 on success, non-zero on error.

Parameters

- `vcpu`: Pointer to VCPU data structure
- `param`: guest physical address. This gpa points to struct *trusty_boot_param*

struct mmio_request
#include <acrn_common.h>

Public Members

uint32_t **direction**

uint32_t **reserved**

int64_t **address**

int64_t **size**

int64_t **value**

struct pio_request
#include <acrn_common.h>

Public Members

uint32_t **direction**

uint32_t **reserved**

int64_t **address**

int64_t **size**

int32_t **value**

struct pci_request
#include <acrn_common.h>

Public Members

uint32_t **direction**

uint32_t **reserved**[3]

int64_t **size**

int32_t **value**

int32_t **bus**

int32_t **dev**

int32_t **func**

int32_t **reg**

struct vhm_request

#include <acrn_common.h>

Public Members

union vhm_request::@0 vhm_request::@1

union vhm_request::@2 vhm_request::reqs

int32_t **valid**

int32_t **client**

int32_t **processed**

struct vhm_request_buffer

#include <acrn_common.h>

Public Members

union vhm_request_buffer::@3 vhm_request_buffer::@4

union

Public Members

struct acrn_create_vm

#include <acrn_common.h> Info to create a VM, the parameter for HC_CREATE_VM hypercall.

Public Members

int32_t **vmid**

created vmid return to VHM. Keep it first field

uint32_t **vcpu_num**

VCPU numbers this VM want to create

uint8_t **GUID**[16]

the GUID of this VM

uint64_t **vm_flag**

uint8_t **reserved**[24]
Reserved for future use

struct acrn_create_vcpu
#include <*acrn_common.h*> Info to create a VCPU.

the parameter for HC_CREATE_VCPU hypercall

Public Members

uint32_t **vcpu_id**
the virtual CPU ID for the VCPU created

uint32_t **pcpu_id**
the physical CPU ID for the VCPU created

struct acrn_set_ioreq_buffer
#include <*acrn_common.h*> Info to set ioreq buffer for a created VM.

the parameter for HC_SET_IOREQ_BUFFER hypercall

Public Members

uint64_t **req_buf**
guest physical address of VM request_buffer

struct acrn_irqline
#include <*acrn_common.h*> Info to assert/deassert/pulse a virtual IRQ line for a VM.

the parameter for HC_ASSERT_IRQLINE/HC_DEASSERT_IRQLINE/HC_PULSE_IRQLINE hypercall

Public Members

uint32_t **intr_type**
interrupt type which could be IOAPIC or ISA

uint32_t **reserved**
reserved for alignment padding

uint64_t **pic_irq**
pic IRQ for ISA type

uint64_t **ioapic_irq**
ioapic IRQ for IOAPIC & ISA TYPE, if -1 then this IRQ will not be injected

struct acrn_msi_entry
#include <*acrn_common.h*> Info to inject a MSI interrupt to VM.

the parameter for HC_INJECT_MSI hypercall

Public Members

uint64_t **msi_addr**
MSI addr[19:12] with dest VCPU ID

uint64_t **msi_data**
MSI data[7:0] with vector

struct acrn_nmi_entry
#include <acrn_common.h> Info to inject a NMI interrupt for a VM.

Public Members

int64_t **vcpu_id**
virtual CPU ID to inject

struct acrn_vm_pci_msix_remap
#include <acrn_common.h> Info to remap pass-through PCI MSI for a VM.

the parameter for HC_VM_PCI_MSIX_REMAP hypercall

Public Members

uint16_t **virt_bdf**
pass-through PCI device virtual BDF#

uint16_t **phys_bdf**
pass-through PCI device physical BDF#

uint16_t **msi_ctl**
pass-through PCI device MSI/MSI-X cap control data

uint16_t **reserved**
reserved for alignment padding

uint64_t **msi_addr**
pass-through PCI device MSI address to remap, which will return the caller after remapping

uint32_t **msi_data**
pass-through PCI device MSI data to remap, which will return the caller after remapping

int32_t **msix**
pass-through PCI device is MSI or MSI-X 0 - MSI, 1 - MSI-X

int32_t **msix_entry_index**
if the pass-through PCI device is MSI-X, this field contains the MSI-X entry table index

uint32_t **vector_ctl**
if the pass-through PCI device is MSI-X, this field contains Vector Control for MSI-X Entry, field defined in MSI-X spec

struct acrn_register
#include <acrn_common.h>

Public Members

uint8_t **space_id**

uint8_t **bit_width**

uint8_t **bit_offset**

uint8_t **access_size**

uint64_t **address**

struct cpu_cx_data
#include <acrn_common.h>

Public Members

struct *acrn_register* **cx_reg**

uint8_t **type**

uint32_t **latency**

uint64_t **power**

struct cpu_px_data
#include <acrn_common.h>

Public Members

uint64_t **core_frequency**

uint64_t **power**

uint64_t **transition_latency**

uint64_t **bus_master_latency**

uint64_t **control**

uint64_t **status**

struct vm_set_memmap
#include <acrn_hv_defs.h> Info to set ept mapping.

the parameter for HC_VM_SET_MEMMAP hypercall

Public Members

uint32_t **type**
map type: MAP_MEM, MAP_MMIO or MAP_UNMAP

uint32_t **reserved**
reserved for alignment padding

uint64_t **remote_gpa**
guest physical address to map

uint64_t **vm0_gpa**
VM0's guest physcial address which remote gpa will be mapped to

uint64_t **length**
length of the map range

uint32_t **prot**
memory attributes: memory type + RWX access right

struct sbuf_setup_param
#include <acrn_hv_defs.h> Setup parameter for share buffer, used for HC_SETUP_SBUF hypercall

Public Members

uint32_t **pcpu_id**
sbuf physical cpu id

uint32_t **sbuf_id**
sbuf id

uint64_t **gpa**
sbuf's guest physical address

struct vm_gpa2hpa
#include <*acrn_hv_defs.h*> Gpa to hpa translation parameter, used for HC_VM_GPA2HPA hypercall

Public Members

uint64_t **gpa**
gpa to do translation

uint64_t **hpa**
hpa to return after translation

struct hc_ptdev_irq
#include <*acrn_hv_defs.h*> Intr mapping info per ptdev, the parameter for HC_SET_PTDEV_INTR_INFO hypercall

Public Members

uint32_t **type**
irq mapping type: INTX or MSI

uint16_t **virt_bdf**
virtual BDF of the ptdev

uint16_t **phys_bdf**
physical BDF of the ptdev

union hc_ptdev_irq::@5 hc_ptdev_irq::is

union

Public Members

INTX remapping info

MSIx remapping info

struct
INTX remapping info

Public Members

virtual IOAPIC/PIC pin

physical IOAPIC pin

is virtual pin from PIC

struct
MSIx remapping info

Public Members

vector count of MSI/MSIX

struct hc_api_version
#include <acrn_hv_defs.h> Hypervisor api version info, return it for HC_GET_API_VERSION hyercall

Public Members

uint32_t **major_version**
hypervisor api major version

uint32_t **minor_version**
hypervisor api minor version

struct trusty_boot_param
#include <acrn_hv_defs.h> Trusty boot params, used for HC_INITIALIZE_TRUSTY

Public Members

uint32_t **size_of_this_struct**
sizeof this structure

uint32_t **version**
version of this structure

uint32_t **base_addr**
trusty runtime memory base address

uint32_t **entry_point**
trusty entry point

uint32_t **mem_size**
trusty runtime memory size

1.8.2 Device Model APIs

This section contains APIs for the SOS Device Model services. Sources for the Device Model are found in the ACRN Device Model GitHub repo

group **acrn_virtio**
virtio API

Defines

VRING_ALIGN

VRING_DESC_F_NEXT

VRING_DESC_F_WRITE

VRING_DESC_F_INDIRECT

VRING_AVAIL_F_NO_INTERRUPT

VRING_USED_F_NO_NOTIFY

VRING_PAGE_BITS

VIRTIO_TYPE_NET

VIRTIO_TYPE_BLOCK

VIRTIO_TYPE_CONSOLE

VIRTIO_TYPE_ENTROPY

VIRTIO_TYPE_BALLOON

VIRTIO_TYPE_IOMEMORY

VIRTIO_TYPE_RPMSG

VIRTIO_TYPE_SCSI

VIRTIO_TYPE_9P

VIRTIO_TYPE_INPUT

VIRTIO_TYPE_RPMB

VIRTIO_TYPE_HECI

VIRTIO_TYPE_AUDIO

VIRTIO_TYPE_IPU

VIRTIO_TYPE_TSN

VIRTIO_TYPE_HYPERDMABUF

VIRTIO_TYPE_HDCP

VIRTIO_TYPE_COREU

INTEL_VENDOR_ID

VIRTIO_VENDOR

VIRTIO_DEV_NET

VIRTIO_DEV_BLOCK

VIRTIO_DEV_CONSOLE

VIRTIO_DEV_RANDOM

VIRTIO_DEV_RPMB

VIRTIO_DEV_HECI

VIRTIO_DEV_AUDIO

VIRTIO_DEV_IPU

VIRTIO_DEV_TSN

VIRTIO_DEV_HYPERDMABUF

VIRTIO_DEV_HDCP

VIRTIO_DEV_COREU

VIRTIO_CR_HOSTCAP

VIRTIO_CR_GUESTCAP

VIRTIO_CR_PFN

VIRTIO_CR_QNUM

VIRTIO_CR_QSEL

VIRTIO_CR_QNOTIFY

VIRTIO_CR_STATUS

VIRTIO_CR_ISR

VIRTIO_CR_CFGVEC

VIRTIO_CR_QVEC

VIRTIO_CR_CFG0

VIRTIO_CR_CFG1

VIRTIO_CR_MSIX

VIRTIO_CR_STATUS_ACK

VIRTIO_CR_STATUS_DRIVER

VIRTIO_CR_STATUS_DRIVER_OK

VIRTIO_CR_STATUS_FEATURES_OK

VIRTIO_CR_STATUS_NEEDS_RESET

VIRTIO_CR_STATUS_FAILED

VIRTIO_CR_ISR_QUEUES

VIRTIO_CR_ISR_CONF_CHANGED

VIRTIO_MSI_NO_VECTOR

VIRTIO_F_NOTIFY_ON_EMPTY

VIRTIO_RING_F_INDIRECT_DESC

VIRTIO_RING_F_EVENT_IDX

VIRTIO_F_VERSION_1

VIRTIO_USE_MSIX

VIRTIO_EVENT_IDX

VIRTIO_BROKED

VIRTIO_LEGACY_PIO_BAR_IDX

VIRTIO_MODERN_PIO_BAR_IDX

VIRTIO_MODERN_MMIO_BAR_IDX

VIRTIO_CAP_COMMON_OFFSET

VIRTIO_CAP_COMMON_SIZE

VIRTIO_CAP_ISR_OFFSET

VIRTIO_CAP_ISR_SIZE

VIRTIO_CAP_DEVICE_OFFSET

VIRTIO_CAP_DEVICE_SIZE

VIRTIO_CAP_NOTIFY_OFFSET

VIRTIO_CAP_NOTIFY_SIZE

VIRTIO_MODERN_MEM_BAR_SIZE

VIRTIO_MODERN_NOTIFY_OFF_MULT

VIRTIO_PCI_CAP_COMMON_CFG

VIRTIO_PCI_CAP_NOTIFY_CFG

VIRTIO_PCI_CAP_ISR_CFG

VIRTIO_PCI_CAP_DEVICE_CFG

VIRTIO_PCI_CAP_PCI_CFG

VIRTIO_COMMON_DFSELECT

VIRTIO_COMMON_DF

VIRTIO_COMMON_GFSELECT

VIRTIO_COMMON_GF

VIRTIO_COMMON_MSIX

VIRTIO_COMMON_NUMQ

VIRTIO_COMMON_STATUS

VIRTIO_COMMON_CFGGENERATION

VIRTIO_COMMON_Q_SELECT

VIRTIO_COMMON_Q_SIZE

VIRTIO_COMMON_Q_MSIX

VIRTIO_COMMON_Q_ENABLE

VIRTIO_COMMON_Q_NOFF

VIRTIO_COMMON_Q_DESCLO

VIRTIO_COMMON_Q_DESCHI

VIRTIO_COMMON_Q_AVAILLO

VIRTIO_COMMON_Q_AVAILHI

VIRTIO_COMMON_Q_USEDLO

VIRTIO_COMMON_Q_USEDHI

VIRTIO_BASE_LOCK(vb)

VIRTIO_BASE_UNLOCK(vb)

VQ_ALLOC

VQ_BROKED

VQ_AVAIL_EVENT_IDX(vq)

VQ_USED_EVENT_IDX(vq)

Functions

static size_t **vring_size**(u_int *qsz*)
Calculate size of a virtual ring, this interface is only valid for legacy virtio.

Return size of a certain virtqueue, in bytes.

Parameters

- qsz: Size of raw data in a certain virtqueue.

static int **vq_ring_ready**(**struct** *virtio_vq_info* **vq*)
Is this ring ready for I/O?

Return 0 on not ready and 1 on ready.

Parameters

- vq: Pointer to struct *virtio_vq_info*.

static int **vq_has_descs**(**struct** *virtio_vq_info* **vq*)
Are there "available" descriptors?

This does not count how many, just returns 1 if there is any.

Return 0 on no available and 1 on available.

Parameters

- vq: Pointer to struct *virtio_vq_info*.

static void **vq_interrupt**(**struct** *virtio_base* **vb*, **struct** *virtio_vq_info* **vq*)
Deliver an interrupt to guest on the given virtqueue.

The interrupt could be MSI-X or a generic MSI interrupt.

Return NULL

Parameters

- vb: Pointer to struct *virtio_base*.
- vq: Pointer to struct *virtio_vq_info*.

static void **virtio_config_changed**(**struct** *virtio_base* **vb*)
Deliver an config changed interrupt to guest.

MSI-X or a generic MSI interrupt with config changed event.

Return NULL.

Parameters

- `vb`: Pointer to struct *virtio_base*.

void **virtio_linkup**(**struct** *virtio_base* **vb*, **struct** *virtio_ops* **vo*, void **pci_virtio_dev*, **struct** pci_vdev **dev*, **struct** *virtio_vq_info* **queues*)

Link a *virtio_base* to its constants, the virtio device, and the PCI emulation.

Return NULL

Parameters

- `vb`: Pointer to struct *virtio_base*.
- `vo`: Pointer to struct *virtio_ops*.
- `pci_virtio_dev`: Pointer to instance of certain virtio device.
- `dev`: Pointer to struct pci_vdev which emulates a PCI device.
- `queues`: Pointer to struct *virtio_vq_info*, normally an array.

int **virtio_interrupt_init**(**struct** *virtio_base* **vb*, int *use_msix*)

Initialize MSI-X vector capabilities if we're to use MSI-X, or MSI capabilities if not.

Wrapper function for virtio_intr_init() for cases we directly use BAR 1 for MSI-X capabilities.

Return 0 on success and non-zero on fail.

Parameters

- `vb`: Pointer to struct *virtio_base*.
- `use_msix`: If using MSI-X.

int **virtio_intr_init**(**struct** *virtio_base* **vb*, int *barnum*, int *use_msix*)

Initialize MSI-X vector capabilities if we're to use MSI-X, or MSI capabilities if not.

Return 0 on success and non-zero on fail.

Parameters

- `vb`: Pointer to struct *virtio_base*.
- `barnum`: Which BAR[0..5] to use.
- `use_msix`: If using MSI-X.

void **virtio_reset_dev**(**struct** *virtio_base* **vb*)

Reset device (device-wide).

This erases all queues, i.e., all the queues become invalid. But we don't wipe out the internal pointers, by just clearing the VQ_ALLOC flag.

Return N/A

Parameters

- `vb`: Pointer to struct *virtio_base*.

void **virtio_set_io_bar**(**struct** *virtio_base* **vb*, int *barnum*)

Set I/O BAR (usually 0) to map PCI config registers.

Return N/A

Parameters

- `vb`: Pointer to struct *virtio_base*.
- `barnum`: Which BAR[0..5] to use.

int **vq_getchain**(**struct** *virtio_vq_info* *vq, uint16_t *pidx, **struct** iovec *iov, int n_iov, uint16_t *flags)

Walk through the chain of descriptors involved in a request and put them into a given iov[] array.

Return number of descriptors.

Parameters

- `vq`: Pointer to struct *virtio_vq_info*.
- `pidx`: Pointer to available ring position.
- `iov`: Pointer to iov[] array prepared by caller.
- `n_iov`: Size of iov[] array.
- `flags`: Pointer to a uint16_t array which will contain flag of each descriptor.

void **vq_retchain**(**struct** *virtio_vq_info* *vq)

Return the currently-first request chain back to the available ring.

Return N/A

Parameters

- `vq`: Pointer to struct *virtio_vq_info*.

void **vq_relchain**(**struct** *virtio_vq_info* *vq, uint16_t idx, uint32_t iolen)

Return specified request chain to the guest, setting its I/O length to the provided value.

Return N/A

Parameters

- `vq`: Pointer to struct *virtio_vq_info*.
- `idx`: Pointer to available ring position, returned by vq_getchain().
- `iolen`: Number of data bytes to be returned to frontend.

void **vq_endchains**(**struct** *virtio_vq_info* *vq, int used_all_avail)

Driver has finished processing "available" chains and calling vq_relchain on each one.

If driver used all the available chains, used_all_avail need to be set to 1.

Return N/A

Parameters

- `vq`: Pointer to struct *virtio_vq_info*.
- `used_all_avail`: Flag indicating if driver used all available chains.

uint64_t **virtio_pci_read**(**struct** vmctx *ctx, int *vcpu*, **struct** pci_vdev *dev, int *baridx*, uint64_t *offset*, int *size*)

Handle PCI configuration space reads.

Handle virtio standard register reads, and dispatch other reads to actual virtio device driver.

Return register value.

Parameters

- `ctx`: Pointer to struct vmctx representing VM context.
- `vcpu`: VCPU ID.
- `dev`: Pointer to struct pci_vdev which emulates a PCI device.
- `baridx`: Which BAR[0..5] to use.
- `offset`: Register offset in bytes within a BAR region.
- `size`: Access range in bytes.

void **virtio_pci_write**(**struct** vmctx *ctx, int *vcpu*, **struct** pci_vdev *dev, int *baridx*, uint64_t *offset*, int *size*, uint64_t *value*)

Handle PCI configuration space writes.

Handle virtio standard register writes, and dispatch other writes to actual virtio device driver.

Return N/A

Parameters

- `ctx`: Pointer to struct vmctx representing VM context.
- `vcpu`: VCPU ID.
- `dev`: Pointer to struct pci_vdev which emulates a PCI device.
- `baridx`: Which BAR[0..5] to use.
- `offset`: Register offset in bytes within a BAR region.
- `size`: Access range in bytes.
- `value`: Data value to be written into register.

void **virtio_dev_error**(**struct** *virtio_base* *base)

Indicate the device has experienced an error.

This is called when the device has experienced an error from which it cannot re-cover. DEVICE_NEEDS_RESET is set to the device status register and a config change intr is sent to the guest driver.

Return N/A

Parameters

- `base`: Pointer to struct *virtio_base*.

int **virtio_set_modern_bar**(**struct** *virtio_base* *base, bool *use_notify_pio*)

Set modern BAR (usually 4) to map PCI config registers.

Set modern MMIO BAR (usually 4) to map virtio 1.0 capabilities and optional set modern PIO BAR (usually 2) to map notify capability. This interface is only valid for modern virtio.

Return 0 on success and non-zero on fail.

Parameters

- `base`: Pointer to struct *virtio_base*.
- `use_notify_pio`: Whether use pio for notify capability.

struct virtio_desc
#include <virtio.h>

Public Members

uint64_t **addr**

uint32_t **len**

uint16_t **flags**

uint16_t **next**

struct virtio_used
#include <virtio.h>

Public Members

uint32_t **idx**

uint32_t **tlen**

struct vring_avail
#include <virtio.h>

Public Members

uint16_t **flags**

uint16_t **idx**

uint16_t **ring**[]

struct vring_used
#include <virtio.h>

Public Members

uint16_t **flags**

uint16_t **idx**

struct *virtio_used* **ring**[]

struct virtio_pci_common_cfg
#include <virtio.h>

Public Members

uint32_t **device_feature_select**

uint32_t **device_feature**

uint32_t **guest_feature_select**

uint32_t **guest_feature**

uint16_t **msix_config**

uint16_t **num_queues**

uint8_t **device_status**

uint8_t **config_generation**

uint16_t **queue_select**

uint16_t **queue_size**

uint16_t **queue_msix_vector**

uint16_t **queue_enable**

uint16_t **queue_notify_off**

uint32_t **queue_desc_lo**

uint32_t **queue_desc_hi**

uint32_t **queue_avail_lo**

uint32_t **queue_avail_hi**

uint32_t **queue_used_lo**

uint32_t **queue_used_hi**

struct virtio_pci_cap

#include <virtio.h>

Public Members

uint8_t **cap_vndr**

uint8_t **cap_next**

uint8_t **cap_len**

uint8_t **cfg_type**

uint8_t **bar**

uint8_t **padding**[3]

uint32_t **offset**

uint32_t **length**

struct virtio_pci_notify_cap

#include <virtio.h>

Public Members

struct *virtio_pci_cap* **cap**

uint32_t **notify_off_multiplier**

struct virtio_pci_cfg_cap
#include <virtio.h>

Public Members

struct *virtio_pci_cap* **cap**

uint8_t **pci_cfg_data**[4]

struct virtio_base
#include <virtio.h> Base component to any virtio device.

Public Members

struct *virtio_ops* ***vops**
virtio operations

int **flags**
VIRTIO_* flags from above

pthread_mutex_t ***mtx**
POSIX mutex, if any

struct pci_vdev ***dev**
PCI device instance

uint64_t **negotiated_caps**
negotiated capabilities

struct *virtio_vq_info* ***queues**
one per nvq

int **curq**
current queue

uint8_t **status**
value from last status write

uint8_t **isr**
ISR flags, if not MSI-X

uint16_t **msix_cfg_idx**
MSI-X vector for config event

uint32_t **legacy_pio_bar_idx**
index of legacy pio bar

uint32_t **modern_pio_bar_idx**
index of modern pio bar

uint32_t **modern_mmio_bar_idx**
index of modern mmio bar

uint8_t **config_generation**
configuration generation

uint32_t **device_feature_select**
current selected device feature

uint32_t **driver_feature_select**
current selected guest feature

struct virtio_ops
#include <virtio.h> Virtio specific operation functions for this type of virtio device.

Public Members

const char ***name**
name of driver (for diagnostics)

int **nvq**
number of virtual queues

size_t **cfgsize**
size of dev-specific config regs

void (***reset**)(void *)
called on virtual device reset

void (***qnotify**)(void *, **struct** *virtio_vq_info* *)
called on QNOTIFY if no VQ notify

int (***cfgread**)(void *, int, int, uint32_t *)
to read config regs

int (***cfgwrite**)(void *, int, int, uint32_t)
to write config regs

void (***apply_features**)(void *, uint64_t)
to apply negotiated features

void (***set_status**)(void *, uint64_t)
called to set device status

uint64_t **hv_caps**
hypervisor-provided capabilities

struct virtio_vq_info
#include <virtio.h> Virtqueue data structure.

Data structure allocated (statically) per virtual queue.

Drivers may change qsize after a reset. When the guest OS requests a device reset, the hypervisor first calls vb->vo->reset(); then the data structure below is reinitialized (for each virtqueue: vb->vo->nvq).

The remaining fields should only be fussed-with by the generic code.

Note: the addresses of desc, avail, and vq_used are all computable from each other, but it's a lot simpler if we just keep a pointer to each one. The event indices are similarly (but more easily) computable, and this time we'll compute them: they're just XX_ring[N].

Public Members

uint16_t **qsize**
size of this queue (a power of 2)

void (***notify**) (void *, **struct** *virtio_vq_info* *)
called instead of notify, if not NULL

struct *virtio_base* ***base**
backpointer to *virtio_base*

uint16_t **num**
the num'th queue in the *virtio_base*

uint16_t **flags**
flags (see above)

uint16_t **last_avail**
a recent value of avail->idx

uint16_t **save_used**
saved used->idx; see vq_endchains

uint16_t **msix_idx**
MSI-X index, or VIRTIO_MSI_NO_VECTOR

uint32_t **pfn**
PFN of virt queue (not shifted!)

volatile struct *virtio_desc* ***desc**
descriptor array

volatile struct *vring_avail* ***avail**
the "avail" ring

volatile struct *vring_used* ***used**
the "used" ring

uint32_t **gpa_desc**[2]
gpa of descriptors

uint32_t **gpa_avail**[2]
gpa of avail_ring

uint32_t **gpa_used**[2]
gpa of used_ring

bool **enabled**
whether the virtqueue is enabled

CHAPTER

TWO

INDICES AND TABLES

- glossary
- genindex

INDEX

I

M

P

S

T

V