Qualcomm Technologies, Inc.

Qualcomm Linux Kernel Guide

80-70018-3 AC

April 10, 2025

Contents

1 Kernel overview

Qualcomm® Linux® facilitates porting and customizing the Linux kernel to devices with Qualcomm hardware SoCs. It uses a long-term support (LTS) Linux kernel (6.6.x) and the GNU compiler collection (GCC) toolchain that's compatible with the Yocto project.

The kernel is integrated and aligned to support Linux through Qualcomm Linux meta layers. Qualcomm Linux provides a way to build and customize the kernel and other packages to define a distribution according to the needs of a device or application.

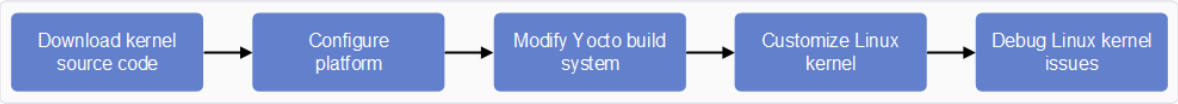

Figure: Qualcomm Linux kernel workflow

The Qualcomm Linux distribution supports the following hardware SoCs and development kits:

Table: Supported hardware SoCs and development kits

Hardware SoCs	Development kit
QCS6490	• QCS6490 RB3 Gen 2 Core Development Kit • QCS6490 RB3 Gen 2 optional mezzanine boards
QCS5430	QCS5430 RB3 Gen 2 Core Development Kit
QCS9075	QCS9075 Core Development Kit
QCS8275	QCS8275 IQ8 Beta Evaluation Kit (EVK)

Note: See the Hardware SoCs that are supported on Qualcomm Linux.

2 Getting started with Qualcomm Linux kernel

This information explains how to download, configure, and develop the Qualcomm® Linux kernel for the Qualcomm Linux development kit and its associated components to build and flash system images.

Before you begin downloading and configuring the Qualcomm Linux kernel, set up your host computer as described in the Qualcomm Linux Build Guide, which also provides information about the common Qualcomm Linux kernel workflows.

2.1 Access Qualcomm Linux kernel source code

To access the source code for the Qualcomm Linux kernel, see the Qualcomm Linux Build Guide and get all the necessary Qualcomm Linux meta layers.

Qualcomm Linux uses the LTS Linux kernel (6.6.x) and supports two software variants:

- Base board support package (BSP)
- Custom BSP

Note:

- **Base BSP** variant uses the upstream LTS Linux kernel (6.6.x) with a limited set of downstream patches. Kernel sources are hosted at kernel/git/stable/linux.git.
- **Custom BSP** variant uses a customized LTS Linux kernel (6.6.x) and is hosted at the Qualcomm repository on CodeLinaro. Kernel sources are hosted at kernel.qclinux.1.0.r1-rel.

Table: Kernel recipe path of the variants

Variant	Kernel recipe path
Base BSP	`recipes-kernel/linux/linux-qcom-base_6.6.bb`

Variant	Kernel recipe path
Custom BSP	`recipes-kernel/linux/linux-qcom-custom_6.6.bb`

Note: The kernel recipes for base BSP and custom BSP are present in the `meta-qcom-hwe` layer.

See the `recipes-kernel/linux/linux-qcom-base_6.6.bb` or `recipes-kernel/linux/linux-qcom-custom_6.6.bb` file to know the Git uniform resource identifier (URI) to download the kernel source code.

For more information about Qualcomm Linux layers, see Qualcomm Linux metadata layers overview.

Note: Yocto recipe refers to Qualcomm Linux kernel sources that are publicly hosted at CodeLinaro.

The following examples show the Qualcomm Linux kernel recipe source URI information for the base BSP and custom BSP:

Base BSP

The following is the source URI information for the base BSP variant:

```
SECTION = "kernel"

SUMMARY = "Linux kernel for QCOM devices"
DESCRIPTION = "Recipe to build Linux kernel from 6.6 LTS branch"

LICENSE = "GPL-2.0-only"
LIC_FILES_CHKSUM = "file://COPYING;
md5=6bc538ed5bd9a7fc9398086aedcd7e46"

inherit kernel

COMPATIBLE_MACHINE = "(qcom)"

SRC_URI = " git://git.kernel.org/pub/scm/linux/kernel/git/stable/
linux.git;protocol=https;branch=linux-6.6.y \
           file://qcom.cfg \
"
```

Custom BSP

The following is the source URI information for the custom BSP variant:

```
SECTION = "kernel"

SUMMARY = "Linux kernel for QCOM devices"
DESCRIPTION = "Recipe to build Linux kernel"

LICENSE = "GPLv2.0-with-linux-syscall-note"
LIC_FILES_CHKSUM = "file://COPYING;
md5=6bc538ed5bd9a7fc9398086aedcd7e46"

inherit kernel

COMPATIBLE_MACHINE = "(qcom)"

SRC_URI = "git://git.codelinaro.org/clo/la/kernel/qcom.git;
protocol=https;rev=350dfd604d2ffbe0cac99bf3459b49114aad11f4;
branch=kernel.qclinux.1.0.r1-rel \
  file://QCLINUX-arm64-dts-qcom-sa8775p-ride-add-board-id-
and.patch \
  ${@bb.utils.contains('DISTRO_FEATURES', 'selinux', ' file:
//selinux.cfg', '', d)} \
  ${@bb.utils.contains('DISTRO_FEATURES', 'selinux', ' file:
//selinux_debug.cfg', '', d)} \
  "
```

2.2 Identify supported Qualcomm machines

The Qualcomm machine configuration files are present in the `meta-qcom-hwe/conf/machine/` directory.

Machine configuration files follow a `<SoC>-<board>-<variant>.conf` naming convention, where:

- `SoC` is the system-on-chip (SoC) name
- `board` is the type of board for which the software is built
- `variant` is the type of product

For example:

In the `qcs6490-rb3gen2-core-kit.conf` filename,

- `qcs6490` is the SoC
- `rb3gen2` is the board

- `core-kit` is the type of product

The same naming convention is followed in platform-specific device tree files.

The following example shows the list of supported machines that are configured for Yocto:

```
ls -l ./LE.QCLINUX.1.0/layers/meta-qcom-hwe/conf/machine/*.conf

meta-qcom-hwe/conf/machine/qcs6490-rb3gen2-core-kit.conf
meta-qcom-hwe/conf/machine/qcs6490-rb3gen2-vision-kit.conf
meta-qcom-hwe/conf/machine/qcs9075-ride-sx.conf
```

The following example shows the machine configuration selection for the development board with Qualcomm chipset:

```
less meta-qcom-hwe/conf/machine/<SoC>-<board>-<variant>.conf

#@TYPE: Machine
#@NAME: <Machine name same as the machine conf file or ``<SoC>-
<board>-<variant>`` >
#@DESCRIPTION: Machine configuration for the development board, with
Qualcomm qcs6490

require conf/machine/include/qcom-<SoC>.inc
```

The following example shows the kernel configuration selection for base and custom BSP variants:

```
less meta-qcom-hwe/conf/machine/include/qcom-base.inc

# Provider for linux kernel
# qcom-base-bsp uses 'linux-qcom-base' as kernel
PREFERRED_PROVIDER_virtual/kernel ?= "linux-qcom-custom"
PREFERRED_PROVIDER_virtual/kernel:qcom-base-bsp ?= "linux-qcom-base"
```

Note:

- `linux-qcom-base` is selected when the base BSP variant is compiled.
- `linux-qcom-custom` is selected when a custom BSP is compiled.

2.3 Build Yocto image recipes and kernel configuration

You can access the Qualcomm Linux image recipes to modify the kernel configurations. The following table lists the supported `meta-qcom-hwe` images:

Table: Qualcomm Linux supported image recipes

Image name	Description
`qcom-console-image`	This is a boot-to-shell image with the package group to bring in all basic packages.
`qcom-multimedia-image`	This image recipe includes recipes for multimedia software components, such as audio, Bluetooth®, camera, computer vision, display, and video.
`qcom-multimedia-test-image`	This image recipe includes tests.

After downloading the Qualcomm Linux code, run the `bitbake` command to build the image.

Note: The base BSP variant uses a `arch/arm64/configs/defconfig` kernel configuration file. Local changes are overlaid on top of `defconfig` from the `meta-qcom-hwe/recipes-kernel/linux/linux-qcom-base-6.6/qcom.cfg` file.

To compile the Qualcomm Linux kernel for the base BSP variant, run the following commands:

```
# Use the filename of Machine conf file for MACHINE parameter
$ MACHINE=<SoC>-<board>-<variant> DISTRO=qcom-wayland QCOM_SELECTED_
BSP=base source setup-environment

# build qcom linux console image
$ bitbake qcom-console-image

# Build Images are produced under "build-qcom-wayland/tmp-glibc/
deploy/images/<SoC>-<board>-<variant>/qcom-console-image/"
```

To build the images listed in Table: Qualcomm Linux supported image recipes, the Qualcomm Linux kernel custom BSP recipe uses one of the following kernel configuration fragments:

Note: Downstream kernel configuration files contain `addons`.

Table: Custom BSP kernel configuration fragments

Kernel configuration fragments	Description
`<kernel_src>/arch/arm64/configs/qcom_defconfig`	Requires the base configuration of the product to be aligned with the upstream kernel
`<kernel_src>/arch/arm64/configs/qcom_debug.config`	Debugs a configuration fragment from the upstream kernel
`<kernel_src>/arch/arm64/configs/qcom_addons.config`	Adds Qualcomm downstream additions on top of the upstream aligned base
`<kernel_src>/arch/arm64/configs/qcom_addons_debug.config`	Enables Qualcomm downstream debug

To build Qualcomm Linux including the kernel for the custom BSP variant, run the following commands:

```
# Use the filename of Machine conf file for MACHINE
parameter
  MACHINE=<SoC>-<board>-<variant> DISTRO=qcom-wayland QCOM_
SELECTED_BSP=custom source setup-environment

# build qcom linux console image
  bitbake qcom-console-image

# Build Images are produced under "build-qcom-wayland/tmp-
glibc/deploy/images/<SoC>-<board>-<variant>/qcom-console-
image/"
```

The kernel default builds are defined in the `linux-qcom-base_6.6.bb` or `linux-qcom-custom_6.6.bb` file.

Qualcomm Linux kernel recipes support number 3 (performance) and number 4 (debug) builds mentioned in the following table. The default build configuration is number 3 (performance). The following kernel configuration fragments are only applicable to the custom BSP variant.

Table: Custom BSP kernel build configuration type and configurations

Serial number	Kernel build configuration type	Defconfig/config fragments
1	Base	`arch/arm64/configs/qcom_defconfig`
2	Debug-enabled base	• `arch/arm64/configs/qcom_defconfig` • `arch/arm64/configs/qcom_debug.config`
3	Base with downstream additions	• `arch/arm64/configs/qcom_defconfig` • `arch/arm64/configs/qcom_addons.config`
4	Debug-enabled base with downstream	• `arch/arm64/configs/qcom_defconfig` • `arch/arm64/configs/qcom_debug.config` • `arch/arm64/configs/qcom_addons.config` • `arch/arm64/configs/qcom_addons_debug.config`

For more information about configuring the kernel, see Kernel configurations.

Compile a Qualcomm debug build

Run the following command to compile a debug build:

```
DEBUG_BUILD=1 bitbake qcom-console-image
```

To modify the build configurations, update the `KERNEL_CONFIG` and `KERNEL_CONFIG_FRAGMENTS` variables in the `linux-qcom-base_6.6.bb` or `linux-qcom-custom_6.6.bb` kernel recipe in the `meta-qcom-hwe` layer.

Example kernel configuration

The following examples show kernel configuration for base and custom BSPs:

Base BSP

The following example shows the kernel configuration using the `linux-qcom-base_6.6.bb` kernel recipe:

```
KERNEL_CONFIG_FRAGMENTS:append = " ${WORKDIR}/qcom.cfg"

S = "${WORKDIR}/git"

# 6.6.38
SRCREV = "2928631d5304b8fec48bad4c7254ebf230b6cc51"
PV = "6.6+git${SRCPV}"

KERNEL_CONFIG ??= "defconfig"
```

Custom BSP

The following example shows the kernel configuration using the `linux-qcom-custom_6.6.bb` kernel recipe:

```
KERNEL_CONFIG ??= "qcom_defconfig"

KERNEL_CONFIG_FRAGMENTS:append = " ${S}/arch/arm64/configs/
qcom_addons.config"
KERNEL_CONFIG_FRAGMENTS:append = " ${@oe.utils.vartrue(
'DEBUG_BUILD', '${S}/arch/arm64/configs/qcom_debug.config',
'', d)}"
KERNEL_CONFIG_FRAGMENTS:append = " ${@oe.utils.vartrue(
'DEBUG_BUILD', '${S}/arch/arm64/configs/qcom_addons_debug.
config', '', d)}"

# Enable selinux support
```

```
SELINUX_CFG = "${@oe.utils.vartrue('DEBUG_BUILD', 'selinux_
debug.cfg', 'selinux.cfg', d)}"
KERNEL_CONFIG_FRAGMENTS:append = " ${@bb.utils.contains(
'DISTRO_FEATURES', 'selinux', '${WORKDIR}/${SELINUX_CFG}', '
', d)}"
```

For more information about the build instructions, see Build with QSC CLI in the Qualcomm Linux Build Guide.

2.4 Access the platform device tree

The Qualcomm device tree source inclusion (DTSI) and device tree source (DTS) files for supported development kits are located under the kernel source in the `arch/arm64/boot/dts/qcom/` directory. These Qualcomm files segregate the upstream-aligned and downstream additions.

The device tree files with `addons` are downstream files. The base BSP variant software doesn't use any downstream device tree files.

`<SoC>-<board>-<variant>.conf` is the Qualcomm Linux machine configuration file that contains the required device tree blob (DTB) selection configuration data.

- `SoC` is any supported Qualcomm Linux SoC. For example, QCS6490 and QCS9075.
- `board` is an RB3 Gen 2 supported product.
- `variant` is the specific type of product, for example core-kit.

The following example shows the DTB inclusion into the device configuration file:

```
KERNEL_DEVICETREE = " \
                     qcom/<SoC>-addons-<variant>.dtb \
                     "
```

QCS6490

The following table lists the Qualcomm® device tree files for the Qualcomm RB3 Gen 2 Development Kit.

Table: Qualcomm device tree source

Device tree source	Details
`arch/arm64/boot/dts/qcom/sc7280.dtsi`	The QCS6490 SoC is derived from SC7280 SoC.
`arch/arm64/boot/dts/qcom/qcm6490.dtsi`	The QCM6490/QCS6490 DTSI that hosts any changes is different from the SC7280 SoC.
`arch/arm64/boot/dts/qcom/qcm6490-addons.dtsi`	Downstream additions specific to the QCM6490/QCS6490 SoC.
`arch/arm64/boot/dts/qcom/qcs6490-rb3gen2.dts`	Device tree source for the QCS6490 Qualcomm RB3 Gen 2 Development Kit.
`arch/arm64/boot/dts/qcom/qcs6490-addons-rb3gen2.dts`	Downstream additions specific to the QCS6490 for the Qualcomm RB3 Gen 2 Development Kit.
`arch/arm64/boot/dts/qcom/qcs5430-addons-rb3gen2.dts`	Downstream additions specific to the QCS5430 feature pack-1 for the Qualcomm RB3 Gen 2 Development Kit.
`arch/arm64/boot/dts/qcom/qcs5430-fp2-addons-rb3gen2.dts`	Downstream additions for the QCS5430 feature pack-2 in the Qualcomm RB3 Gen 2 Development Kit.
`arch/arm64/boot/dts/qcom/qcs5430-fp2p5-addons-rb3gen2-vision-mezz.dts`	Downstream additions for the QCS5430 feature pack-2.5 in the RB3 Gen 2 vision Mezzanine Development Kit.
`arch/arm64/boot/dts/qcom/qcs5430-fp3-addons-rb3gen2-vision-mezz.dts`	Downstream additions for the QCS5430 feature pack-3 in the RB3 Gen 2 vision Mezzanine Development Kit.

QCS9075

The following table lists the Qualcomm device tree files for the QCS9075 Development Kit. By default, the addons DTB is used.

Table : Qualcomm device tree source

Device tree source	Details
`arch/arm64/boot/dts/qcom/sa8775p.dtsi`	The QCS9075 SoC is derived from SA8775P.
`arch/arm64/boot/dts/qcom/sa8775p-ride.dtsi`	Qualcomm Linux development kit for QCS9075.
`arch/arm64/boot/dts/qcom/qcs9075-ride.dts`	Qualcomm Linux development kit for QCS9075 ride.
`arch/arm64/boot/dts/qcom/qcs9075-addons-ride.dts`	Qualcomm Linux development kit for the QCS9075 ride with downstream additions.
`arch/arm64/boot/dts/qcom/qcs9075-ride-r3.dts`	Qualcomm Linux development kit for the QCS9075 ride R3.
`arch/arm64/boot/dts/qcom/qcs9075-addons-ride-r3.dts`	Qualcomm Linux development kit for the QCS9075 ride R3 with downstream additions.
`arch/arm64/boot/dts/qcom/qcs9075-iq-9075-evk.dts`	Qualcomm Linux development kit for the Qualcomm Dragonwing™ IQ-9075 EVK.
`arch/arm64/boot/dts/qcom/qcs9075-addons-iq-9075-evk.dts`	Qualcomm Linux development kit for the Qualcomm Dragonwing™ IQ-9075 EVK with downstream additions.

QCS8275

The following table lists the Qualcomm device tree files for the QCS8275 development kit. By default, the addons DTB is used.

Table : Qualcomm device tree source

Device tree source	Details
`arch/arm64/boot/dts/qcom/qcs8300.dtsi`	The QCS8275 SoC is derived from QCS8300.
`arch/arm64/boot/dts/qcom/qcs8300-addons.dtsi`	Qualcomm Linux development kit for QCS8275 with downstream additions.
`arch/arm64/boot/dts/qcom/qcs8300-ride.dts`	Qualcomm Linux development kit for QCS8275 ride.
`arch/arm64/boot/dts/qcom/qcs8300-addons-ride.dts`	Qualcomm Linux development kit for the QCS8275 ride with downstream additions.

2.5 Manage out-of-tree DT overlay fragments

SoC machine configuration files include additional out-of-tree DTB overlays (DTBO).

Graphics, camera, wireless local area network (WLAN) drivers, and their device tree configurations are maintained out of the kernel source tree. The respective device tree source overlay (DTSO) fragments are overlaid at the build time onto the base DTBs.

The DTBO configuration is done in the `conf/machine/<SoC>-<board>-<variant>.conf` file.

The following example shows the DTBO configuration:

```
# OUT_OF_KERNEL_DTSO - qcs6490-rb3gen2-core-kit.conf
# Additional list of DTBOs to be overlaid on top of base kernel
devicetree files
# Format - KERNEL_TECH_DTBOS[<base-dtb-name>] = "<dtbo1 <dtbo2> ..."
# For example:

KERNEL_TECH_DTBOS[qcs6490-addons-rb3gen2] = " \
qcm6490-graphics.dtbo qcm6490-wlan-rb3.dtbo \
```

```
qcm6490-display-rb3.dtbo qcm6490-bt.dtbo \
qcm6490-video.dtbo qcm6490-wlan-upstream.dtbo \
"
```

2.6 Manage out-of-tree kernel modules

Most of the kernel drivers are compiled from upstream kernel sources. However, some drivers are maintained outside the kernel source tree and are built as kernel modules using the Yocto build system.

In the following example, Bluetooth® is an out-of-tree kernel module. The recipe located at `recipes-connectivity/bt_dlkm_kernel/bt-dlkm-kernel_git.bb` compiles the driver as a kernel module. To autoload the kernel module on boot, you can add the module name to the `KERNEL_MODULE_AUTOLOAD` variable.

```
# Example out-of-tree Kernel module recipe
DESCRIPTION = "QCOM BT drivers"
LICENSE = "GPL-2.0-only"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/${LICENSE};
md5=801f80980d171dd6425610833a22dbe6"

inherit module

SRC_URI += "git://git.codelinaro.org/clo/le/platform/vendor/qcom-
opensource/bt-kernel.git;protocol=https;
rev=9e5cf29625d60f78e88440c94d096a9139445c00;branch=bt-performant.
qclinux.1.0.r1-rel \
         file://bt_dlkm \
         file://bt_dlkm.service \
         "

S = "${WORKDIR}/git"

RPROVIDES:${PN} += "kernel-module-bt-kernel"

EXTRA_OEMAKE += "MACHINE='${MACHINE}'"
MAKE_TARGETS = "modules"
MODULES_INSTALL_TARGET = "modules_install"
# Kernel module to be autoloaded
KERNEL_MODULE_AUTOLOAD += "bt_fm_slim"
```

2.7 Build the device image

To build the full image and kernel, run the following Yocto commands:

```
# setup the build environment
  export SHELL=/bin/bash

  source setup-environment
# (select "<SoC> meta-qcom" → Enter → select "qcom-wayland meta-
qcom-distro" → Enter)

# build qcom linux console image
  bitbake qcom-console-image

# Build kernel only
  bitbake linux-qcom-base

# Build Images are produced under
# build-qcom-wayland/tmp-glibc/deploy/images/<SoC>-<board>-<variant>/
qcom-console-image/
```

For more information about how to build different images for SoC-supported Qualcomm distribution, see Kernel configurations.

Note: To build a custom BSP, use `linux-qcom-custom`.

2.8 Bring up the device

The `efi.bin` file contains the systemd-boot (boot manager), kernel, and initramfs images. The Yocto build packs these images into `efi.bin` that's flashed to the EFI system partition (ESP) to boot the device.

Device tree blobs are updated or added to the `KERNEL_DEVICETREE` variable in the `conf/machine/<SoC>-<board>-<variant>.conf` file. These device tree blobs are used to compile the `dtb.bin` file.

Flash the `efi.bin` and `dtb.bin` images using the following fastboot commands:

```
# Bring the device in fastboot mode
# cd to the source root location
$ cd build-qcom-wayland/tmp-glibc/deploy/images/<SoC>/qcom-console-
image
$ fastboot flash efi efi.bin
```

```
$ fastboot flash dtb_a dtb.bin
$ fastboot reboot
```

After the Qualcomm Linux build is generated, the kernel build is completed and the kernel-related images are created in the `tmp-glibc/deploy/images/<SoC>-<board>-<variant>/` directory.

The following table lists the kernel build artifacts:

Table : Kernel build artifacts

Image	Image name	Build deploy path	Details
Kernel executable and linking format (ELF)	`vmlinux`	`tmp-glibc/deploy/images/<SoC>-<board>-<variant>/`	Output kernel ELF with debug symbols.
Initramfs	`initramfs-qcom-image-<SoC>-<board>-<variant>.cpio.gz`	`tmp-glibc/deploy/images/<SoC>-<board>-<variant>/`	Initramfs in copy in, copy out (CPIO) file format.
Kernel image	`Image`	`tmp-glibc/deploy/images/<SoC>-<board>-<variant>/`	Kernel raw image binary, systemd-boot doesn't support compressed images.
Kernel modules	Different kernel dynamically loadable kernel modules (DLKMs) `modules-<SoC>-<board>-<variant>.tgz`	`tmp-glibc/deploy/images/<SoC>-<board>-<variant>/`	Kernel drivers modules.

Image	Image name	Build deploy path	Details
Device tree blobs	`<SoC>-addons-<VARIANT>.dtb`	`tmp-glibc/deploy/images/<SoC>-<board>-<variant>/`	Individual device tree blobs.
ESP partition	`efi.bin`	`tmp-glibc/deploy/images/<SoC>-<board>-<variant>/`	All required boot images including systemd-boot, kernel, and initramfs are packaged into the extensible firmware interface (`efi.bin`) binary.
DTB partition	`dtb.bin`	`tmp-glibc/deploy/images/<SoC>-<board>-<variant>/`	All the DTBO files are merged into the `dtb.bin` binary.

3 Kernel features

The Qualcomm® Linux® kernel offers the following key features and advancements related to the mainline Linux kernel on Qualcomm platforms:

- The Qualcomm Linux BSP is tailored to support devices using Qualcomm platforms.
- The device tree corresponds to the Qualcomm Linux development kits.
- Multiple customized build configurations are available to suit your requirements.
- The Qualcomm Linux kernel integrates into the Yocto build system.
- The Qualcomm Linux kernel aligns with the upstream LTS kernel. Qualcomm-specific additions are maintained separately.
- Support to configure, customize, and build kernel images that's flashed and booted on devices using Qualcomm hardware SoCs.
- Virtualization support for untrusted virtual machines using the Gunyah™ Hypervisor Software.
- Periodic LTS merges from the corresponding LTS kernel branch to get the latest security and stability fixes.

3.1 UEFI boot manager

Qualcomm Linux supports systemd-boot as the Universal extensible firmware interface (UEFI) boot manager to load and boot the Linux kernel. The Qualcomm Linux kernel, in this case, is built as an EFI stub.

Boot flow and architecture overview

A cold boot refers to the process of starting the system from a power-off state. The cold boot process involves the following steps:

1. Cold boot begins the execution from the primary boot loader (PBL) that sets up the initial system for the eXtensible Boot Loader (XBL).
2. The XBL performs wider system initialization including DDR initialization and loads a UEFI image.
3. UEFI provides a rich firmware interface that loads systemd-boot as a boot manager on Qualcomm Linux to manage the OS images.

For more information about complete cold boot flow, see Cold boot architecture in the Qualcomm Linux Boot Guide.

Systemd-boot

Systemd-boot is a UEFI boot manager that executes EFI images, provides boot entries, and supports unified kernel images (UKI). Systemd-boot supports the following components:

- Boot entries: The type 1 boot loader specification entries are in the `loader/entries/` directory on the ESP. These files describe Qualcomm Linux kernel images with the associated initrd images, and other EFI executables.
- UKI: The type 2 boot loader specification EFI unified kernel images are executable EFI binaries located in the `/EFI/Linux/` directory on the ESP.

  For more information about the boot entries and unified kernel images, see BootLoader Specification.

Systemd-boot is a part of the systemd package of the meta Yocto layer. The Yocto recipe uses the `systemd-boot_254.4.bb` recipe file to build systemd-boot, and the `uki.bbclass` recipe file to handle the ESP image generation.

For more information about systemd-boot, see Configure and secure boot with systemd-boot and UKI.

Qualcomm Linux kernel as EFI stub

The EFI boot stub allows booting the Qualcomm Linux kernel directly without a conventional EFI boot loader.

The boot firmware can load the EFI image as an executable file when the Qualcomm Linux kernel is compiled with the `CONFIG_EFI_STUB` kernel configuration option. In this case, the firmware loader navigates to the `EFI boot stub` location in the EFI image `drivers/firmware/efi/libstub/` to boot the kernel.

For Arm® (Arm64), where compressed kernel support isn't available, the kernel image functions as a portable executable (PE) file format or common object file format (COFF) image, and the EFI stub is linked into the kernel.

For more information about booting the Linux kernel as an EFI image, see The EFI Boot Stub.

Boot images and ESP or boot partition

The ESP or boot partition serves as a storage location for the `efi.bin` image that packages systemd-boot and UKI. The UKI includes kernel image, initramfs, and kernel command-line arguments.

The UEFI firmware launches the UEFI boot loader and loads the kernel boot images. The ESP is formatted with the file allocation table (FAT) file system supported in the UEFI specification. For more information about the UKI image format and the corresponding support in base Qualcomm Linux meta layers, see systemd-boot.

DTB selection

All the device tree blobs are packaged as part of `dtb.bin`. The UEFI selects and loads the right DTB for the Qualcomm Linux kernel.

To understand the boot time DTB selection, see Platform device tree and kernel configuration.

3.2 Configure the DTB support

Qualcomm Linux uses the following procedures to choose device tree files, change kernel configuration, and build out-of-tree kernel modules.

Platform device tree

Qualcomm Linux supports a device tree overlay feature to maintain and merge out-of-device tree blobs with the baseline device tree blobs. SoC device tree support is present in the kernel source in the `arch/arm64/boot/dts/qcom` directory.

Qualcomm device tree source lists the device tree and maintains a clear separation to contain downstream additions for provisioning the upstream aligned base distinctly.

The downstream content is hosted outside the kernel source. The out-of-tree drivers maintain their device tree overlay additions outside the kernel. During the build process, the corresponding device tree overlay is merged with the base board DTB and a unified DTB is generated, which is used with boot images.

The following figure shows the build process for device tree overlay, and DTB generation for custom BSP on QCS6490:

Note:

- Base BSP follows the same build process without the **addons** downstream device-tree files.
- All supported Qualcomm hardware SoCs follow the same build process for DT overlay and DTB generation.

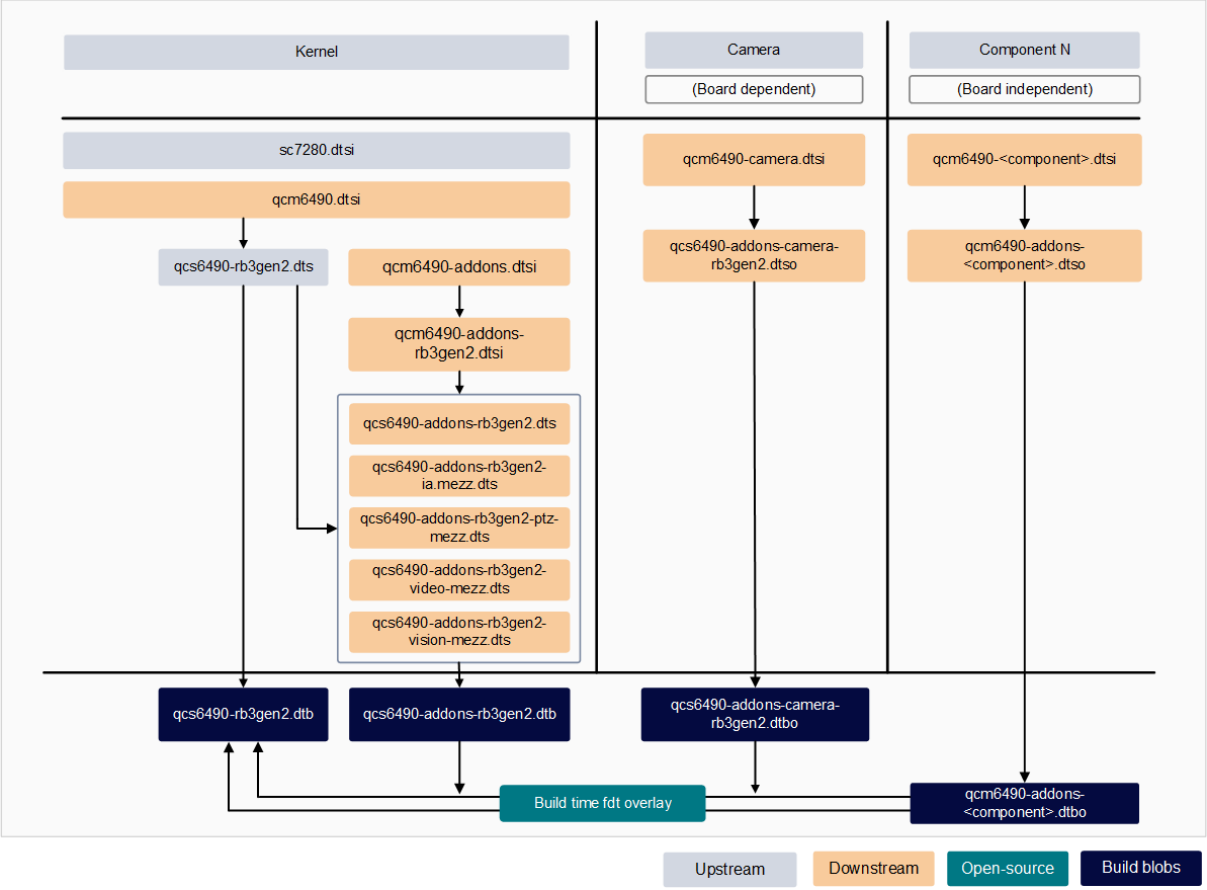

Figure: Device tree overlay on QCS6490

The build system uses an `fdt_overlay` tool to provision the out-of-tree device tree overlay handling during the build.

The DTBO merge process supports merging the DTBO files listed in the `meta-qcom-hwe/conf/machine/<SoC>-<board>-<variant>.conf` file.

The following is an example of how device tree variables are enumerated on QCS6490. See the board-specific machine configuration file for platform-specific information.

```
# List of dtbs for corresponding supported qcs6490 platforms
KERNEL_DEVICETREE = " \
                    qcom/<SoC>-addons-<Variant>.dtb \
                    "
# Additional list of dtbos to be overlaid on top of base kernel
devicetree files
# Format - KERNEL_TECH_DTBOS[<base-dtb-name>] = "<dtbo1 <dtbo2> ..."
# For example:

KERNEL_TECH_DTBOS[qcs6490-addons-rb3gen2] = " \
```

```
qcm6490-graphics.dtbo qcm6490-wlan-rb3.dtbo \
qcm6490-display-rb3.dtbo qcm6490-bt.dtbo \
qcm6490-video.dtbo qcm6490-wlan-upstream.dtbo \
"
```

DTBO merge sequence

The `merge_dtbos()` task in the `image-qcom-deploy.bbclass` class merges DTBOs using the `fdt_overlay` tool.

The `merge_dtbos()` task iteration occurs in the following sequence:

1. DTBO filenames get listed in the configuration file.
2. DTBO filenames are iterated through the `KERNEL_DEVICETREE` variable.
3. For each base DTB, the DTBO merge task overlays the DTBOs listed in the `KERNEL_TECH_DTBOS` variable.
4. After the overlay process is completed, the final DTB output is generated.

Kernel configurations

The `linux-kernel-qcom-base_6.6.bb`, or `linux-qcom-custom_6.6.bb` kernel recipe file uses the following configuration fragments to build the images:

For example, `linux-qcom-custom_6.6.bb` uses the following fragments to append the kernel:

```
KERNEL_CONFIG ??= "qcom_defconfig"
KERNEL_CONFIG_FRAGMENTS:append = " ${S}/arch/arm64/configs/qcom_
addons.config"
KERNEL_CONFIG_FRAGMENTS:append = " ${@oe.utils.vartrue('DEBUG_BUILD',
'${S}/arch/arm64/configs/qcom_debug.config', '', d)}"
KERNEL_CONFIG_FRAGMENTS:append = " ${@oe.utils.vartrue('DEBUG_BUILD',
'${S}/arch/arm64/configs/qcom_addons_debug.config', '', d)}"

# Enable selinux support
SELINUX_CFG = "${@oe.utils.vartrue('DEBUG_BUILD', 'selinux_debug.cfg
', 'selinux.cfg', d)}"
KERNEL_CONFIG_FRAGMENTS:append = " ${@bb.utils.contains('DISTRO_
FEATURES', 'selinux', '${WORKDIR}/${SELINUX_CFG}', '', d)}"
```

To generate the debug build using debug configuration fragments, run the following commands:

```
# Set DEBUG_BUILD to 1 to compile debug-enabled build

  source setup-environment
  DEBUG_BUILD=1 bitbake linux-kernel-custom
```

For more information about different configuration fragments, see Getting started with Qualcomm Linux kernel.

3.3 Remoteproc subsystem

The Remoteproc framework is used to load firmware into other subsystems, such as the audio digital signal processor (aDSP) and compute digital signal processor (cDSP) on Qualcomm platforms.

The remoteproc framework is divided into two parts:

- Core framework: The core framework contains a common logic. It loads the firmware and starts or stops the remote processor.
- Remoteproc drivers: The drivers contain platform-specific operations to manage the corresponding cores. A remoteproc driver registers a remoteproc instance and a set of operations with the core framework.

Several heterogeneous remote processors are present on an SoC in an asymmetric multiprocessing (ASMP) configuration. The remote processors run different instances of firmware or OS.

For more information about the standard framework that Qualcomm Linux uses to manage other cores, see Remote Processor Framework.

Qualcomm remoteproc support

Qualcomm has enhanced the remoteproc framework for peripheral firmware authentication using the upstream PAS driver.

The Remoteproc driver in Qualcomm Linux implements a peripheral authentication service (PAS) driver, which is a TrustZone-based peripheral image loader for remote processors on Qualcomm SoC devices.

For more information, see qcom_q6v5_pas.c.

The remote processors are used to manage the lifecycle of various co-processors, such as aDSP, cDSP, modem peripheral subsystem (MPSS), and wireless processor subsystem (WPSS). The supported co-processors on QCS6490-based boards and RB3 Gen 2 Development Kit are aDSP, cDSP, and WPSS.

Note: List of supported remote processor subsystems depends upon the SoC in use.

User interface: The remoteproc framework tracks registered remoteproc devices and provides a user interface to boot and shutdown the devices. The user space can query and change the

current state of a remoteproc using the following `sysfs` interface:

```
remoteproc user space interface
# start remoteproc:
  echo "start" > /sys/class/remoteproc/remoteprocN/state

# stop remoteproc:
  echo "stop" > /sys/class/remoteproc/remoteprocN/state
```

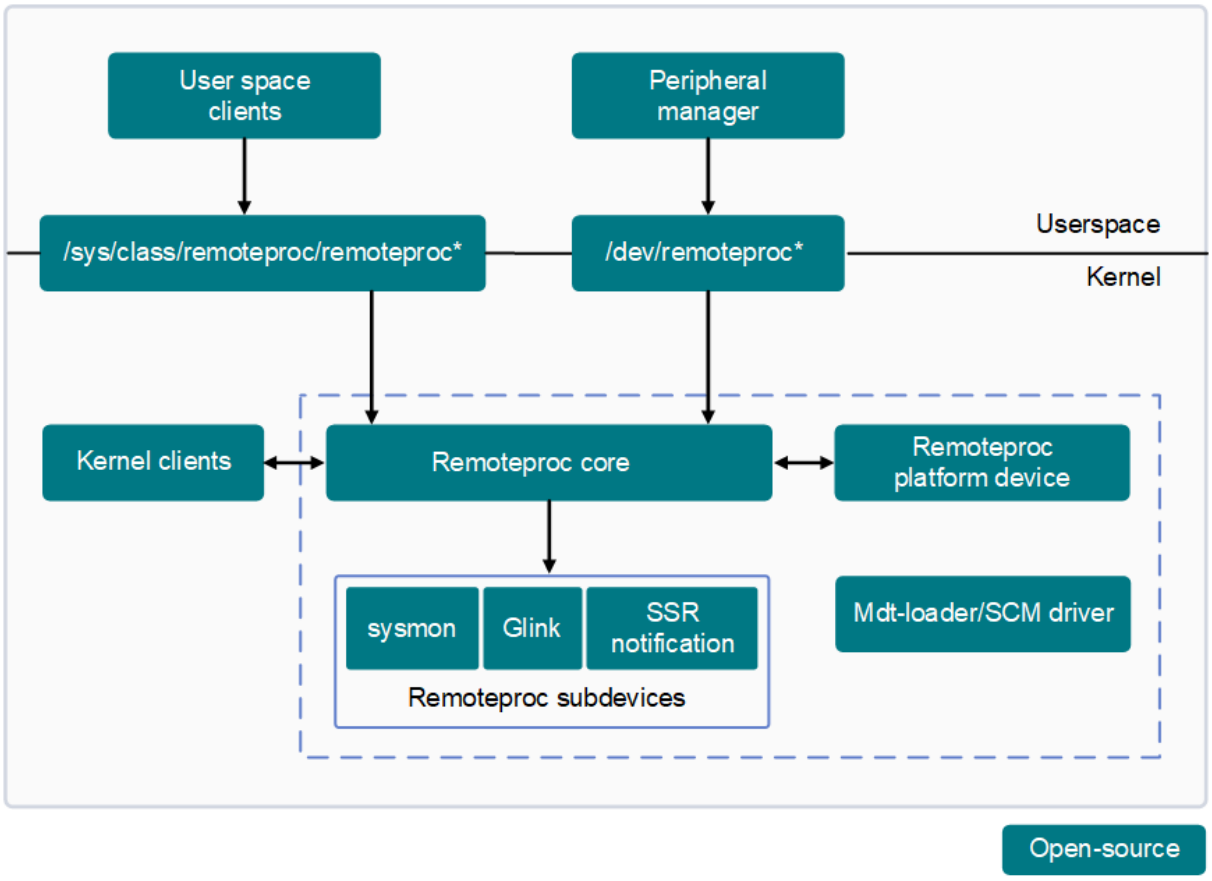

Figure : Remoteproc support

Remoteproc recovery and subsystem restart

The remoteproc subdevices manage communication contexts for remote processor subsystem crash and recovery scenarios.

Entities, such as communication contexts, must determine when a remote processor boots or shuts down. Each subdevice has two operations:

- `probe()` called after the remoteproc `start()`
- `remove()` called before `stop()`

When a fatal error occurs on a remote processor, the remoteproc driver handles it and invokes `rproc_report_crash()`, which triggers recovery of the faulty remote processor.

The recovery handler does the following:

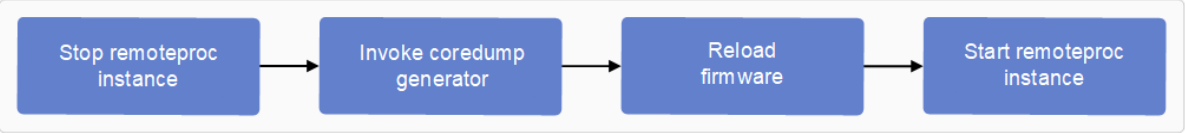

Figure: Remoteproc crash sequence

To enable and disable recovery using the recovery `debugfs` node from the user space, do the following:

```
mount -t debugfs nodev /sys/kernel/debug
echo disabled > /sys/kernel/debug/remoteproc/remoteprocN/recovery
```

Coredump

Remote processor coredumps are supported for debugging the subsystem crash issues using the Qualcomm Linux kernel devcoredump feature.

When a subsystem crash occurs, devcoredump exposes a snapshot of the memory of the recovering remoteproc in the `/sys/class/devcoredump/devcdN/data` node, and an associated crash `uevent` is sent to the user space. When the sysfs node is read, the sysfs provides the segments in an ELF container, and a write operation frees up the resources and destroys the devcoredump instance.

A debugfs node is exposed to enable and disable coredump from the user space.

```
# Disable coredumps:
  echo disabled > /sys/kernel/debug/remoteproc/remoteprocN/coredump

# Enable coredumps:
  echo enabled > /sys/kernel/debug/remoteproc/remoteprocN/coredump
#
```

```
# N represents the index number of the remote processor.
```

Note: To collect the full RAM dump for crash analysis, Qualcomm SoCs implement kernel panic on remoteproc crash (fatal error on remote processor). The kernel panic mode is enabled only when the coredump is disabled.

The devcoredump `/sys/class/devcoredump/devcdN/data` node is created on a subsystem crash only if coredump is enabled for the subsystem. To enable the coredump, run the `echo enabled > /sys/kernel/debug/remoteproc/remoteprocN/coredump` command. The `devcdN/data` node is temporary and is removed after a timeout. To debug the issues, use the `copy` command to copy the `devcdN/data` node to your local device for secure usage. The delete timeout is defined in `base/devcoredump.c`.

The following is an example command to copy the coredumps:

```
# Copy Coredump to a local device
  cp /sys/class/devcoredump/devcdN/data > /var/spool/crash/dump_file.
elf
#
# N represents the index of coredump. It is incremented each time a
new coredump is generated.
```

To transfer the coredump file to a host device and debug the issues, use the Qualcomm Crash Analysis Portal (QCAP).

Note: For more information about QCAP, see Parse RAM dumps using QCAP.

Remoteproc configuration and firmware

The remoteproc devices are configured in the respective device tree files using the DT bindings. The configuration includes compatible device names, memory regions, interrupt settings, and clock references, according to the DT bindings, in the `Documentation/devicetree/bindings/remoteproc/` directory.

The following example shows the QCS6490 aDSP remoteproc device tree configuration in the `arch/arm64/boot/dts/qcom/qcs6490.dtsi` file:

```
remoteproc_adsp: remoteproc@3000000 {
          compatible = "qcom,sc7280-adsp-pas";
          reg = <0x0 0x03000000 0x0 0x100>;

          interrupts-extended = <&pdc 6 IRQ_TYPE_EDGE_RISING>,
```

```
                                    <&adsp_smp2p_in 0 IRQ_TYPE_EDGE_
RISING>,
                                    <&adsp_smp2p_in 1 IRQ_TYPE_EDGE_
RISING>,
                                    <&adsp_smp2p_in 2 IRQ_TYPE_EDGE_
RISING>,
                                    <&adsp_smp2p_in 3 IRQ_TYPE_EDGE_
RISING>,
                                    <&adsp_smp2p_in 7 IRQ_TYPE_EDGE_
RISING>;
          interrupt-names = "wdog", "fatal", "ready", "handover",
                            "stop-ack", "shutdown-ack";

          clocks = <&rpmhcc RPMH_CXO_CLK>;
          clock-names = "xo";
          <snip>
 };
```

Check the QCS9075 device tree configuration file in the `arch/arm64/boot/dts/qcom/sa8775p.dtsi` path.

Note: To get the device tree configuration file for all the Qualcomm SoCs, see the respective platform DTSI file.

All firmware files are present in the `/lib/firmware` directory in `rootfs` and the related configurations are done in the corresponding board device tree `arch/arm64/boot/dts/qcom/qcs6490-rb3gen2.dts` file. For other Qualcomm hardware SoCs, see `arch/arm64/boot/dts/qcom/<SoC>-<board>.dts` files.

The following example shows the remoteproc DT configuration:

```
&remoteproc_adsp {
firmware-name = "qcom/qcs6490/adsp.mdt";
status = "okay";
};
```

Sample logs

The following log is displayed when a remoteproc boots successfully:

```
**
\# remoteproc success**
"remoteproc remoteproc1: remote processor 4080000.remoteproc is now
up"
```

The following log is displayed when a remoteproc firmware load fails:

```
**
\# remoteproc failure**
Error log: "remoteproc remoteproc0: Direct firmware load for qcom/
qcs6490/modem.mdt failed with error -2"

# Caused by: error loading firmware.
# Solution: ensure that the modem firmware is copied to /lib/
firmware/qcom/qcs6490 in rootfs.
```

3.4 Memory management and configuration

The Qualcomm® Linux kernel baseline supports all memory management features and allocators. The following information outlines customizing the memory map and performing heap management.

For more information about Linux kernel memory management, see Memory management.

Memory map describes the areas that are reserved for subsystems, such as modem, camera, aDSP, and cDSP during kernel boot.

The memory map sets `no-map` for the carved out regions in the DTSI, making them inaccessible to the kernel.

Configurable carved-out regions are defined in the `arch/arm64/boot/dts/qcom/qcs6490.dtsi` file under the `reserved-memory` node.

```
reserved-memory {
                cdsp_secure_heap_mem: cdsp-secure-heap@81800000 {
                        reg = <0x0 0x81800000 0x0 0x1e00000>;
                        no-map;
                };

                camera_mem: camera@84300000 {
                        reg = <0x0 0x84300000 0x0 0x500000>;
                        no-map;
```

```
                };

                wpss_mem: wpss@0x84800000 {
                        reg = <0x0 0x84800000 0x0 0x1900000>;
                        no-map;
                };

                adsp_mem: adsp@86100000 {
                        reg = <0x0 0x86100000 0x0 0x2800000>;
                        no-map;
                };
};
```

Note: For Qualcomm SoCs, see the SoC-specific Qualcomm DTSI files to get this information.

The following early boot log shows carved out region creation for different subsystems:

```
[    0.000000] OF: reserved mem: 0x0000000081800000..
0x00000000835fffff (30720 KiB) nomap non-reusable cdsp-secure-
heap@81800000
[    0.000000] OF: reserved mem: 0x0000000084300000..
0x00000000847fffff (5120 KiB) nomap non-reusable camera@84300000
[    0.000000] OF: reserved mem: 0x0000000084800000..
0x00000000860fffff (25600 KiB) nomap non-reusable wpss@0x84800000
[    0.000000] OF: reserved mem: 0x0000000086100000..
0x00000000888fffff (40960 KiB) nomap non-reusable adsp@86100000
```

Qualcomm Linux distribution supports contiguous memory allocator (CMA) to allocate large physically contiguous memory. CMA reserves a large physically contiguous memory area at boot time and provides physically continuous memory to CMA allocation. When not in use, CMA memory is available to the kernel buddy allocator for movable allocations.

To change the size of the default CMA region, run `cma=size_in_MB` in the kernel command-line arguments. For more information on how to use the kernel parameter, see the following example:

```
cma=nn[MG]@[start[MG][-end[MG]]]
                        [KNL,CMA]
                        Sets the size of kernel global memory area
for
                        contiguous memory allocations and optionally
the
                        placement constraint by the physical address
range of
```

```
                        memory allocations. A value of 0 disables CMA
                        altogether. For more information, see
                        kernel/dma/contiguous.c
```

The custom CMA regions are defined under the `reserved-memory` node with a `shared-dma-pool` compatible tag indicating the CMA region:

```
adsp_heap_mem: adsp-heap {
                        compatible = "shared-dma-pool";
                        alloc-ranges = <0x0 0x00000000 0x0
0xffffffff>;
                        reusable;
                        alignment = <0x0 0x400000>;
                        size = <0x0 0xc00000>;
                };
```

The following is the sample log for an aDSP CMA memory region reserved on boot:

```
[    0.000000] OF: reserved mem: initialized node adsp-heap,
compatible id shared-dma-pool
[    0.000000] OF: reserved mem: 0x00000000ff000000..
0x00000000ffbfffff (12288 KiB) map reusable adsp-heap
```

DMA-BUF heaps are supported on the Qualcomm Linux distribution to allocate custom CMA heaps. With DMA-BUF heaps, a device file is present for each heap in the `/dev/dma_heap` file system. Apart from the system heap and the common CMA reserved heap, you can create your own CMA-type DMA-BUF heaps.

Supported heaps

The following table lists the heaps that are supported by default on the Qualcomm Linux distribution:

Table: Default supported heaps

Heap name	Dev node	Description	Usage
System	`/dev/dma_heap/system`	The kernel creates the default DMA-BUF heap.	All generic use cases must follow the system heap that uses the common Linux memory management buddy allocator underneath.

Heap name	Dev node	Description	Usage
Reserved	`/dev/dma_heap/reserved`	The default CMA-type heap created in the system uses the default *reserved* CMA region.	If you need contiguous memory due to any constraints, use the CMA heap.
Custom CMA heaps	`/dev/dma_heap/my_cma_heap`	User-defined CMA-type heaps.	If you want to create your own CMA-type heap for specific custom CMA heaps.

Use DMA-BUF heaps

The following is a sample program that demonstrates how to use the DMA-BUF system heap created by the Qualcomm Linux kernel in `/dev/dma_heap/system`:

```
include <stdio.h>
include <stdlib.h>
include <fcntl.h>
include <errno.h>
include <unistd.h>
include <sys/ioctl.h>
include <linux/dma-buf.h>
#include <linux/dma-heap.h>

define DMA_HEAP_NAME "system"
define SZ_4 0x00000004  // to allocate a 4K buffer

int main()
{
int fd, dma_buf_fd;

struct dma_heap_allocation_data dma_alloc_data = {
   .len = SZ_4,
   .fd_flags = O_RDWR | O_CLOEXEC,
};

struct dma_buf_sync sync_start = {
   .flags = DMA_BUF_SYNC_START,
};
struct dma_buf_sync sync_end = {
   .flags = DMA_BUF_SYNC_END,
};
```

```
fd = open("/dev/dma_heap/system", O_RDWR);
if (fd < 0) {
   perror("open");
   return errno;
}

dma_buf_fd = ioctl(fd, DMA_HEAP_IOCTL_ALLOC, &dma_alloc_data);
if (dma_buf_fd < 0) {
   perror("ioctl");
   return errno;
}
printf("Allocated DMA buffer with fd %d\n", dma_buf_fd);

if (ioctl(dma_buf_fd, DMA_BUF_IOCTL_SYNC, &sync_start)) {
   perror("ioctl DMA_BUF_IOCTL_SYNC start");
   return errno;
}

//        Do something with the buffer here

if (ioctl(dma_buf_fd, DMA_BUF_IOCTL_SYNC, &sync_end))
{
   perror("ioctl DMA_BUF_IOCTL_SYNC end");
   return errno;
}

if (close(dma_buf_fd)) {
   perror("close");
   return errno;
}

if (close(fd)) {
   perror("close");
   return errno;
}

return 0;
}
```

3.5 Scheduler

The scheduler decides the order in which the processes must run. The scheduler runs the processes from the runqueue of each CPU.

The kernel baseline supports the standard Linux scheduler solution. The kernel uses Energy Aware Scheduling (EAS) to choose the right CPU for task placement based on the CPU energy consumption.

EAS overrides the completely fair scheduler (CFS) task wake-up balancing code. It uses the energy model (EM) of the CPUs and the per entity load tracking (PELT) signals to choose an energy-efficient target CPU during wake-up balance in a system with asymmetric CPU topology.

For basic scheduler documentation, see Scheduler.

CPU topology and EAS

The Qualcomm SoCs have a heterogeneous CPU topology that's differentiated in terms of CPU capacity metrics used in EAS.

EAS uses the concept of *capacity* to differentiate CPUs with different computing capabilities. The capacity of a CPU represents the amount of work it finishes when running at its highest frequency compared to the most capable CPU of the system. Capacity values are normalized in a 1024 range (the most powerful CPU/cluster is configured at 1024).

EAS builds the capacity of a cluster based on the Dhrystone million instructions per second (DMIPS) value specified in the CPU node and the maximum frequency supported by the cluster.

The following are the `sysfs` nodes for CPU topology and capacity:

Table: CPU topology and capacity

Commands	Purpose
`cat /sys/devices/system/cpu/cpu*/cpu_capacity`	To get the capacity associated with each CPU in the system.
`cat /sys/devices/system/cpu/cpufreq/policy*/related_cpus`	To get a list of the CPUs associated with each cluster in the system.

For more information about how a scheduler uses CPU capacity, see Capacity Aware Scheduling.

Per entity load tracking

Per entity load tracking (PELT) is the mechanism used to track load accounting for `sched_entities/groups/runqueues`.

The PELT clock multiplier is tuned to meet power and performance requirements.

The command-line parameter `sched_pelt_multiplier` allows you to set a clock multiplier. The clock multiplier tunes the PELT ramp up/down speed to tune the system for power and performance.

With lower half-life time, the CPU utilization of tasks is accumulated faster. The tasks are placed on performance CPUs, thereby leading to better performance with high power consumption. The CPU utilization also gets accumulated to select the higher CPU frequency within less time.

Note: The half-life time also affects how the task utilization ramps down. The UTIL EST feature provides faster ramp down.

For more information about PELT, see Per-entity load tracking.

SchedUtil governor

The SchedUtil CPU frequency governor is the default governor in the Qualcomm Linux kernel. SchedUtil predicts optimal operating points (OPPs) based on the CPU utilization, maintaining coherence between frequency requests and energy predictions.

The SchedUtil CPU frequency governor is tied to EAS, and it tries to predict at which OPP all the CPUs run next to estimate their energy consumption.

For more information, see SchedUtil.

Utilization clamping (UCLAMP)

The UCLAMP operation allows performance management of tasks/task groups from the user space.

UCLAMP allows the placement of tasks or task groups and allows performance hints or constraints. This mechanism is used to influence the scheduler placement decisions or to influence the frequency guidance of the cluster.

For more information, see UCLAMP.

3.6 Dynamic voltage and frequency scaling (DVFS)

The Qualcomm Linux kernel baseline supports several DVFS implementations to manage frequency scaling in a dynamic way corresponding to the system requirement.

CPU DVFS governors

You can select a governor to tune the system for power or performance while changing the `CPUfreq` governor (`powersave/performace/schedutil`). By default, the CPU frequency governor is set to `performance` on Qualcomm Linux kernel.

Table: CPU DVFS variables

Variable	Description	Path
`scaling_governor`	Set the governor to performance.	`echo performance > /sys/devices/system/cpu/cpufreq/policy*/scaling_governor`
`scaling_max_freq`	Set the maximum frequency of the CPU cluster.	`/sys/devices/system/cpu/cpufreq/policy*/scaling_max_freq`
`scaling_min_freq`	Set the minimum frequency of the CPU cluster.	`/sys/devices/system/cpu/cpufreq/policy*/scaling_min_freq`

For more information about `CPUfreq`, see CPU frequency and voltage scaling code in the Linux(TM) kernel.

Cache and memory DVFS governors

The static map DVFS governors align the CPU operating frequencies with the following:

- Level 3 cache (L3)
- Last level cache controller (LLCC)
- DDR RAM

If the CPU frequency is at the maximum level, either by setting the CPU frequency governor to performance, or due to load in the system, then the L3, LLCC, and DDR also run at their maximum frequencies.

The driver for this governor is present in the `drivers/cpufreq/qcom-cpufreq-hw.c` file, and static mapping is present in the `arch/arm64/boot/dts/qcom/sc7280.dtsi` file.

Note: For Qualcomm SoCs, see the platform-specific DTSI files to get this information.

The other bandwidth monitor (BWMON) governor is used to vote for LLCC and DDR frequencies, based on the measured traffic between CPU to LLCC and CPU to DDR. The driver for this governor is present in the `drivers/soc/qcom/icc-bwmon.c` file.

4 Enable virtualization

Virtualization uses software to create an abstraction layer over computer hardware.

The abstraction layer divides the hardware elements of a single computer, such as processor, memory, and storage into different virtual entities, called as virtual machines (VMs). Each VM runs its own OS and behaves like an independent computer, though it's running on the same underlying computer hardware.

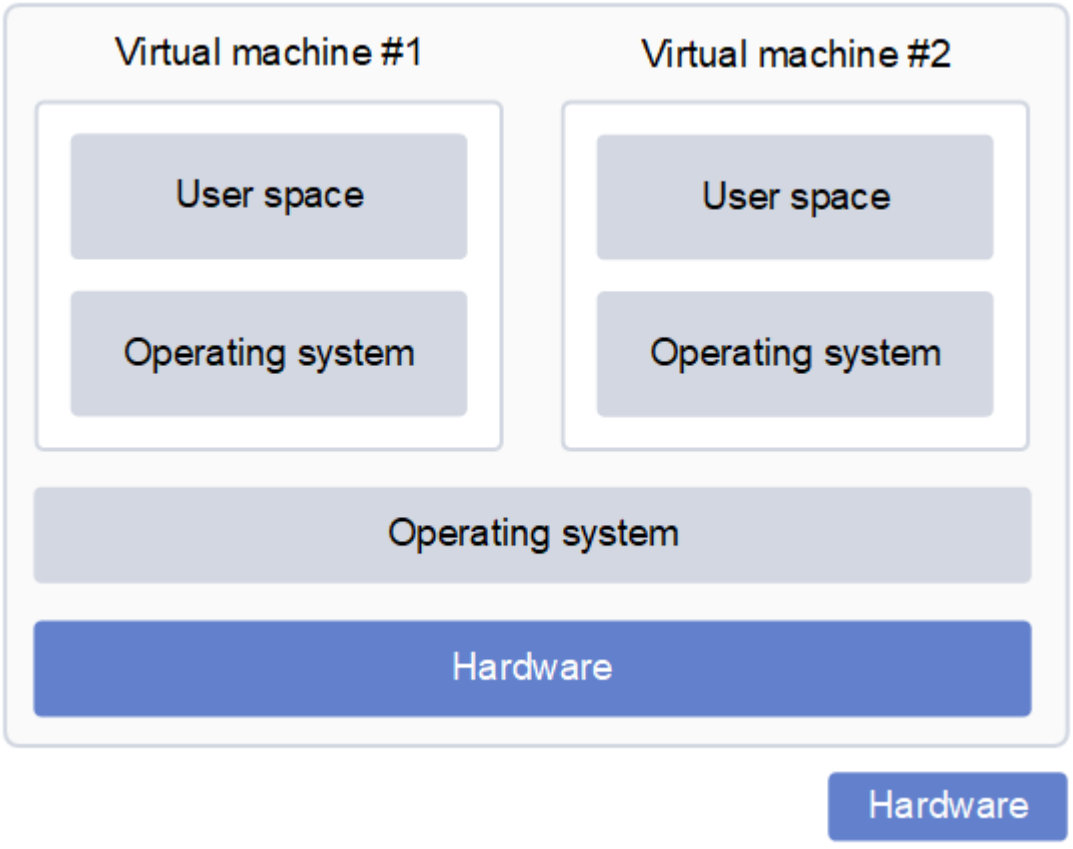

Figure: Virtualization

Note: Virtualization is supported on the custom BSP variant only.

4.1 Qualcomm virtualization solution

Qualcomm uses Gunyah, a type-1 hypervisor designed for strong security, high performance, OS independence, and higher CPU privilege level.

Gunyah doesn't depend on any lower-privileged OS kernel or code for its core functionality to offer increased security. Gunyah is designed for isolated virtual machine use cases, and for launching isolated virtual machines from a relatively less trusted host virtual machine.

The following figure shows the Qualcomm virtualization solution architecture:

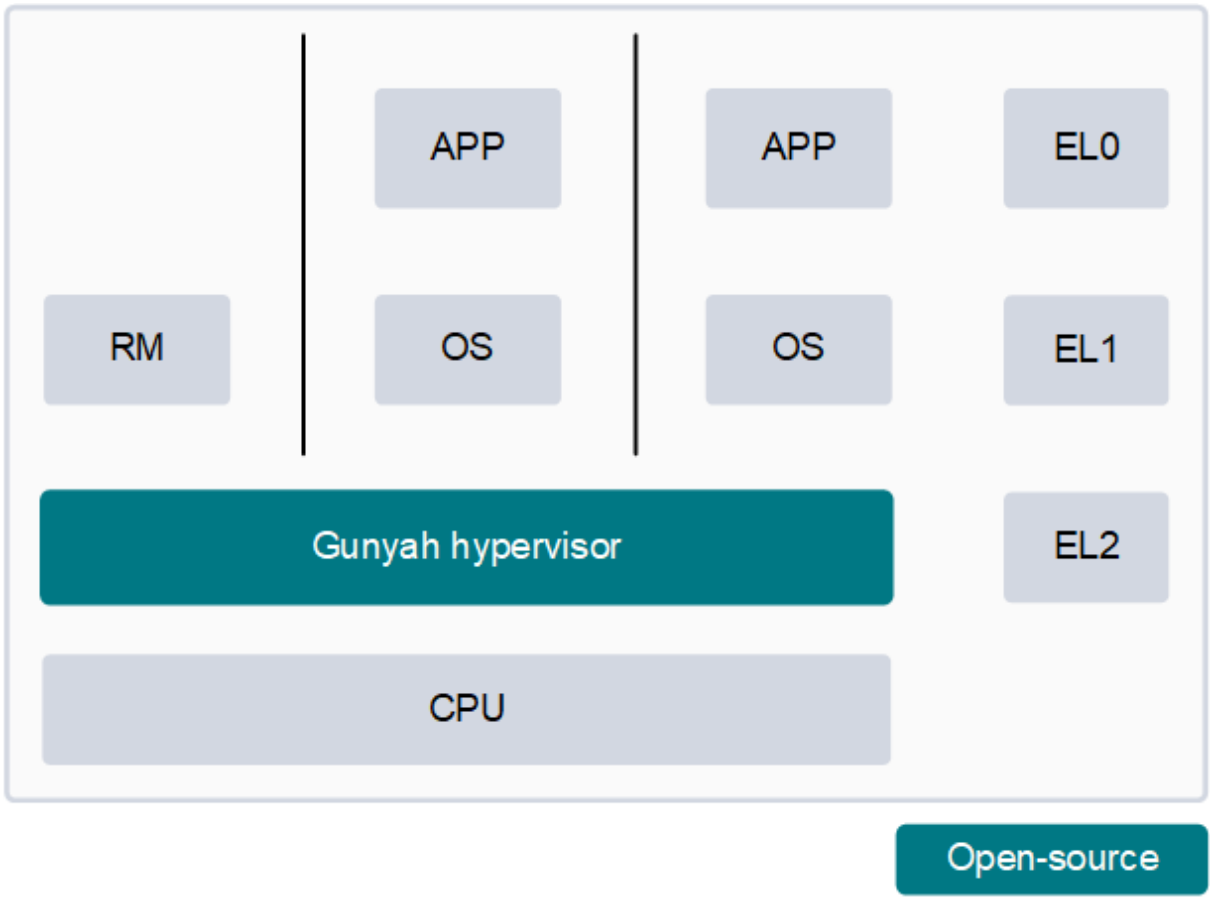

Figure: Qualcomm virtualization solution architecture

Gunyah provides the following features:

Table: Features of Gunyah

Features	Description
Threads and scheduling	Scheduler schedules virtual CPUs (VCPUs) on physical CPUs and allows time-sharing of the CPUs.

Features	Description
Memory management	Gunyah tracks memory ownership and memory usage under its control. Gunyah provides low-level dynamic memory management APIs on top of which the higher-level donation, lending, and sharing are built.
Interrupt virtualization	Gunyah manages interrupts that are routed directly to the assigned VM.
Inter-VM communication	Message queues and doorbells provide mechanisms for communication between the VMs.
Device virtualization	Para-virtualization of devices is supported using inter-VM communication and virtio primitives. The hardware virtualization and emulation support low-level architecture features and devices, such as CPU timers and interrupt controllers.
Resource manager	Gunyah supports a root VM that initially owns all the VM memory and input/output resources. The Gunyah resource manager is the default-bundled root VM and provides high-level services including dynamic VM management, secure memory donation, lending, and sharing.

The Gunyah hypervisor support is added to the crosvm virtual memory monitor (VMM). The crosvm interacts with the Gunyah VM manager in the kernel through the `/dev/gunyah` interface and creates a VM.

The following figure shows the blocks in the Qualcomm virtualization solution:

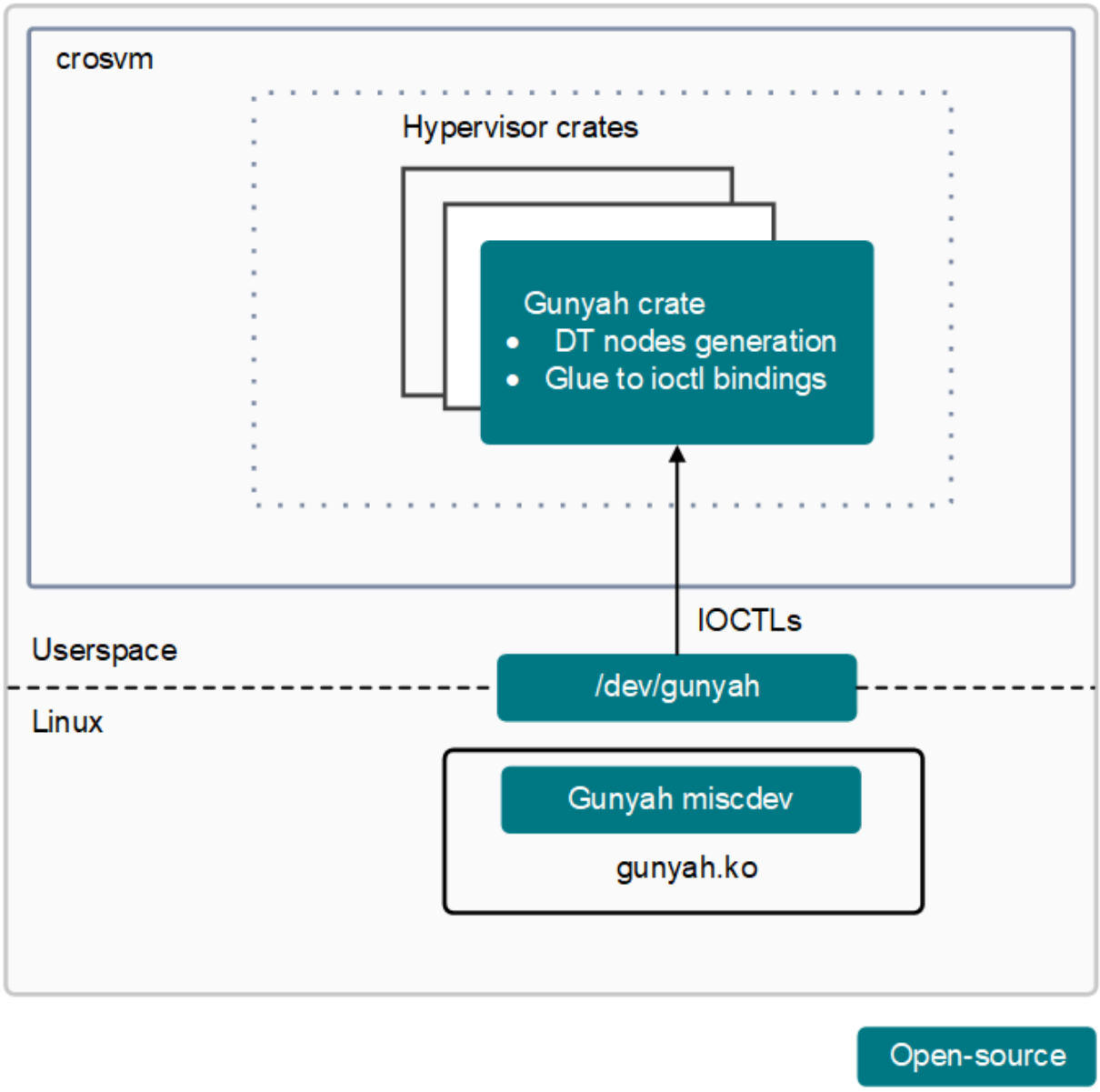

Figure: Qualcomm virtualization solution

For more information about type 1 hypervisor, see gunyah-hypervisor.

Guest VM build support

To compile the guest VM kernel and initrd images, use the following procedure:

During the guest VM kernel compilation, both the guest kernel and the minimal `rootfs` are built. `qcom_vm_defconfig` serves as the base configuration for compiling the guest kernel. The `qcom_vm_debug.config` builds the debug variant of the guest VM.

`crosvm` serves as the VMM and interacts with the Gunyah VM manager in the kernel through the `/dev/gunyah` interface to create VMs.

Build a guest VM

For more information about `crosvm` support, see Create secondary virtual machine.

After the build process completes, certain images are generated. These images get stored in the `<workspace>/build-qcom-wayland/tmp-glibc/deploy/images/<SoC>/qcom-console-image` directory.

To compile the guest VM, run the following command:

```
export SHELL=/bin/bash && MACHINE=SOC DISTRO=qcom-wayland source
setup-environment

bitbake qcom-console-image
```

Guest VM initrd image

- The `svm-initramfs-qcom-image-<SoC>.cpio.gz` file is created for the guest VM. This image contains an initrd image, used during the boot process.
- The `svm-initramfs-qcom-image-<SoC>.cpio.gz` file is generated in the `build-qcom-wayland/tmp-glibc/deploy/images/<SoC>/svm-initramfs-qcom-image` directory.
- On the booted device, the image is stored in the `/var/gunyah/initrd.img` file within the system rootfs.

Guest VM kernel image

- `svm-initramfs-qcom-image-<SoC>.cpio.gz` image is generated for the secondary virtual machine (SVM), which is the guest kernel image.
- The SVM kernel image is generated in the `build-qcom-wayland/tmp-glibc/deploy/images/<SoC>/linux-svm-kernel-qcom` directory.
- On the booted device, the image is in a `/var/gunyah/Image` file within the system rootfs.

Both the `initrd.img` and the SVM kernel images are packed in the `system.img` file. The final `system.img` file is found in the same directory as the other images `<workspace>/build-qcom-wayland/tmp-glibc/deploy/images/<SoC>/qcom-console-image`.

Launch guest virtual machine

Use the `crosvm` VMM and the Gunyah hypervisor to launch the guest virtual machines.

To launch the VM, run the following command:

```
# /usr/bin/crosvm --log-level=debug --no-syslog run --disable-sandbox
--hypervisor \
                                        gunyah --protected-vm-
without-firmware \
                                        --serial=type=stdout,
hardware=virtio-console,console,stdin,num=1 \
                                        --serial=type=stdout,
hardware=serial,earlycon,num=1 \
                                        --initrd /var/gunyah/
initrd.img --no-balloon --no-rng \
                                        --params "rw root=/dev/ram
earlyprintk=serial panic=0" \
                                        /var/gunyah/Image
```

The following are the supported parameters for the `crosvm` command to launch guest VMs:

Table: Supported parameters

Parameters	Description
`/usr/bin/crosvm`	Shows the path of the `crosvm` binary
`--log-level=debug`	Sets the logging level to debug while running `crosvm`

Parameters	Description
`--no-syslog`	Disables syslog
`--disable-sandbox`	Disables sandboxing
`--hypervisor gunyah`	Specifies the hypervisor to be used (in this case, `gunyah`)
`--protected-vm-without-firmware`	Indicates that the VM is protected and doesn't require firmware
`--serial=...`	Configures serial devices for input/output
`--initrd /var/gunyah/initrd.img`	Specifies the initial RAM disk (initrd) image
`--no-balloon`	Disables memory ballooning
`--no-rng`	Disables the entropy source used to seed the guest OS entropy pool
`--params "..."`	Specifies the kernel command-line options
`/var/gunyah/Image`	Specifies the path to the kernel image

View VM and boot logs

The following sample output shows the VM logs:

```
bash-5.1# cat /proc/cpuinfo
processor       : 0
BogoMIPS        : 38.40
Features        : fp asimd evtstrm aes pmull sha1 crc32 atomics cpuid
asimdrdm dcpop asimddp
CPU implementer : 0x00
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0x048
CPU revision    : 0

bash-5.1# cat /proc/meminfo
MemTotal:         161656 kB
MemFree:          124940 kB
MemAvailable:     124032 kB
Buffers:               0 kB
Cached:            19816 kB
SwapCached:            0 kB
Active:               12 kB
```

The following code snippet shows the boot logs:

```
[2022-04-28T17:43:03.290477745+00:00 DEBUG crosvm::crosvm::sys::
linux] creating hypervisor: Gunyah { device: None }
[    0.000000] Booting Linux on physical CPU 0x0000000000
[0x000f0480]
[    0.000000] Linux version 6.6.0 (oe-user@oe-host) (aarch64-qcom-
linux-gcc (GCC) 11.4.0, GNU ld (GNU Binutils) 2.38.20220708) #1 SMP
PREEMPT Wed Feb  7 04:56:59 UTC 2024
[    0.000000] KASLR enabled
[    0.000000] random: crng init done
[    0.000000] Machine model: linux,dummy-virt
[    0.000000] earlycon: uart8250 at MMIO 0x00000000000003f8 (options
'')
[    0.000000] printk: bootconsole [uart8250] enabled
[    0.000000] efi: UEFI not found.
[    0.000000] software IO TLB: Reserved memory: created restricted
DMA pool at 0x0000000090000000, size 64 MiB
[    0.000000] OF: reserved mem: initialized node restricted_dma_
reserved@90000000, compatible id restricted-dma-pool
[    0.000000] OF: reserved mem: 0x0000000090000000..
```

```
0x0000000093ffffff (65536 KiB) map non-reusable restricted_dma_
reserved@90000000
[    0.000000] NUMA: No NUMA configuration found
[    0.000000] NUMA: Faking a node at [mem 0x0000000080000000-
0x0000000093ffffff]
[    0.000000] NUMA: NODE_DATA [mem 0x8ff509c0-0x8ff52fff]
[    0.000000] Zone ranges:
[    0.000000]   DMA      [mem 0x0000000080000000-0x0000000093ffffff]
[    0.000000]   DMA32    empty
[    0.000000]   Normal   empty
[    0.000000] Movable zone start for each node
[    0.000000] Early memory node ranges
[    0.000000]   node   0: [mem 0x0000000080000000-
0x0000000093ffffff]
[    0.000000] Initmem setup node 0 [mem 0x0000000080000000-
0x0000000093ffffff]
[    0.000000] On node 0, zone DMA: 16384 pages in unavailable ranges
[    0.000000] cma: Reserved 32 MiB at 0x000000008d600000 on node -1
[    0.000000] psci: probing for conduit method from DT.
[    0.000000] psci: PSCIv1.1 detected in firmware.
[    0.000000] psci: Using standard PSCI v0.2 function IDs
[    0.000000] psci: MIGRATE_INFO_TYPE not supported.
[    0.000000] psci: SMC Calling Convention v1.3
[    0.000000] percpu: Embedded 31 pages/cpu s86568 r8192 d32216
u126976
[    0.000000] Detected PIPT I-cache on CPU0
[    0.000000] CPU features: detected: GIC system register CPU
interface
[    0.000000] CPU features: detected: Spectre-v4
[    0.000000] CPU features: kernel page table isolation forced ON by
KASLR
[    0.000000] CPU features: detected: Kernel page table isolation
(KPTI)
[    0.000000] alternatives: applying boot alternatives
[    0.000000] Kernel command line: panic=-1 console=hvc0
earlycon=uart8250,mmio,0x3f8 rw root=/dev/ram earlyprintk=serial
panic=0
[    0.000000] Unknown kernel command line parameters
"earlyprintk=serial", will be passed to user space.
[    0.000000] Dentry cache hash table entries: 65536 (order: 7,
524288 bytes, linear)
[    0.000000] Inode-cache hash table entries: 32768 (order: 6,
262144 bytes, linear)
[    0.000000] Fallback order for Node 0: 0
```

```
[    0.000000] Built 1 zonelists, mobility grouping on.  Total pages:
80640
[    0.000000] Policy zone: DMA
[    0.000000] mem auto-init: stack:off, heap alloc:off, heap free:
off
[    0.000000] software IO TLB: area num 1.
[    0.000000] software IO TLB: mapped [mem 0x0000000089600000-
0x000000008d600000] (64MB)
[    0.000000] Memory: 116344K/327680K available (14528K kernel code,
3938K rwdata, 7112K rodata, 6080K init, 571K bss, 178568K reserved,
32768K cma-reserved)
[    0.000000] SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=1,
Nodes=1
[    0.000000] trace event string verifier disabled
[    0.000000] rcu: Preemptible hierarchical RCU implementation.
[    0.000000] rcu:     RCU event tracing is enabled.
[    0.000000] rcu:     RCU restricting CPUs from NR_CPUS=256 to nr_
cpu_ids=1.
[    0.000000]  Trampoline variant of Tasks RCU enabled.
[    0.000000]  Tracing variant of Tasks RCU enabled.
[    0.000000] rcu: RCU calculated value of scheduler-enlistment
delay is 25 jiffies.
[    0.000000] rcu: Adjusting geometry for rcu_fanout_leaf=16, nr_
cpu_ids=1
[    0.000000] NR_IRQS: 64, nr_irqs: 64, preallocated irqs: 0
[    0.000000] GICv3: 988 SPIs implemented
[    0.000000] GICv3: 0 Extended SPIs implemented
[    0.000000] Root IRQ handler: gic_handle_irq
[    0.000000] GICv3: GICv3 features: 16 PPIs
[    0.000000] GICv3: CPU0: found redistributor 0 region 0:
0x000000003ffd0000
[    0.000000] rcu: srcu_init: Setting srcu_struct sizes based on
contention.
[    0.000000] arch_timer: cp15 timer(s) running at 19.20MHz (virt).
[    0.000000] clocksource: arch_sys_counter: mask: 0xffffffffffffff
max_cycles: 0x46d987e47, max_idle_ns: 440795202767 ns
[    0.000000] sched_clock: 56 bits at 19MHz, resolution 52ns, wraps
every 4398046511078ns
[    0.002616] arm-pv: using stolen time PV
[    0.003945] Console: colour dummy device 80x25
[    0.005300] Calibrating delay loop (skipped), value calculated
using timer frequency.. 38.40 BogoMIPS (lpj=76800)
[    0.008475] pid_max: default: 32768 minimum: 301
[    0.009923] LSM: initializing lsm=capability,selinux,integrity
```

```
[    0.011702] SELinux:  Initializing.
[    0.012884] Mount-cache hash table entries: 1024 (order: 1, 8192
bytes, linear)
[    0.015180] Mountpoint-cache hash table entries: 1024 (order: 1,
8192 bytes, linear)
[    0.017809] cacheinfo: Unable to detect cache hierarchy for CPU 0
[    0.019820] RCU Tasks: Setting shift to 0 and lim to 1 rcu_task_
cb_adjust=1.
[    0.022107] RCU Tasks Trace: Setting shift to 0 and lim to 1 rcu_
task_cb_adjust=1.
[    0.024472] rcu: Hierarchical SRCU implementation.
[    0.025891] rcu:     Max phase no-delay instances is 1000.
[    0.027767] EFI services will not be available.
[    0.029190] smp: Bringing up secondary CPUs ...
[    0.030713] smp: Brought up 1 node, 1 CPU
[    0.032002] SMP: Total of 1 processors activated.
[    0.033471] CPU features: detected: 32-bit EL0 Support
[    0.035048] CPU features: detected: Data cache clean to the PoU
not required for I/D coherence
[    0.037742] CPU features: detected: Common not Private
translations
[    0.039678] CPU features: detected: CRC32 instructions
[    0.041275] CPU features: detected: LSE atomic instructions
[    0.043105] CPU features: detected: Privileged Access Never
[    0.044891] CPU: All CPU(s) started at EL1
[    0.046161] alternatives: applying system-wide alternatives
[    0.050071] devtmpfs: initialized
[    0.051500] clocksource: jiffies: mask: 0xffffffff max_cycles:
0xffffffff, max_idle_ns: 7645041785100000 ns
[    0.054767] futex hash table entries: 256 (order: 2, 16384 bytes,
linear)
[    0.057122] pinctrl core: initialized pinctrl subsystem
[    0.058872] DMI not present or invalid.
[    0.060334] NET: Registered PF_NETLINK/PF_ROUTE protocol family
[    0.062534] DMA: preallocated 128 KiB GFP_KERNEL pool for atomic
allocations
[    0.064964] DMA: preallocated 128 KiB GFP_KERNEL|GFP_DMA pool for
atomic allocations
[    0.067353] DMA: preallocated 128 KiB GFP_KERNEL|GFP_DMA32 pool
for atomic allocations
[    0.069943] audit: initializing netlink subsys (disabled)
[    0.071632] audit: type=2000 audit(0.056:1): state=initialized
audit_enabled=0 res=1
[    0.074141] thermal_sys: Registered thermal governor 'step_wise'
```

```
[    0.074143] thermal_sys: Registered thermal governor 'power_
allocator'
[    0.076118] cpuidle: using governor menu
[    0.079536] hw-breakpoint: found 6 breakpoint and 4 watchpoint
registers.
[    0.081693] ASID allocator initialised with 32768 entries
[    0.083662] Serial: AMBA PL011 UART driver
[    0.085582] Modules: 2G module region forced by RANDOMIZE_MODULE_
REGION_FULL
[    0.087812] Modules: 0 pages in range for non-PLT usage
[    0.087813] Modules: 516176 pages in range for PLT usage
[    0.091914] HugeTLB: registered 1.00 GiB page size, pre-allocated
0 pages
[    0.099837] HugeTLB: 0 KiB vmemmap can be freed for a 1.00 GiB
page
[    0.101815] HugeTLB: registered 32.0 MiB page size, pre-allocated
0 pages
[    0.107895] HugeTLB: 0 KiB vmemmap can be freed for a 32.0 MiB
page
[    0.109875] HugeTLB: registered 2.00 MiB page size, pre-allocated
0 pages
[    0.116064] HugeTLB: 0 KiB vmemmap can be freed for a 2.00 MiB
page
[    0.118084] HugeTLB: registered 64.0 KiB page size, pre-allocated
0 pages
[    0.124190] HugeTLB: 0 KiB vmemmap can be freed for a 64.0 KiB
page
[    0.128423] ACPI: Interpreter disabled.
[    0.132206] iommu: Default domain type: Translated
[    0.133734] iommu: DMA domain TLB invalidation policy: strict mode
[    0.135749] SCSI subsystem initialized
[    0.140957] usbcore: registered new interface driver usbfs
[    0.142744] usbcore: registered new interface driver hub
[    0.144353] usbcore: registered new device driver usb
[    0.149976] pps_core: LinuxPPS API ver. 1 registered
[    0.151488] pps_core: Software ver. 5.3.6 - Copyright 2005-2007
Rodolfo Giometti <giometti@linux.it>
[    0.162194] PTP clock support registered
[    0.163436] EDAC MC: Ver: 3.0.0
[    0.164532] scmi_core: SCMI protocol bus registered
[    0.166176] FPGA manager framework
[    0.171350] Advanced Linux Sound Architecture Driver Initialized.
[    0.173213] vgaarb: loaded
[    0.174176] clocksource: Switched to clocksource arch_sys_counter
```

```
[    0.235827] VFS: Disk quotas dquot_6.6.0
[    0.237108] VFS: Dquot-cache hash table entries: 512 (order 0,
4096 bytes)
[    0.239428] pnp: PnP ACPI: disabled
[    0.241483] NET: Registered PF_INET protocol family
[    0.243293] IP idents hash table entries: 8192 (order: 4, 65536
bytes, linear)
[    0.245817] tcp_listen_portaddr_hash hash table entries: 256
(order: 0, 4096 bytes, linear)
[    0.248604] Table-perturb hash table entries: 65536 (order: 6,
262144 bytes, linear)
[    0.251034] TCP established hash table entries: 4096 (order: 3,
32768 bytes, linear)
[    0.253323] TCP bind hash table entries: 4096 (order: 5, 131072
bytes, linear)
[    0.255664] TCP: Hash tables configured (established 4096 bind
4096)
[    0.257584] UDP hash table entries: 256 (order: 1, 8192 bytes,
linear)
[    0.259650] UDP-Lite hash table entries: 256 (order: 1, 8192
bytes, linear)
[    0.261768] NET: Registered PF_UNIX/PF_LOCAL protocol family
[    0.263829] RPC: Registered named UNIX socket transport module.
[    0.265673] RPC: Registered udp transport module.
[    0.267186] RPC: Registered tcp transport module.
[    0.268641] RPC: Registered tcp-with-tls transport module.
[    0.270380] RPC: Registered tcp NFSv4.1 backchannel transport
module.
[    0.272421] PCI: CLS 0 bytes, default 64
[    0.273680] Unpacking initramfs...
[    0.278241] kvm [1]: HYP mode not available
[    0.279783] Initialise system trusted keyrings
[    0.286209] workingset: timestamp_bits=42 max_order=16 bucket_
order=0
[    0.288474] squashfs: version 4.0 (2009/01/31) Phillip Lougher
[    0.294313] NFS: Registering the id_resolver key type
[    0.295881] Key type id_resolver registered
[    0.297197] Key type id_legacy registered
[    0.302225] nfs4filelayout_init: NFSv4 File Layout Driver
Registering...
[    0.304407] nfs4flexfilelayout_init: NFSv4 Flexfile Layout Driver
Registering...
[    0.310289] 9p: Installing v9fs 9p2000 file system support
[    0.320607] NET: Registered PF_ALG protocol family
```

```
[    0.322022] Key type asymmetric registered
[    0.330205] Asymmetric key parser 'x509' registered
[    0.331833] Block layer SCSI generic (bsg) driver version 0.4
loaded (major 245)
[    0.334140] io scheduler mq-deadline registered
[    0.342201] io scheduler kyber registered
[    0.343459] io scheduler bfq registered
[    0.353648] pci-host-generic 10000.pci: assigned reserved memory
node restricted_dma_reserved@90000000
[    0.362220] pci-host-generic 10000.pci: host bridge /pci ranges:
[    0.364055] pci-host-generic 10000.pci:      MEM 0x0002000000..
0x0003ffffff -> 0x0002000000
[    0.370193] pci-host-generic 10000.pci:      MEM 0x0094800000..
0xfffffffffff -> 0x0094800000
[    0.372927] pci-host-generic 10000.pci: Memory resource size
exceeds max for 32 bits
[    0.382221] PCI: OF: PROBE_ONLY enabled
[    0.383535] pci-host-generic 10000.pci: ECAM at [mem 0x00010000-
0x0100ffff] for [bus 00]
[    0.386134] pci-host-generic 10000.pci: PCI host bridge to bus
0000:00
[    0.394279] pci_bus 0000:00: root bus resource [bus 00]
[    0.395912] pci_bus 0000:00: root bus resource [mem 0x02000000-
0x03ffffff]
[    0.397972] pci_bus 0000:00: root bus resource [mem 0x94800000-
0xfffffffffff]
[    0.406329] pci 0000:00:00.0: [8086:1237] type 00 class 0x060000
[    0.409065] pci 0000:00:01.0: [1af4:1043] type 00 class 0x00ff00
[    0.419597] pci 0000:00:01.0: reg 0x10: [mem 0x02000000-
0x02007fff]
[    0.425686] Freeing initrd memory: 6464K
[    0.428105] pci 0000:00:01.0: PME# supported from D0 D3hot D3cold
[    0.430575] pci 0000:00:02.0: [1b36:0011] type 00 class 0xffff00
[    0.432661] pci 0000:00:02.0: reg 0x10: [mem 0x02008000-
0x0200800f]
[    0.436076] pci 0000:00:00.0: Limiting direct PCI/PCI transfers
[    0.438471] virtio-pci 0000:00:01.0: assigned reserved memory node
restricted_dma_reserved@90000000
[    0.443123] Serial: 8250/16550 driver, 4 ports, IRQ sharing
enabled
[    0.445534] 2e8.U6_16550A: ttyS0 at MMIO 0x2e8 (irq = 14, base_
baud = 115200) is a 16550A
[    0.448422] 2f8.U6_16550A: ttyS1 at MMIO 0x2f8 (irq = 14, base_
baud = 115200) is a 16550A
```

```
[    0.451159] 3e8.U6_16550A: ttyS2 at MMIO 0x3e8 (irq = 15, base_
baud = 115200) is a 16550A
[    0.453983] 3f8.U6_16550A: ttyS3 at MMIO 0x3f8 (irq = 15, base_
baud = 115200) is a 16550A
[    0.456935] msm_serial: driver initialized
[    0.460508] printk: console [hvc0] enabled
[    0.461772] printk: bootconsole [uart8250] disabled
[    0.475305] loop: module loaded
[    0.475957] megasas: 07.725.01.00-rc1
[    0.476374] tun: Universal TUN/TAP device driver, 1.6
[    0.477187] VFIO - User Level meta-driver version: 0.3
[    0.477691] usbcore: registered new interface driver usb-storage
[    0.478075] i2c_dev: i2c /dev entries driver
[    0.478575] sdhci: Secure Digital Host Controller Interface driver
[    0.479311] sdhci: Copyright(c) Pierre Ossman
[    0.479612] sdhci-pltfm: SDHCI platform and OF driver helper
[    0.479932] ledtrig-cpu: registered to indicate activity on CPUs
[    0.480343] SMCCC: SOC_ID: ARCH_SOC_ID not implemented, skipping .
...
[    0.480864] usbcore: registered new interface driver usbhid
[    0.481329] usbhid: USB HID core driver
[    0.482042] NET: Registered PF_PACKET protocol family
[    0.482342] 9pnet: Installing 9P2000 support
[    0.482635] Key type dns_resolver registered
[    0.484383] registered taskstats version 1
[    0.484775] Loading compiled-in X.509 certificates
[    0.486896] page_owner is disabled
[    0.487629] Key type .fscrypt registered
[    0.487943] Key type fscrypt-provisioning registered
[    0.488391] clk: Disabling unused clocks
[    0.488759] ALSA device list:
[    0.489050]   No soundcards found.
[    0.490311] Freeing unused kernel memory: 6080K
[    0.498264] Run /init as init process
Starting version 250.5+
bash: cannot set terminal process group (-1): Inappropriate ioctl for
device
bash: no job control in this shell
bash-5.1#
```

Launch a VM with two CPUs and default RAM

To launch a guest VM with two CPUs, use the `--cpus num-cores=2` parameter:

```
#/usr/bin/crosvm --log-level=debug --no-syslog run --disable-sandbox
--hypervisor \
                                          gunyah --protected-vm-
without-firmware \
                                          --cpus num-cores=2 \
                                          --serial=type=stdout,
hardware=virtio-console,console,stdin,num=1 \
                                          --serial=type=stdout,
hardware=serial,earlycon,num=1 \
                                          --initrd /var/gunyah/
initrd.img --no-balloon --no-rng \
                                          --params "rw root=/dev/ram
earlyprintk=serial panic=0" \
                                          /var/gunyah/Image
```

The following code snippet shows the VM logs:

```
bash-5.1# cat /proc/cpuinfo
processor       : 0
BogoMIPS        : 38.40
Features        : fp asimd evtstrm aes pmull sha1 crc32 atomics cpuid
asimdrdm dcpop asimddp
CPU implementer : 0x00
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0x048
CPU revision    : 0

processor       : 1
BogoMIPS        : 38.40
Features        : fp asimd evtstrm aes pmull sha1 crc32 atomics cpuid
asimdrdm dcpop asimddp
CPU implementer : 0x00
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0x048
CPU revision    : 0

bash-5.1# cat /proc/meminfo
MemTotal:         161656 kB
MemFree:          124940 kB
```

```
MemAvailable:     124032 kB
Buffers:               0 kB
Cached:            19816 kB
SwapCached:            0 kB
Active:               12 kB
```

Launch a VM with two CPUs and 512 MB RAM

To launch a guest VM with two CPUs and 512 MB RAM, use the `--cpus num-cores=2 `` and ``--mem size=512` parameters.

```
#/usr/bin/crosvm --log-level=debug --no-syslog run --disable-sandbox
--hypervisor \
                                          gunyah --protected-vm-
without-firmware \
                                          --cpus num-cores=2 --mem
size=512      \
                                          --serial=type=stdout,
hardware=virtio-console,console,stdin,num=1 \
                                          --serial=type=stdout,
hardware=serial,earlycon,num=1              \
                                          --initrd /var/gunyah/
initrd.img --no-balloon --no-rng            \
                                          --params "rw root=/dev/ram
earlyprintk=serial panic=0"           \
                                          /var/gunyah/Image
```

The following code snippet shows the VM logs:

```
bash-5.1# cat /proc/cpuinfo
processor       : 0
BogoMIPS        : 38.40
Features        : fp asimd evtstrm aes pmull sha1 crc32 atomics cpuid
asimdrdm dcpop asimddp
CPU implementer : 0x00
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0x048
CPU revision    : 0

processor       : 1
BogoMIPS        : 38.40
Features        : fp asimd evtstrm aes pmull sha1 crc32 atomics cpuid
asimdrdm dcpop asimddp
```

```
CPU implementer : 0x00
CPU architecture: 8
CPU variant     : 0x0
CPU part        : 0x048
CPU revision    : 0

bash-5.1# cat /proc/meminfo
MemTotal:         418304 kB
MemFree:          378356 kB
MemAvailable:     376048 kB
Buffers:               0 kB
Cached:            20308 kB
SwapCached:            0 kB
Active:                0 kB
```

Launch a VM with block device

To launch the VM, run the following command:

```
#/usr/bin/crosvm --log-level=debug --no-syslog run --disable-sandbox
--hypervisor \
                                          gunyah --protected-vm-
without-firmware \
                                          --block disk.img \
                                          --cpus num-cores=2 --mem
size=256      \
                                          --serial=type=stdout,
hardware=virtio-console,console,stdin,num=1 \
                                          --serial=type=stdout,
hardware=serial,earlycon,num=1              \
                                          --initrd /var/gunyah/
initrd.img --no-balloon --no-rng            \
                                          --params "rw root=/dev/ram
earlyprintk=serial panic=0"           \
                                          /var/gunyah/Image
```

- Use the `--block` flag to specify the block device name (in this case, disk.img).
- This flag creates `/dev/vda`, `/dev/vdb`, and `/dev/vd*` nodes in the guest VM.

```
bash-5.1# ls -al /dev/block/
total 0
drwxr-xr-x    2 root     root           220 Jan  1 00:00 .
drwxr-xr-x    9 root     root          2980 Jan  1 00:00 ..
```

```
lrwxrwxrwx    1 root     root             6 Jan  1 00:00 254:0 -> ../
vda
lrwxrwxrwx    1 root     root             8 Jan  1 00:00 7:0 -> ../
loop0
lrwxrwxrwx    1 root     root             8 Jan  1 00:00 7:1 -> ../
loop1
```

To mount the device in the guest VM, run the following command: `mount -t ext4 /dev/vda /mount-point`.

```
bash-5.1# mkdir /data
bash-5.1# mount -t ext4 /dev/vda /data
[  112.311376] EXT4-fs (vda): mounted filesystem 4b577765-65ad-4fd8-
910c-0eac437d8afb r/w with ordered data mode. Quota mode: none.
```

KVM overview

A kernel-based virtual machine (KVM) is an open-source virtualization module integrated into the Linux kernel. This integration allows the KVM to act as a hypervisor. KVM facilitates hardware-assisted virtualization for guest operating systems.

Note: KVM is supported on QCS8275 and QCS9075 SoCs only.

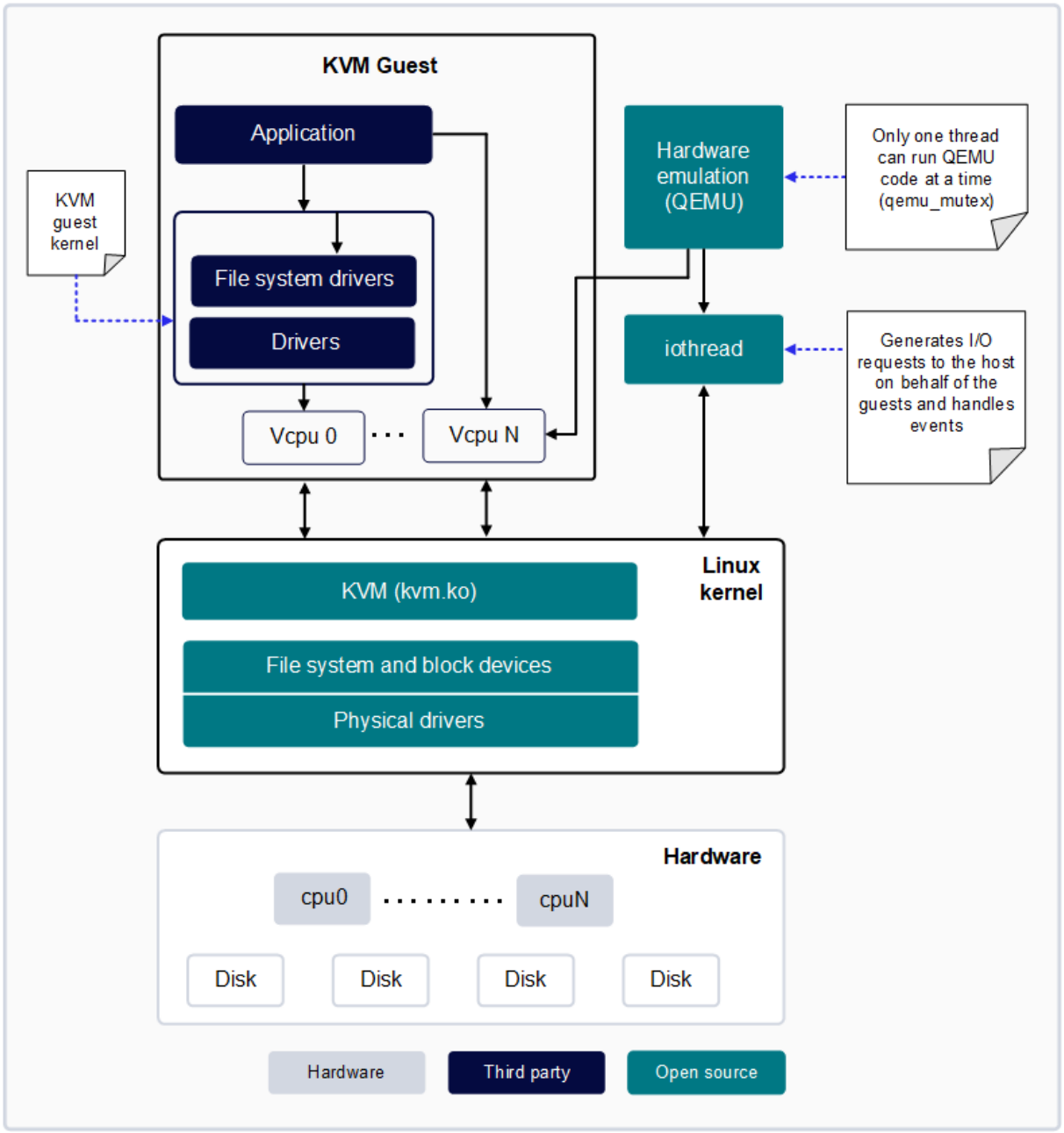

Figure: High-level overview of KVM/QEMU virtualization

The following are the prerequisites to enable KVM Hypervisor on Arm CPU-based systems:

- The Boot loader starts the Linux kernel in the exception level 2 (EL2)
- `CONFIG_KVM` is enabled in the kernel configuration

To verify if KVM is available on the device, run the following command to verify the presence of `/dev/kvm`:

```
ls -l /dev/kvm
```

In the Qualcomm Linux default boot flow, the Linux Kernel starts in EL1 while Gunyah assumes the role of the Hypervisor. For instructions on booting the Linux Kernel in EL2 to enable the KVM Hypervisor, see UEFI.

Consider the following when enabling KVM:

- Virtual host extensions (VHE) are enabled by default on all the Qualcomm Linux SoCs that support Arm (v8.1) or later instruction sets. This configuration allows the entire Linux Kernel to run at EL2 with slight modifications. The guest OS kernel and user space run at EL1 and EL0. With VHE, transitions between the guest and host incur less penalty because the guest and host kernels operate at different exception levels.
- The Linux Kernel manages all accessible non-secure memory. It doesn't support protected use cases that require specific isolation from the Linux kernel. The Gunyah hypervisor controls the S2 page tables of the Linux Kernel in the default boot flow.
- The peripheral image loading (PIL) services aren't supported. Hence, any applications that require aDSP, cDSP, and neural signal processor (NSP) don't work when KVM is enabled.
- The power state coordination interface (PSCI) for KVM is supported on the Linux kernel. PSCI ensures that CPU hotplug and low-power modes (LPM) function correctly with KVM.

Virtual machine manager

Quick emulator (QEMU) is used as the virtual machine manager (VMM) for virtualization. Libvirt acts as a management layer that interacts with QEMU to start, stop, and manage virtual machines.

QEMU

Use QEMU as a VMM with the KVM to run VMs at near-native performance. QEMU supports emulated devices, for example, virtio devices, that provide high-performance I/O operations and allows access to USB and peripheral component interconnect (PCI) devices from the VM. The QEMU process and its threads manage a single VM. Invoke QEMU directly to create the VMs. For more information, see QEMU's.

Libvirt

Libvirt is a suite of tools, including an API library, a daemon (libvirtd), and a command-line utility (virsh), for managing VMs. The virsh utility is useful for managing multiple VMs. For more information, see virsh.

Launch guest VM

To launch an ARM64-based VM, use QEMU or virsh commands that internally work with libvirt. The commands allow you to configure the VM with CPU, memory, and storage. Specify the number of virtual CPUs and the amount of memory allocated to the VM. Additionally, the VM can boot with a ramdisk (initrd) and a `.ext4` root file system.

Note: Verify that the guest kernel image (`Image`), root filesystem CPIO (`rootfs.cpio.gz`), and root filesystem image (`rootfs.ext4`) are present in the `/mnt/overlay/guest` directory in the host filesystem before launching the guest. These file paths and formats are examples for reference.

QEMU

To bring up the VMs with various configurations, run the following commands:

Boot with ramdisk

```
qemu-system-aarch64 \
   -M virt -m 2G \
   -kernel /mnt/overlay/guest/Image \
   -initrd /mnt/overlay/guest/rootfs.cpio.gz \
   -cpu host --enable-kvm -smp 4 -nographic
```

Boot with rootfs image

```
qemu-system-aarch64 \
   -M virt -m 2G \
   -kernel /mnt/overlay/guest/Image \
   -drive file=/mnt/overlay/guest/rootfs.ext4,if=virtio,format=raw \
   -append "root=/dev/vda" \
   -cpu host --enable-kvm -smp 4 -nographic
```

Libvirt

Libvirt enables you to use QEMU and KVM to create, interact, and manage VMs.

VM management with virsh

virsh provides a range of commands to create, control, and monitor VMs.

The following are the virsh commands used for VM management:

```
# Define a new VM domain called initrd_simple
virsh define /mnt/overlay/libvirt_initrd_simple.xml
```

```
# List all domains and check the state of initrd_simple, it is not
started yet
virsh list --all
```

```
# start the initrd_simple VM
virsh start initrd_simple
```

```
# connect to console
virsh console initrd_simple
```

```
# Disconnect from console by pressing (Ctrl + ]) key combination
```

```
# shutdown the VM
```

```
virsh shutdown initrd_simple
```

```
# Undefine the VM if there is no intention to restart/resue it
virsh undefine initrd_simple
```

Boot with ramdisk

Copy the following XML content to `/mnt/overlay/guest/libvirt_initrd_simple.xml` on the host:

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/
qemu/1.0'>
  <name>simple_initrd</name>
   <!-- specify VM memory in KB  -->
  <memory unit='KiB'>2097152</memory>
```

```
   <!-- 4 vCPUs affined to all host CPUs  -->
  <vcpu placement='static'>4</vcpu>
  <resource>
    <partition>/machine</partition>
  </resource>
  <os>
    <type arch='aarch64' machine='virt-6.2'>hvm</type>
    <!-- specify kernel/initrd file  -->
    <kernel>/mnt/overlay/guest/Image</kernel>
    <initrd>/mnt/overlay/guest/rootfs.cpio.gz</initrd>
    <boot dev='hd'/>
  </os>
  <features>
    <gic version='3'/>
  </features>
  <cpu mode='host-passthrough' check='none'/>
  <devices>
    <!-- Guest console will be available over pty on the host -->
    <console type='pty'>
    </console>
  </devices>
</domain>
```

Boot with rootfs image

Copy the following XML file to the host file system in the `/mnt/overlay/guest/libvirt_rootfs_simple.xml` file. Use the VM management commands with `rootfs_simple` as VM domain/name instead of `initrd_simple`.

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/
qemu/1.0'>
  <name>simple_rootfs</name>
  <memory unit='KiB'>2097152</memory>
  <vcpu placement='static'>4</vcpu>
  <resource>
    <partition>/machine</partition>
  </resource>
  <os>
    <type arch='aarch64' machine='virt-6.2'>hvm</type>
    <kernel>/mnt/overlay/guest/Image</kernel>
    <cmdline>root=/dev/vda</cmdline>
    <boot dev='hd'/>
  </os>
```

```
  <features>
    <gic version='3'/>
  </features>
  <cpu mode='host-passthrough' check='none'/>
  <devices>
    <console type='pty'>
    </console>
    <disk type="file" device="disk">
      <driver name="qemu" type="raw"/>
      <!-- specify rootfs image file  -->
      <source file="/mnt/overlay/guest/rootfs.ext4"/>
      <target dev="vda" bus="virtio"/>
    </disk>
  </devices>
</domain>
```

Virtio framework

Virtio abstracts devices in a paravirtualized hypervisor environment. It provides an I/O paravirtualization framework to interact with the paravirtualized (paravirt) devices. The virtual machine monitor (VMM) or hypervisor (HYP) emulates most of the devices exposed to virtual machines (VMs) using virtio.

Key features

The following are the key features of virtual network:

- Paravirtualization and full virtualization
  - Full virtualization: In a fully virtual environment, the guest VM doesn't recognize the hypervisor, it runs without modifications. However, full virtualization results in higher overhead due to the device emulation.
  - Paravirtualization: In a paravirtualize environment, the VM recognizes the hypervisor and requires modifications to the OS. Paravirtualization allows efficient communication between the guest and the host.
- Virtio architecture

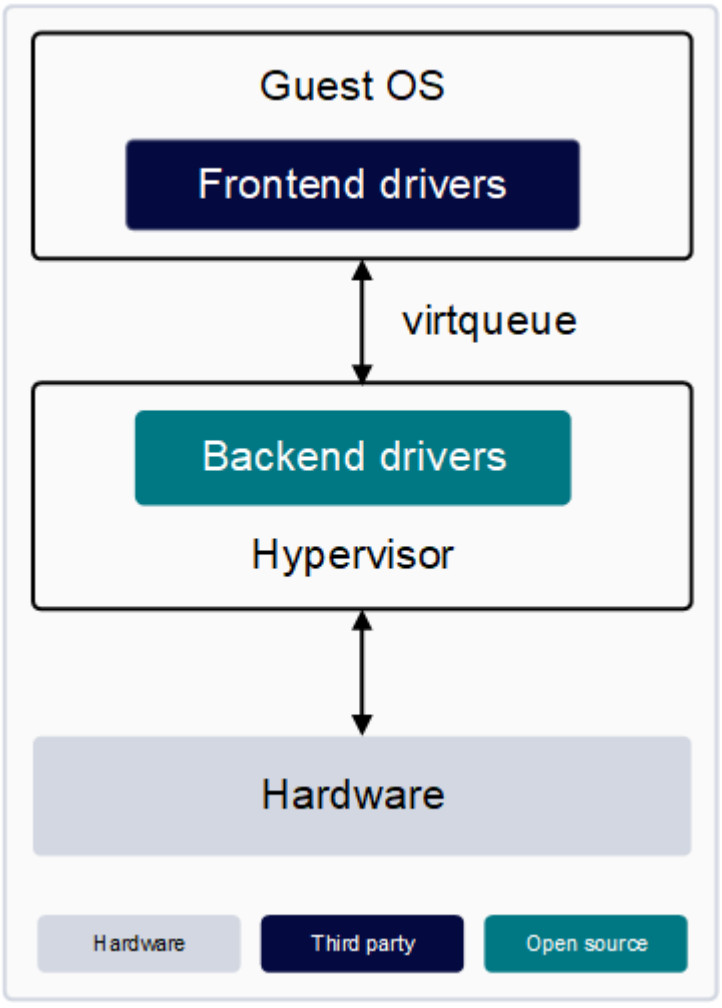

Figure : A high-level overview of the virtio architecture

  - Front-end drivers: Implements in the guest OS. The front-end drivers interact

with the back-end drivers in the hypervisor

  - Back-end drivers: Implements in the VMM/HYP. The back-end drivers handle the actual device emulation, and interacts with the front-end drivers through virtual queues
  - Virtual queues: Virtio uses virtual queues (virtqueues) to facilitate communication between front-end and back-end drivers. These queues are implemented as rings to manage guest-to-hypervisor transitions efficiently.

- Vhost

  Vhost is a protocol that offloads the virtio data plane implementation to another element (user process or kernel module) to enhance performance. This offloading reduces the overhead of context switching between the guest VM and the hypervisor.

  - Vhost-net: A kernel-level implementation that allows the data plane to bypass the QEMU process, to reduce latency and improves performance. For more information about v-host-net, see Introduction to virtio-networking and vhost-net.
  - Vhost-user: A user-space implementation that handles the data plane with a separate process, to provide flexibility and better performance for workloads. For more information about v-host-user, see Vhost-user Protocol.

Benefits

The following are the advantages of using virtio framework:

- Standardization: Virtio provides a common interface for device emulation. It promotes code reuse and efficiency across different virtualization platforms.
- Flexibility: Virtio supports block devices and network devices.

Virtio interfaces

The following are the virtio-supported interfaces:

9P transport overview

`virtio-9p` enables you to share files between the host and the VMs under the 9P (Plan 9 file system) protocol to enhance the performance within the virtio framework.

Note: Ensure that the following configurations are enabled in the host and guest kernels:

```
Host side: CONFIG_NET_9P, CONFIG_NET_9P_VIRTIO
```

```
Guest side: CONFIG_NET_9P, CONFIG_NET_9P_VIRTIO, CONFIG_9P_FS
```

To create a `virtio-9p` shared directory, define it on the host, and configure the guest VM to mount it. See the following example:

```
root@qcs9100-ride-sx:~# ls -la /mnt/overlay/test_dir
total 12
drwxr-xr-x. 2 root root 4096 Apr 29 21:36 .
drwxr-xr-x. 7 root root 4096 Apr 29 21:36 ..
-rw-r--r--. 1 root root    8 Apr 29 21:36 file.txt
root@qcs9100-ride-sx:~# cat /mnt/overlay/test_dir/file.txt
testing
root@qcs9100-ride-sx:~#
```

Create a `/mnt/overlay/test_dir` directory on the host to share it with the guest VM. Pass this information either in the libvirt XML configuration or as QEMU arguments. To configure the 9P transport, run the following command:

```
qemu-system-aarch64 \
-M virt -m 2G \
-kernel /mnt/overlay/guest/Image \
-drive file=/mnt/overlay/guest/rootfs.ext4,if=virtio,format=raw \
-append "root=/dev/vda" \
-cpu host --enable-kvm -smp 4 -nographic \
-fsdev local,id=fsdev0,path=/mnt/overlay/test_dir,security_
model=passthrough \
-device virtio-9p-pci,fsdev=fsdev0,mount_tag=hostshare
```

To identify and mount the shared directory on the guest VM, use the `fsdev0` and `hostshare` filesystem devices.

To launch the Libvirt guest VM from the libvirt interface with the following sample XML, see Libvirt.

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/
qemu/1.0'>
```

```
  <name>simple_9p</name>
  <memory unit='KiB'>2097152</memory>
  <vcpu placement='static'>4</vcpu>
  <resource>
    <partition>/machine</partition>
  </resource>
  <os>
    <type arch='aarch64' machine='virt-6.2'>hvm</type>
    <kernel>/mnt/overlay/guest/Image</kernel>
    <cmdline>root=/dev/vda</cmdline>
    <boot dev='hd'/>
  </os>
  <features>
    <gic version='3'/>
  </features>
  <cpu mode='host-passthrough' check='none'/>
  <devices>
    <console type='pty'>
    </console>
    <disk type="file" device="disk">
      <driver name="qemu" type="raw"/>
        <!-- Specify rootfs image file here -->
      <source file="/mnt/overlay/guest/rootfs.ext4"/>
      <target dev="vda" bus="virtio"/>
    </disk>
    <filesystem type='mount' accessmode='mapped' fmode='644' dmode=
'755'>
      <!-- Specify the directory to be shared -->
      <source dir='/mnt/overlay/test_dir'/>
      <!-- Specify the mount_tag to identify the mount point -->
      <target dir='hostshare'/>
    </filesystem>
  </devices>
</domain>
```

Note: Copy the XML content to `/mnt/overlay/guest/libvirt_virtio_9p.xml` file in the host.

After the guest VM boots up, verify if the virtio device is probed successfully.

```
root@v8a-arm64:~# lspci
00:00.0 Host bridge: Red Hat, Inc. QEMU PCIe Host bridge
00:01.0 PCI bridge: Red Hat, Inc. QEMU PCIe Root port
```

```
00:01.1 PCI bridge: Red Hat, Inc. QEMU PCIe Root port
00:01.2 PCI bridge: Red Hat, Inc. QEMU PCIe Root port
01:00.0 Unclassified device [0002]: Red Hat, Inc. Virtio 1.0
filesystem (rev 01)
02:00.0 SCSI storage controller: Red Hat, Inc. Virtio 1.0
block device (rev 01)
root@v8a-arm64:~#root@v8a-arm64:~# cat /sys/bus/pci/devices/
0000\:01\:00.0/virtio0/uevent
DRIVER=9pnet_virtio
MODALIAS=virtio:d00000009v00001AF4
root@v8a-arm64:~#
```

To mount the shared directory in the guest VM, run the following command:

```
mount -t 9p -o trans=virtio hostshare mountpoint
```

VSOCK overview

VSOCK is a virtual socket interface that allows communication between VMs and the host OS. It's used in KVM and QEMU. VSOCK is supported through a virtio interface. virtio-vsock is a vhost-based virtio device where the host kernel manages all the data transfer while the KVM hypervisor controls the information.

To create a VSOCK connection in a KVM and QEMU environment, specify the context identifier (CID) and port the number. The CID is a unique identifier assigned to each VM in a VSOCK environment, which is used to route communication between the host and the VMs. The host has a CID of 2 while VMs are assigned CIDs starting from 3 and above.

The following table lists the CID values:

Table: CID values

CID	Description
-1	Any address for binding
0	Hypervisor
1	Loopback
2	Host

Note: Ensure that the following configurations are enabled in the host and guest kernels.

```
Host side: CONFIG_VSOCKETS, CONFIG_VHOST_VSOCK
```

```
Guest side: CONFIG_VSOCKETS, CONFIG_VIRTIO_VSOCKETS
```

- Enable VSOCK device: To enable the VSOCK device, specify the device in the VM configuration file (libvirt XML), or pass an argument to QEMU.

```
qemu-system-aarch64 \
-machine virt -m 2G \
-kernel /mnt/overlay/guest/Image \
-drive file=/mnt/overlay/guest/rootfs.ext4,if=virtio,format=raw
\
-append "root=/dev/vda" \
-cpu host --enable-kvm -smp 4 -nographic \
-device vhost-vsock-pci,guest-cid=73
```

In this command, `guest-cid=73` specifies the CID for the VM.

- Launch the guest VM using the libvirt interface: To use the VSOCK device with libvirt, define the VSOCK device in the VM XML configuration file.

  To launch the guest VM using the libvirt interface, see Libvirt.

Copy the XML content to `/mnt/overlay/guest/libvirt_virtio_vsock.xml` on the host. After the Guest VM boots up, verify that the device is probed successfully.

```
root@v8a-arm64:~# lspci
00:00.0 Host bridge: Red Hat, Inc. QEMU PCIe Host bridge
00:01.0 PCI bridge: Red Hat, Inc. QEMU PCIe Root port
00:01.1 PCI bridge: Red Hat, Inc. QEMU PCIe Root port
00:01.2 PCI bridge: Red Hat, Inc. QEMU PCIe Root port
01:00.0 SCSI storage controller: Red Hat, Inc. Virtio 1.0
block device (rev 01)
02:00.0 Communication controller: Red Hat, Inc. Virtio 1.0
socket (rev 01)
root@v8a-arm64:~#
root@v8a-arm64:~# cat /sys/bus/pci/devices/0000\:02\:00.0/
virtio1/uevent
DRIVER=vmw_vsock_virtio_transport
MODALIAS=virtio:d00000013v00001AF4
root@v8a-arm64:~#
```

To verify the host-to-guest and guest-to-host interaction, use the `socat` utility. Ensure that the `socat` utility is supported on host and guest.

- On the host, verify the following port:

```
socat STDIN VSOCK-LISTEN:1234
```

- On the guest, connect to the host using the host CID (2):

```
socat STDOUT VSOCK-CONNECT:2:1234
```

- In this setup, the guest and host show the same values.

Virtio block overview

The virtio block is a standardized way to present block devices to the VM. Each virtio block device appears as a disk inside the guest VM. The virtio block allows the VMs to read and write operations.

The following are the key features of the virtio-block device:

- Simplicity: Easy to implement and use
- Performance: Designed to minimize overhead and maximize throughput
- Flexibility: Designed to use with various types of storage back ends

Note: Ensure that `CONFIG_VIRTIO_BLK` is enabled in the guest kernel.

To use a virtio-block device, do the following:

1. To configure the VM for including a virtio-block device, specify the device in the VM configuration file (libvirt XML), or pass an argument to QEMU.
   a. To enable the virtio-block device in the VM, use the following QEMU command:

```
qemu-system-aarch64 \
-M virt -m 2G \
-kernel /mnt/overlay/guest/Image \
-drive file=/mnt/overlay/guest/rootfs.ext4,if=virtio,
format=raw \
-append "root=/dev/vda" \
-cpu host --enable-kvm -smp 4 -nographic \
-drive file=/mnt/overlay/guest/disk.img,if=virtio,format=raw
```

   In this command, `rootfs.ext4` and `disk.img` are the disks inside the guest VM.

   b. Launch the guest VM using the libvirt interface: To launch the guest VM with the following sample XML, see Libvirt:

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/
domain/qemu/1.0'>
  <name>simple_block_device</name>
  <memory unit='KiB'>2097152</memory>
```

```
  <vcpu placement='static'>4</vcpu>
  <resource>
    <partition>/machine</partition>
  </resource>
  <os>
    <type arch='aarch64' machine='virt-6.2'>hvm</type>
    <kernel>/mnt/overlay/guest/Image</kernel>
    <cmdline>root=/dev/vda</cmdline>
    <boot dev='hd'/>
  </os>
  <features>
    <gic version='3'/>
  </features>
  <cpu mode='host-passthrough' check='none'/>
  <devices>
    <console type='pty'>
    </console>
    <disk type="file" device="disk">
      <driver name="qemu" type="raw"/>
      <!-- specify rootfs image file  -->
      <source file="/mnt/overlay/guest/rootfs.ext4"/>
      <target dev="vda" bus="virtio"/>
    </disk>
    <disk type="file" device="disk">
      <driver name="qemu" type="raw"/>
      <!-- specify a different disk image file -->
      <source file="/mnt/overlay/guest/disk.img"/>
      <target dev="vdb" bus="virtio"/>
    </disk>
  </devices>
</domain>
```

Note: Copy the XML content to `/mnt/overlay/guest/libvirt_virtio_blk.xml` on the host.

2. The guest OS detects the virtio-block device and initializes it. In this process, the guest driver communicates with the device to set up the necessary data structures and queues.

```
root@v8a-arm64:~# dmesg | grep virtio
[    0.225425] virtio-pci 0000:01:00.0: enabling device
(0000 -> 0002)
[    0.227073] virtio-pci 0000:02:00.0: enabling device
(0000 -> 0002)
```

```
[    0.234012] virtio_blk virtio0: 4/0/0 default/read/
poll queues
[    0.235405] virtio_blk virtio0: [vda] 2220734 512-byte
logical blocks (1.14 GB/1.06 GiB)
[    0.241213] virtio_blk virtio1: 4/0/0 default/read/
poll queues
[    0.242418] virtio_blk virtio1: [vdb] 8192000 512-byte
logical blocks (4.19 GB/3.91 GiB)
root@v8a-arm64:~#
root@v8a-arm64:~# lspci
00:00.0 Host bridge: Red Hat, Inc. QEMU PCIe Host bridge
00:01.0 PCI bridge: Red Hat, Inc. QEMU PCIe Root port
00:01.1 PCI bridge: Red Hat, Inc. QEMU PCIe Root port
00:01.2 PCI bridge: Red Hat, Inc. QEMU PCIe Root port
01:00.0 SCSI storage controller: Red Hat, Inc. Virtio 1.0
block device (rev 01)
02:00.0 SCSI storage controller: Red Hat, Inc. Virtio 1.0
block device (rev 01)
root@v8a-arm64:~#
```

3. After initializing, you can use the virtio-block and other block devices. The guest OS performs read and write operations on the virtio driver and passes the operations to the host for processing.
4. To manage the virtio-block device, use the standard tools and commands available in the guest OS. For example, `lsblk` and `df`.

Virtio-IOMMU overview

Virtio-IOMMU is a paravirtualized input/output memory management unit (IOMMU) that provides direct memory access (DMA) management in the virtual environments. Virtio-IOMMU integrates with the existing software APIs such as virtual function I/O (VFIO) and removes the need for page table emulation, making it a lightweight and high-performance solution. Virtio-IOMMU acts as a proxy for physical IOMMUs and manages the devices assigned to the guest. As virtual IOMMU, it manages, emulates, and paravirtualizes the devices.

The following are the key features of Virtio-IOMMU:

- Paravirtualization: Leverages existing transport mechanisms and reduces overhead
- Flexibility: Supports PCI passthrough and shares virtual memory
- Integration: Integrates with software APIs and enhances compatibility. For example, VFIO

Note: Ensure that `CONFIG_VIRTIO_IOMMU` is enabled in the guest kernel.

To use a Virtio-IOMMU device, do the following:

1. To configure the VM for including a Virtio-IOMMU device, specify the device in the VM configuration file (XML), or pass an argument to QEMU.
   a. Enable a Virtio-IOMMU device: To enable the Virtio-IOMMU device in the VM, use the following QEMU command:

```
qemu-system-aarch64 \
-M virt -m 2G \
-kernel /mnt/overlay/guest/Image \
-drive file=/mnt/overlay/guest/rootfs.ext4,if=virtio,
format=raw \
-append "root=/dev/vda" \
-cpu host --enable-kvm -smp 4 -nographic \
-device virtio-iommu-pci
```

   b. Launch guest VM using the libvirt interface: To launch the guest VM with the following sample XML, see Libvirt.

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/
schemas/domain/qemu/1.0'>
  <name>simple_iommu</name>
  <memory unit='KiB'>2097152</memory>
  <vcpu placement='static'>4</vcpu>
  <resource>
    <partition>/machine</partition>
  </resource>
```

```
  <os>
    <type arch='aarch64' machine='virt-6.2'>hvm</
type>
    <kernel>/mnt/overlay/guest/Image</kernel>
    <cmdline>root=/dev/vda</cmdline>
    <boot dev='hd'/>
  </os>
  <features>
    <gic version='3'/>
  </features>
  <cpu mode='host-passthrough' check='none'/>
  <devices>
    <controller type='pci' index='0' model='pcie-root
'>
      <alias name='pcie.0'/>
    </controller>
    <console type='pty'>
    </console>
    <disk type="file" device="disk">
      <driver name="qemu" type="raw"/>
      <!-- specify rootfs image file  -->
      <source file="/mnt/overlay/guest/rootfs.ext4"/>
      <target dev="vda" bus="virtio"/>
    </disk>
  </devices>
  <qemu:commandline>
    <qemu:arg value="-device"/>
    <qemu:arg value='{"driver":"virtio-iommu-pci",
"bus":"pcie.0","addr":"0x3"}'/>
  </qemu:commandline>
</domain>
```

Note: Copy the XML content to `/mnt/overlay/guest/libvirt_virtio_iommu.xml` on the host.

2. The guest OS detects the Virtio-IOMMU device and initializes it. In this process, the guest driver communicates with the device to set up the necessary data structures and mappings.

```
root@v8a-arm64:~# dmesg | grep virtio
[    0.184166] virtio-pci 0000:00:03.0: enabling device
(0000 -> 0002)
[    0.189122] virtio_iommu virtio0: input address: 64 bits
[    0.189516] virtio_iommu virtio0: page mask:
```

```
0xfffffffffffff000
[    0.220135] virtio-pci 0000:01:00.0: Adding to iommu
group 0
[    0.220676] virtio-pci 0000:01:00.0: enabling device
(0000 -> 0002)
[    0.223065] virtio_blk virtio1: 4/0/0 default/read/poll
queues
[    0.224713] virtio_blk virtio1: [vda] 2220734 512-byte
logical blocks (1.14 GB/1.06 GiB)
root@v8a-arm64:~#
```

The DMESG logs show that the virtio-block device is attached to the IOMMU domain, which is a part of the IOMMU group `0`.

3. After initializing, the Virtio-IOMMU device manages DMA operations for attached devices. The guest OS performs mapping and unmapping operations with the virtio driver and passes the operations to the host for processing.

   The following example shows Virtio-IOMMU device operations:

```
root@v8a-arm64:~# cat /sys/kernel/iommu_groups/0/devices/
0000\:01\:00.0/virtio1/uevent
DRIVER=virtio_blk
MODALIAS=virtio:d00000002v00001AF4
root@v8a-arm64:~#
```

Virtio-net overview

The virtio-net device is a virtual network device that provides an interface for network operations in virtualized environments. It offers high performance and low overhead.

The following are the key features of the virtio-net device:

- Efficiency: Minimizes overhead and maximizes throughput.
- Simplicity: Easy to implement and use.
- Flexibility: Supports various network configurations and back ends.

Note: Ensure that `CONFIG_VIRTIO_NET` is enabled in the guest kernel.

To use a virtio-net device, do the following:

1. To configure the VM for including a virtio-net device, specify the device in the VM configuration file (libvirt XML), or pass an argument to QEMU.

a. To enable the virtio-net device in the VM, use the following QEMU command:

```
qemu-system-aarch64 \
-M virt -m 2G \
-kernel /mnt/overlay/guest/Image \
-drive file=/mnt/overlay/guest/rootfs.ext4,if=virtio,
format=raw \
-append "root=/dev/vda" \
-cpu host --enable-kvm -smp 4 -nographic \
-netdev tap,id=net0,ifname=tap0,script=no,
downscript=no \
-device virtio-net-pci,netdev=net0
```

Note: This command creates a Tap interface on the host. To create the Tap interface manually, run the following command:

```
ip tuntap add dev tap0 mode tap
```

b. To launch the guest VM with the following sample XML, see Libvirt.

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/
schemas/domain/qemu/1.0'>
  <name>simple_net</name>
  <memory unit='KiB'>2097152</memory>
  <vcpu placement='static'>4</vcpu>
  <resource>
    <partition>/machine</partition>
  </resource>
  <os>
    <type arch='aarch64' machine='virt-6.2'>hvm</
type>
    <kernel>/mnt/overlay/guest/Image</kernel>
    <cmdline>root=/dev/vda</cmdline>
    <boot dev='hd'/>
  </os>
  <features>
    <gic version='3'/>
  </features>
  <cpu mode='host-passthrough' check='none'/>
  <devices>
    <console type='pty'>
    </console>
    <disk type="file" device="disk">
```

```
      <driver name="qemu" type="raw"/>
        <!-- Specify rootfs image file here -->
      <source file="/mnt/overlay/guest/rootfs.ext4"/>
      <target dev="vda" bus="virtio"/>
    </disk>
    <interface type='ethernet'>
      <target dev='tap0'/>
      <model type='virtio'/>
    </interface>
  </devices>
</domain>
```

To create a tap interface for direct communication between the host VM and the guest VM, copy the XML content to `/mnt/overlay/guest/libvirt_virtio_net.xml` on the host.

2. The guest operating system detects the virtio-net device and initializes it. In this process, the guest driver communicates with the device to set up the necessary data structures and queues.

```
root@v8a-arm64:~# dmesg | grep virtio
[    0.394365] virtio-pci 0000:01:00.0: enabling device
(0000 -> 0002)
[    0.396364] virtio-pci 0000:02:00.0: enabling device
(0000 -> 0002)
[    0.404496] virtio_blk virtio1: 4/0/0 default/read/poll
queues
[    0.406048] virtio_blk virtio1: [vda] 2220734 512-byte
logical blocks (1.14 GB/1.06 GiB)
[    0.980919] virtio_net virtio0 enp1s0: renamed from eth0
root@v8a-arm64:~# lspci
00:00.0 Host bridge: Red Hat, Inc. QEMU PCIe Host bridge
00:01.0 PCI bridge: Red Hat, Inc. QEMU PCIe Root port
00:01.1 PCI bridge: Red Hat, Inc. QEMU PCIe Root port
00:01.2 PCI bridge: Red Hat, Inc. QEMU PCIe Root port
01:00.0 Ethernet controller: Red Hat, Inc. Virtio 1.0
network device (rev 01)
02:00.0 SCSI storage controller: Red Hat, Inc. Virtio 1.0
block device (rev 01)
root@v8a-arm64:~#
```

3. Operate: After you initialize and configure the virtio-net device, use it as other network interface. The guest OS performs standard network operations with the virtio driver and passes the operations to the host for processing.

To use the Tap interface, run the following network configurations on the host VM and the guest VM:

- Host configuration:

```
root@qcs9100-ride-sx:~# ip addr add 192.168.100.1/24 dev tap0
root@qcs9100-ride-sx:~# ip link set dev tap0 up
```

- Guest configuration:

```
root@v8a-arm64:~# ip addr add 192.168.100.2/24 dev enp1s0
root@v8a-arm64:~# ip link set dev enp1s0 up
```

To run the network traffic on this interface, use the following example:

```
root@v8a-arm64:~# ping 192.168.100.1
PING 192.168.100.1 (192.168.100.1): 56 data bytes
64 bytes from 192.168.100.1: seq=0 ttl=64 time=0.364 ms
64 bytes from 192.168.100.1: seq=1 ttl=64 time=0.156 ms
```

Virtio-serial and virtio-console overview

The virtio-serial device provides an interface for serial communication between the host VM and the guest VM in virtual environments.

The following are the key features of the virtio-serial device:

- Multiple ports: Supports multiple serial ports, allows various communication channels
- Efficiency: Minimizes overhead and maximizes throughput
- Flexibility: Supports different types of data exchange. For example, console access and file transfer

---

Note: Ensure that `CONFIG_VIRTIO_CONSOLE` is enabled in the guest kernel.

To use a virtio-serial device, do the following:

1. To configure the VM for including a virtio-serial device, specify the device in the VM configuration file (libvirt XML), or pass an argument to QEMU.
   a. To enable the virtio-net device in the VM, use the following QEMU command:

```
qemu-system-aarch64 \
-M virt -m 2G \
-kernel /mnt/overlay/guest/Image \
-drive file=/mnt/overlay/guest/rootfs.ext4,if=virtio,
```

```
format=raw \
-append "root=/dev/vda" \
-cpu host --enable-kvm -smp 4 -nographic \
-device virtio-serial-pci,id=virtio-serial0 \
-chardev pty,id=charconsole0 \
-device virtconsole,chardev=charconsole0,id=console0 \
-chardev socket,path=/tmp/qemu.sock,server=on,wait=off,
id=charchannel0 \
-device virtserialport,chardev=charchannel0,name=org.
qemu.guest_agent.0
```

b. To launch the guest VM with the following sample XML, see Libvirt.

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/
domain/qemu/1.0'>
  <name>simple_serial</name>
  <memory unit='KiB'>2097152</memory>
  <vcpu placement='static'>4</vcpu>
  <resource>
    <partition>/machine</partition>
  </resource>
  <os>
    <type arch='aarch64' machine='virt-6.2'>hvm</type>
    <kernel>/mnt/overlay/guest/Image</kernel>
    <cmdline>root=/dev/vda</cmdline>
    <boot dev='hd'/>
  </os>
  <features>
    <gic version='3'/>
  </features>
  <cpu mode='host-passthrough' check='none'/>
  <devices>
    <console type='pty'>
    </console>
    <disk type="file" device="disk">
      <driver name="qemu" type="raw"/>
        <!-- Specify rootfs image file here -->
      <source file="/mnt/overlay/guest/rootfs.ext4"/>
      <target dev="vda" bus="virtio"/>
    </disk>
    <controller type='virtio-serial' index='0'>
      <alias name='virtio-serial0'/>
    </controller>
    <console type='pty'>
```

```
      <target type='virtio' port='0'/>
    </console>
    <channel type='unix'>
      <target type='virtio' name='org.qemu.guest_agent.0'/>
      <address type='virtio-serial' controller='0' bus='0' port=
'1'/>
    </channel>
  </devices>
</domain>
```

Note: Copy the XML content to `/mnt/overlay/guest/virtio_serial.xml` on the host.

2. The guest operating system detects the virtio-serial device and initializes it. In this process, the guest driver communicates with the device to set up the necessary data structures and ports.

```
root@v8a-arm64:~# lspci
00:00.0 Host bridge: Red Hat, Inc. QEMU PCIe Host bridge
00:01.0 PCI bridge: Red Hat, Inc. QEMU PCIe Root port
00:01.1 PCI bridge: Red Hat, Inc. QEMU PCIe Root port
00:01.2 PCI bridge: Red Hat, Inc. QEMU PCIe Root port
01:00.0 Communication controller: Red Hat, Inc. Virtio 1.0
console (rev 01)
02:00.0 SCSI storage controller: Red Hat, Inc. Virtio 1.0
block device (rev 01)
root@v8a-arm64:~#
root@v8a-arm64:~# cat /sys/bus/virtio/devices/virtio0/uevent
DRIVER=virtio_console
MODALIAS=virtio:d00000003v00001AF4
root@v8a-arm64:~#
```

3. After initialization, you can use the virtio-serial device as a serial device. The guest OS performs read and write operations with the virtio driver and passes the operations to the host for processing.
4. To manage the virtio-serial device, do the following:
    - Monitor the performance
    - Verify the device status
    - Perform maintenance tasks
    - Use the standard tools and commands available in the guest OS

Memory balloon device support

The traditional virtio memory balloon device manages the guest memory. It allows the host system to reclaim memory from VMs. The memory balloon instructs VMs to return a portion of their memory to the host. This process involves inflating the memory balloon inside the Guest VM, which reduces the memory available for other tasks within the VM. The Guest OS decides which memory pages to give back to the host, indicating which pages it doesn't need or access. The host then un-maps these pages from the guest VM and marks them as unavailable for the guest VM and allows the host system to use them. If the guest VM requires more memory later, the host deflates the balloon to return the pages. This feature allows each guest VM to continue running while its available memory is managed.

Use the following for the virtio-balloon device to relocate physical memory between a guest VM and the host:

1. Balloon inflation: The guest driver allocates memory and informs the host. The host then reuses the inflated memory for other VMs.
2. Balloon deflation: After informing the host, the guest driver frees the previously allocated memory and enables the guest VM to use the deflated memory.

`Target balloon size` controls the balloon inflation or deflation through a request to change the guest VM memory size. The resize request is sent through QEMU monitor mode or through virsh commands.

The following are the key features of the virtio-balloon device:

- Memory management: Adjusts memory allocation in real time based on the requirements.
- Efficiency: Reclaims unused memory from idle VMs and redistributes it to the active ones.
- Flexibility: Enables various virtualization setups to optimize resource usage.

Note: Ensure that `CONFIG_VIRTIO_BALLOON` is enabled in the guest kernel.

To use a virtio-balloon device, do the following:

1. Configure the VM for including a virtio-balloon device, specify the device in the VM configuration file (libvirt XML), or pass an argument to QEMU.
    a. To enable the virtio-balloon device in the VM, use the following QEMU command:

```
qemu-system-aarch64 \
-M virt -m 2G \
-kernel /mnt/overlay/guest/Image \
-drive file=/mnt/overlay/guest/rootfs.ext4,if=virtio,
format=raw \
-append "root=/dev/vda" \
```

```
-cpu host --enable-kvm -smp 4 -nographic \
-device virtio-balloon-pci,id=balloon0
```

b. To launch the guest VM with the following sample XML, see Libvirt:

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/
domain/qemu/1.0'>
  <name>simple_balloon</name>
  <memory unit='KiB'>2097152</memory>
  <vcpu placement='static'>4</vcpu>
  <resource>
    <partition>/machine</partition>
  </resource>
  <os>
    <type arch='aarch64' machine='virt-6.2'>hvm</type>
    <kernel>/mnt/overlay/guest/Image</kernel>
    <cmdline>root=/dev/vda</cmdline>
    <boot dev='hd'/>
  </os>
  <features>
    <gic version='3'/>
  </features>
  <cpu mode='host-passthrough' check='none'/>
  <devices>
    <console type='pty'>
    </console>
    <disk type="file" device="disk">
      <driver name="qemu" type="raw"/>
        <!-- Specify rootfs image file here -->
      <source file="/mnt/overlay/guest/rootfs.ext4"/>
      <target dev="vda" bus="virtio"/>
    </disk>
    <memballoon model='virtio'/>
  </devices>
</domain>
```

Note: Copy the XML content to `/mnt/overlay/guest/libvirt_virtio_balloon.xml` file on the host.

2. The guest OS detects the virtio-balloon device and initializes it. In this process, the guest driver communicates with the device to set up the necessary data structures and memory management mechanisms.

```
root@v8a-arm64:~# lspci
00:00.0 Host bridge: Red Hat, Inc. QEMU PCIe Host bridge
00:01.0 PCI bridge: Red Hat, Inc. QEMU PCIe Root port
00:01.1 PCI bridge: Red Hat, Inc. QEMU PCIe Root port
00:01.2 PCI bridge: Red Hat, Inc. QEMU PCIe Root port
01:00.0 SCSI storage controller: Red Hat, Inc. Virtio 1.0
block device (rev 01)
02:00.0 Unclassified device [00ff]: Red Hat, Inc. Virtio
1.0 memory balloon (rev 01)
root@v8a-arm64:~#
```

3. After initializing, the virtio-balloon device adjusts the memory allocated to the VM. The guest OS performs memory inflation and deflation operations, which are handled by the virtio driver, and passes it to the host for processing.

To adjust the guest memory, run the following virsh commands on the host:

- To verify the guest VM initial memory (2 GB), run the following command:

```
root@v8a-arm64:~# cat /proc/meminfo  | grep Mem
MemTotal:        1966180 kB
MemFree:         1807432 kB
MemAvailable:    1817552 kB
root@v8a-arm64:~#
```

- To inflate the balloon and reduce the guest VM memory to 512 MB, run the following command:

```
root@qcs9100-ride-sx:~# virsh setmem simple_balloon 512M --live
```

- To verify the guest VM memory after inflation, run the following command:

```
root@v8a-arm64:~# cat /proc/meminfo  | grep Mem
MemTotal:         393316 kB
MemFree:          235708 kB
MemAvailable:     245836 kB
root@v8a-arm64:~#
```

- To deflate the balloon (Guest VM memory back to 2 GB):

```
root@qcs9100-ride-sx:~# virsh setmem simple_balloon 2G --live
```

- To verify the guest VM memory after deflation:

```
root@v8a-arm64:~# cat /proc/meminfo  | grep Mem
MemTotal:        1966180 kB
```

```
MemFree:         1811844 kB
MemAvailable:    1821988 kB
root@v8a-arm64:~#
```

Access host devices

You can use the enumerated devices on the host system from the guest. The device assignment interface to the guest varies based on the device type. Some devices act as a back end for a device emulated by QEMU, for example, UART. Configure universal asynchronous receiver transmitter (UART) as a character device (chardev) back end on the host. For more information about QEMU chardev, see Character device options.

PCI and USB devices work on the kernel VFIO framework.

- PCI passthrough allows a guest VM to directly access a physical PCI device on the host. The VFIO framework in the Linux kernel is used for the device assignment. For more information about the VFIO, see VFIO - Virtual Function I/O.
- A USB device operates in Passthrough mode, allowing the guest VM to use USB peripherals connected to the host. QEMU emulates the USB controller and works with libusb on the host to present USB peripherals to the guest. For more information, see USB emulation.

USB passthrough overview

Identify the USB device that's enumerated on the host computer to assign it to the guest. To see all the USB devices that are enumerated on the host, use the `lusb` command. Find the device by its `vendor id` and `product id`.

In the following example, `vendor id` is 0x0781 and `product id` is 0x5567 for the USB memory stick.

```
sh-5.1# lsusb
Bus 002 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 001 Device 004: ID 0781:5567 SanDisk Corp. Cruzer Blade
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 003 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

Note: Copy the following XML file on the host file system in `/mnt/overlay/guest/libvirt_usb.xml` file and modify the XML file per `vendor id` and `product id`.

Launch a guest VM with the following sample XML and do the following:

1. After the guest VM is launched, connect to the console.
2. Verify the `lsusb` output in the guest VM.

3. Ensure that the output lists the USB device assigned to the guest.

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/
qemu/1.0'>
  <name>simple_usb</name>
  <memory unit='KiB'>2097152</memory>
  <vcpu placement='static'>4</vcpu>
  <resource>
    <partition>/machine</partition>
  </resource>
  <os>
    <type arch='aarch64' machine='virt-6.2'>hvm</type>
    <kernel>/mnt/overlay/guest/Image</kernel>
    <cmdline>root=/dev/vda</cmdline>
    <boot dev='hd'/>
  </os>
  <features>
    <gic version='3'/>
  </features>
  <cpu mode='host-passthrough' check='none'/>
  <devices>
    <console type='pty'>
    </console>
    <disk type="file" device="disk">
      <driver name="qemu" type="raw"/>
        <!-- Specify rootfs image file here -->
      <source file="/mnt/overlay/guest/rootfs.ext4"/>
      <target dev="vda" bus="virtio"/>
    </disk>
    <controller type='usb' index='0' model='qemu-xhci' ports='4'/>
    <hostdev mode='subsystem' type='usb' managed='yes'>
      <source>
        <!-- Specify USB device VID/PID here -->
        <vendor id='0x0781'/>
        <product id='0x5567'/>
      </source>
    </hostdev>
  </devices>
</domain>
```

PCI overview

Identify the PCI device that's enumerated on the host computer to assign it to the guest. To see all the PCI devices that are enumerated on the host, use the `lspci` command. Find the device by its `[domain:]bus:device.function`. For more information about `lspci` command, see

lspci(8)—Linux manual page.

The following example shows supported values for domain, bus, device, and function for `Ethernet controller: Aquantia Corp. AQC107 NBase-T/IEEE 802.3bz` device:

- Domain is 0001
- Bus is 01
- Device is 00
- Function is 0 for

```
sh-5.1# lspci
0000:00:00.0 PCI bridge: Qualcomm Device 0115
0000:01:00.0 Network controller: Qualcomm QCNFA765 Wireless Network
Adapter (rev 01)
0001:00:00.0 PCI bridge: Qualcomm Device 0115
0001:01:00.0 Ethernet controller: Aquantia Corp. AQC107 NBase-T/IEEE
802.3bz
```

Note: Copy the XML content to `/mnt/overlay/guest/libvirt_pci.xml` file and change the `hostdev` section in the XML file according to your device requirements.

To launch the guest VM, see Libvirt. After you launch the guest VM, connect it to the console and verify the `lspci` output in the guest. Ensure that the output lists the PCI device assigned to the guest.

Note: The output shows different slots based on the guest PCI topology selected by QEMU and Libvirt.

When `managed` is marked as *yes* in `hostdev` in the following XML, the PCI device gets detached from the host before it passes on to the guest, and re-attached to the host after the guest exits. For more information, see USB / PCI / SCSI devices.

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/
qemu/1.0'>
  <name>simple_pci</name>
  <memory unit='KiB'>2097152</memory>
  <vcpu placement='static'>4</vcpu>
  <resource>
    <partition>/machine</partition>
  </resource>
  <os>
    <type arch='aarch64' machine='virt-6.2'>hvm</type>
```

```
    <kernel>/mnt/overlay/guest/Image</kernel>
    <cmdline>root=/dev/vda</cmdline>
    <boot dev='hd'/>
  </os>
  <features>
    <gic version='3'/>
  </features>
  <cpu mode='host-passthrough' check='none'/>
  <devices>
    <console type='pty'>
    </console>
    <disk type="file" device="disk">
      <driver name="qemu" type="raw"/>
      <source file="/mnt/overlay/guest/rootfs.ext4"/>
      <target dev="vda" bus="virtio"/>
    </disk>
    <hostdev mode='subsystem' type='pci' managed='yes'>
      <source>
        <!-- specify your PCI device details here -->
        <address domain='0x0001' bus='0x01' slot='0x00' function='0x0
'/>
      </source>
    </hostdev>
  </devices>
</domain>
```

UART passthrough overview

Identify the `/dev/ttyX` teletype (TTY) device file corresponding to the UART to use it from the guest. Ensure that no application on the host is using this interface.

Note: Copy the XML content to `/mnt/overlay/guest/libvirt_uart.xml` file on the host and modify the TTY device file as per your device requirements.

Launch a guest VM and connect to the console. The serial interface at `/dev/virtio-ports/hostserial` acts as a front end to the real UART on the host. Use the following XML file:

```
<domain type='kvm' xmlns:qemu='http://libvirt.org/schemas/domain/
qemu/1.0'>
  <name>simple_uart</name>
  <memory unit='KiB'>2097152</memory>
  <vcpu placement='static'>4</vcpu>
  <resource>
    <partition>/machine</partition>
```

```
  </resource>
  <os>
    <type arch='aarch64' machine='virt-6.2'>hvm</type>
    <kernel>/mnt/overlay/guest/Image</kernel>
    <cmdline>root=/dev/vda</cmdline>
    <boot dev='hd'/>
  </os>
  <features>
    <gic version='3'/>
  </features>
  <cpu mode='host-passthrough' check='none'/>
  <devices>
    <console type='pty'>
    </console>
    <channel type='dev'>
      <source path='/dev/ttyMSM0'/>
      <target type='virtio' name='hostserial'/>
      <address type='virtio-serial' controller='0' bus='0' port='1'/>
    </channel>
    <disk type="file" device="disk">
      <driver name="qemu" type="raw"/>
      <source file="/mnt/overlay/guest/rootfs.ext4"/>
      <target dev="vda" bus="virtio"/>
    </disk>
  </devices>
</domain>
```

KVM customizations

The following features allow you to modify the KVM hypervisor to meet specific requirements:

Remote command execution

To enable the QEMU guest agent in the guest OS user space, run the remote commands on the guest VMs from the host. For more information about `qemu-ga`, see QEMU Guest Agent.

`qemu-agent-command` is the virsh command that includes `guest-exec` and `guest-exec-status` as the subcommands to execute and verify the status or output of the guest VMs. For more information about how to configure `qemu-ga` through virtio-serial interface, see QEMU Guest Agent.

For more information about how `virt-exec` wraps virsh commands and provides an interface, see kvm-qemu/virt-exec.

The following example shows how to run `/proc/meminfo` on the guest VM from the host computer using a `virt-exec` utility:

```
sh-5.2# ./virt-exec testvm cat /proc/meminfo | grep -i total
MemTotal:         1968056 kB
SwapTotal:              0 kB
VmallocTotal:   133141626880 kB
CmaTotal:           32768 kB
HugePages_Total:        0
```

Watchdog configuration

A QEMU-emulated I6300 ESB watchdog device is supported in the guest VM. The Linux kernel has a driver for this watchdog and exposes the standard watchdog character device. To enable the watchdog, set `CONFIG_I6300ESB_WDT` in the guest kernel configuration. A guest user space daemon must pet the watchdog.

The guest Libvirt XML has an option to select `poweroff` or `reset` on watchdog timeout. For more information, see Watchdog devices.

Add the following snippet to the guest XML to enable I6300 ESB watchdog emulation in QEMU. The default action is `reset`, which restarts the guest.

```
<devices>
  <watchdog model='i6300esb'/>
</devices>
```

KVM traces

To enable KVM traces, run the following command:

```
echo 1 > /sys/kernel/tracing/events/kvm/enable
```

The KVM guest interactions are recorded in kernel function trace (ftrace). To inspect interactions between KVM and guest VMs, read the kernel trace buffer.

QEMU also supports trace events, which can be redirected to kernel ftrace. For more information about how to redirect QEMU trace back-end events to the ftrace buffer, see Trace backends. This option is selected by default in the Qualcomm Linux release.

To launch a guest VM with virtio traces enabled in the QEMU code, run the following command:

```
qemu-system-aarch64 \
    -M virt -m 2G \
    -kernel /mnt/overlay/guest/Image \
    -drive file=/mnt/overlay/guest/rootfs.ext4,if=virtio,format=raw \
    -append "root=/dev/vda" \
    -cpu host --enable-kvm -smp 4 -nographic \
    -trace "virtio*"
```

5 Real-time (RT) kernel overview

A real-time system is a deterministic system, where response to an event is expected in a set time.

A system is classified as compatible with RT if:

- It's devoid of unbounded latency.
- The maximum response time is calculated with precision.
- It meets the set criteria for scheduling of tasks (latency and deadline).

Linux can be configured as a real-time operating system (RTOS) in which real-time tasks have well-defined periodic execution cycles (cycle time) and meet execution criteria within specified limits (jitter).

When considering Linux as RTOS, apply `PREEMPT_RT` patches and enable configurations for a fully preemptible kernel.

The RT kernel maintainers maintain out-of-tree `PREEMPT_RT` patches for every kernel version released.

To install the patches, see Versions of PREEMPT_RT patches.

Note: The real-time support is for kernel space process and not for user space.

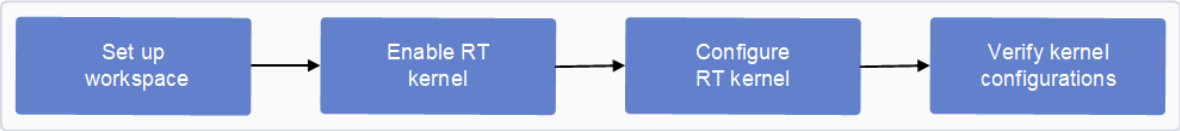

Figure : Build sequence

5.1 Set up workspace

The Qualcomm Linux kernel supports the LTS RT kernel (6.6.x), which is maintained through the Yocto recipe in the `meta-qcom-realtime` layer in the `recipes-kernel/linux/linux-kernel-qcom-rt_6.6.bb` file.

The Linux RT kernel recipe in Qualcomm Linux is referred as `linux-qcom-base-rt`.

For more information about cloning the workspace and getting all Qualcomm Linux meta layers to use Qualcomm RT Linux, see Sync and build with real-time Linux

5.2 Enable RT kernel

The Qualcomm Linux `meta-qcom-realtime` layer supports a `linux-qcom-base-rt` recipe that fetches and builds the Qualcomm Linux kernel for the supported machines by default.

The `meta-qcom-realtime` layer applies the changes on top of the existing layer. During the kernel build, `meta-qcom-realtime` layer appends the upstream `PREEMPT_RT` patches based on the kernel version, and allows real-time configurations.

Note:

- Use linux-qcom-custom-rt_6.6.bb for custom BSP.
- Use linux-qcom-base-rt_6.6.bb for base BSP.

The following is the sample code for the RT base reference file:

```
require recipes-kernel/linux/linux-qcom-base-rt_6.6.bb

SECTION = "RT kernel"
SUMMARY = "Linux Real time kernel for QCOM devices"
DESCRIPTION = "Recipe to build real time Linux kernel"

SRC_URI:append = "https://cdn.kernel.org/pub/linux/kernel/projects/
rt/6.6/older/patch-6.6.14-rt21.patch.gz;
md5sum=91969a704a73aa918c89d3027bdd3634 \
```

```
                file://qcom_rt.cfg \
                file://0001-arch-Kconfig-Add-RT-kernel-support.patch \

S = "${WORKDIR}/kernel"
KERNEL_CONFIG_FRAGMENTS:append = " ${WORKDIR}/qcom_rt.cfg"
```

For more information about supported machines, see Identify supported Qualcomm machines.

5.3 Configure RT kernel

- To configure the RT kernel, use the following procedure:

```
SRC_URI:append = "https://cdn.kernel.org/pub/linux/kernel/projects/
rt/6.6/older/patch-6.6.14-rt21.patch.gz;
md5sum=91969a704a73aa918c89d3027bdd3634 \
                file://qcom_rt.cfg \
                file://0001-arch-Kconfig-Add-RT-kernel-support.patch \
```

- To apply the external configurations on the RT kernel:
  - Maintain the configuration file in the `meta-qcom-realtime/recipes-kernel/linux/linux-qcom-base-rt/configs/qcom_rt.cfg` recipe.
  - Append the configuration file to `KERNEL_CONFIG_FRAGMENTS` in the `meta-qcom-realtime/recipes-kernel/linux/linux-qcom-base-rt_6.6.bb` file.
- To add a configuration fragment to the RT kernel, make the following change:

```
KERNEL_CONFIG_FRAGMENTS:append = " ${WORKDIR}/qcom_rt.cfg"
```

- To modify the kernel command-line, add the command-line parameter into the `meta-qcom-realtime/conf/layer.conf` file.
- The following example shows how to modify a command-line:

```
KERNEL_CMDLINE_EXTRA = "root=/dev/disk/by-partlabel/system rw
rootwait console=ttyMSM0,115200n8 pcie_pme=nomsi earlycon
idle=poll skew_tick=1 rcu_nocbs=1-3 rcu_nocb_poll nohz_full=1-3
irqaffinity=4-7 isolcpus=1-3"
```

5.4 Kernel configurations

Optional and mandatory kernel configurations are used in the RT kernel.

To enable full preemption in the RT kernel, use `CONFIG_PREEMPT_RT`.

The `CONFIG_PREEMPT_RT` flag is enabled by default as part of the `linux-qcom-base-rt_6.6.bb` recipe.

The following example shows the kernel configuration:

```
# CONFIG_PREEMPT_RT
  zcat proc/config.gz | grep CONFIG_PREEMPT
# CONFIG_PREEMPT_NONE is not set
# CONFIG_PREEMPT_VOLUNTARY is not set
# CONFIG_PREEMPT is not set
CONFIG_PREEMPT_RT=y
```

To optimize the RT kernel response, use the following configurations:

```
# CONFIG_NO_HZ
  zcat proc/config.gz | grep NO_HZ
CONFIG_NO_HZ_COMMON=y
# CONFIG_NO_HZ_IDLE is not set
CONFIG_NO_HZ_FULL=y
# CONFIG_NO_HZ is not set

#CONFIG_CPUSETS
 zcat proc/config.gz | grep CPUSETS
CONFIG_CPUSETS=y
```

Set the following kernel configuration options when affining the RT task:

- `CONFIG_NO_HZ_FULL` - When enabled, it configures the kernel to avoid sending scheduling-clock interrupts to CPUs with a single runnable task.
- `CONFIG_CPUSETS` - Use the `CONFIG_CPUSETS` configuration option to enable `cpuset`, where the CPU is grouped to form a set.

When affining the RT task to a set of CPUs, use the `CONFIG_CPUSETS` configuration option:

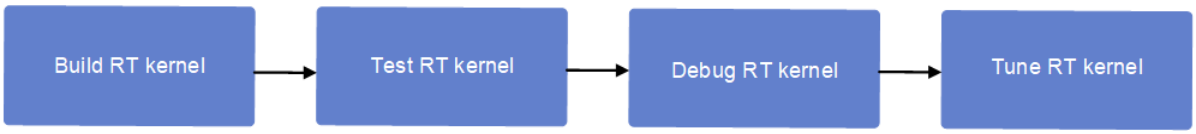

Figure: RT kernel verification

5.5 Build a RT kernel

To build the RT kernel, run the following commands:

```
MACHINE=<SoC>-<board>-<variant> DISTRO=qcom-wayland QCOM_SELECTED_
BSP=base source setup-environment

#verify realtime layer is present

bitbake-layers show-layers | grep realtime
   meta-qcom-realtime    /layers/meta-qcom-realtime  12

# compile
  bitbake qcom-console-image
```

5.6 Test RT kernel

A suite of tests is available in the Linux foundation RT test suite.

The cyclic test determines the best- and worst-case latencies for an RT application.

1. Access the RT test suite source code at rt-tests/rt-tests.git.
2. To include the RT test suites as a part of the image, change the `layer.conf` file in the standard Yocto build:

   ```
   IMAGE_INSTALL:append = "rt-tests numactl"
   ```

3. Run the following cyclic test:

   ```
   cgexec -g cpuset:core1-3 cyclictest -a 1-3 -t 3 -m -l 100000000
   -i 1000 -p 99 -h 800
   ```

For more information, see RT-Tests.

5.7 Debug RT kernel

The following methods are used for troubleshooting issues in the RT kernel:

- How to verify that the underlying kernel is real time?

  After the boot is complete, run the following commands to verify the kernel type:

  ```
  uname -r
  6.6-rt15
  uname -v
  SMP PREMPT_RT
  ```

- How to make drivers RT compliant?

  Ensure that synchronization primitives don't break RT assumptions.

  The following are examples of the scenarios that can break RT assumptions and cause unexpected system behavior:

```
/* Acquiring a preemptible lock in non preemptible context */

preempt_disable( )
......
spin_lock( )

/* Acquiring a preemptible lock in non preemptible context /

raw_spin_lock( )
......
spin_lock( )

/* Acquiring a non preemptible lock in preemptible context /

local_lock_irq( )
   .....
raw_spin_lock( )
```

- Debug RT kernel configurations

  The following are the debug configurations for the RT kernel:

  – `CONFIG_DEBUG_ATOMIC_SLEEP` - Verify for sleep inside an atomic section.

  – `CONFIG_PROVE_RAW_LOCK_NESTING` - Allows the `raw_spinlock` vs. spinlock nesting. Ensures that the lock nesting rules for `PREEMPT_RT` kernels aren't violated.

    While debugging, you can switch from an RT kernel to a non-RT kernel. To switch, make changes to the `meta-qcom-realtime/conf/layer.conf` file to change virtual/kernel settings in `linux-qcom-base-rt` and compile again.

    To make the changes, run the following commands:

```
- PREFERRED_PROVIDER_virtual/kernel = "linux-qcom-base"
+ PREFERRED_PROVIDER_virtual/kernel = "linux-qcom-base-rt"
```

For more information about locking primitives with the RT kernel, see Lock types and their rules—The Linux Kernel documentation.

5.8 Tune RT kernel

Tune the RT kernel to achieve a deterministic latency for RT tasks in the device.

Set the CPU cores that run RT tasks to run at maximum operating frequency while preventing thermal mitigation of CPU frequency. For example, in an idle sleep scenario, RT tasks face scheduling latency due to CPU wake time delay.

For a system with eight CPU cores, use the following system configuration:

- RT tasks as outlined in the following example are affined to cores 1-3.
- CPU cores 1-3 run in isolated mode with all read-copy-update (RCU) affined to cores 4-7.
- IRQs are affined to cores 4-7 in the kernel command-line arguments.
- System sleep is disabled for CPU cores 1-3 and is set to run at maximum frequency.

```
# Disable low power mode and sleep for CPU 1-3
  echo 1 > /sys/devices/system/cpu/cpu1/cpuidle/state2/disable
  echo 1 > /sys/devices/system/cpu/cpu1/cpuidle/state1/disable
  echo 1 > /sys/devices/system/cpu/cpu1/cpuidle/state0/disable
  echo 1 > /sys/devices/system/cpu/cpu2/cpuidle/state0/disable
  echo 1 > /sys/devices/system/cpu/cpu2/cpuidle/state1/disable
  echo 1 > /sys/devices/system/cpu/cpu2/cpuidle/state2/disable
  echo 1 > /sys/devices/system/cpu/cpu3/cpuidle/state2/disable
  echo 1 > /sys/devices/system/cpu/cpu3/cpuidle/state1/disable
  echo 1 > /sys/devices/system/cpu/cpu3/cpuidle/state0/disable

# Enable max freq for cpu group 0-3
  echo 1958400 > /sys/devices/system/cpu/cpufreq/policy0/
scaling_min_freq

  echo performance > /sys/devices/system/cpu/cpufreq/policy0/
scaling_governor

# form cgroup for core 0-3
  mkdir /sys/fs/cgroup/cpuset/core0/
  mkdir /sys/fs/cgroup/cpuset/core1/
  mkdir /sys/fs/cgroup/cpuset/core2/
  mkdir /sys/fs/cgroup/cpuset/core3/

  echo 0 > /sys/fs/cgroup/cpuset/core0/cpuset.mems
  echo 0 > /sys/fs/cgroup/cpuset/core1/cpuset.mems
  echo 0 > /sys/fs/cgroup/cpuset/core2/cpuset.mems
  echo 0 > /sys/fs/cgroup/cpuset/core3/cpuset.mems

  echo 0 > /sys/fs/cgroup/cpuset/core0/cpuset.cpus
```

```
echo 1 > /sys/fs/cgroup/cpuset/core1/cpuset.cpus
echo 2 > /sys/fs/cgroup/cpuset/core2/cpuset.cpus
echo 3 > /sys/fs/cgroup/cpuset/core3/cpuset.cpus

mkdir /sys/fs/cgroup/cpuset/core1-3/
echo 0 > /sys/fs/cgroup/cpuset/core1-3/cpuset.mems
echo 1-3 > /sys/fs/cgroup/cpuset/core1-3/cpuset.cpus
```

The following example shows how to add a kernel command-line parameter to disable RCU callbacks (**rcu_nocbs**) in `meta-qcom-realtime/conf/layer.conf`:

- CPU cores 1-3
- IRQ affine to core 4-7
- Isolate CPU 1-3

```
KERNEL_CMDLINE_EXTRA = "root=/dev/disk/by-partlabel/system rw
rootwait console=ttyMSM0,115200n8 pcie_pme=nomsi earlycon skew_tick=1
**rcu\_nocbs**=1-3 rcu_nocb_poll nohz_full=1-3 irqaffinity=4-7
isolcpus=1-3"
```

6 Yocto support

The Qualcomm Linux `meta-qcom-hwe` layer supports `linux-qcom-base_6.6.bb` and `linux-qcom-custom_6.6.bb` recipes that fetch and build the Qualcomm Linux kernel for supported machines.

For more information about available recipes, see Build the Yocto image recipes and kernel configurations.

6.1 Retrieve the kernel source

Fetch the kernel source from the CodeLinaro repository as a part of BitBake builds.

Yocto kernel

```
MACHINE=<SoC>-<board>-<variant> DISTRO=qcom-wayland source setup-
environment
bitbake qcom-console-image

# kernel source is downloaded under following location
  ls build-qcom-wayland/tmp-glibc/work-shared/<SoC>-<board>-<variant>
/kernel-source/
  arch     Documentation  ipc       MAINTAINERS  samples   virt
  block    drivers        Kbuild    Makefile     scripts
  certs    fs             Kconfig   mm           security
  COPYING  include        kernel    net          sound
  CREDITS  init           lib       README       tools
  crypto   io_uring       LICENSES  rust         usr
```

Note: Machine name for the previous commands is the same as the SoC-board-variant name. For example, for the QCS6490 SoC, `QCS6490` is the machine name, `rb3gen2` is the board, and `Visionkit` is the variant.

For more information about the setup and obtaining a source, see the Qualcomm Linux Build Guide.

6.2 Retrieve the kernel recipe

Qualcomm Linux maintains the `meta-qcom-hwe` layer and hosts kernel recipe files at the following locations:

- Base BSP: `meta-qcom-hwe/recipes-kernel/linux//linux-qcom-base_6.6.bb` file.
- Custom BSP: `meta-qcom-hwe/recipes-kernel/linux//linux-qcom-custom_6.6.bb` file.

`PREFERRED_PROVIDER_virtual/kernel` is defined as `linux-qcom-base` or `linux-qcom-custom` in the `meta-qcom-hwe/conf/machine/include/qcom-base.inc` file.

For the kernel recipe, run the following commands:

```
# kernel recipe location
   ls -l meta-qcom-hwe/recipes-kernel/linux/
   linux-kernel-headers-install_6.6.bb
   linux-kernel-qcom-headers_6.6.bb
   linux-qcom-base-6.6
   linux-qcom-base_6.6.bb
   linux-qcom-custom
   linux-qcom-custom_6.6.bb
```

6.3 Unpack the kernel source

Unpack the kernel source for development and customization after cloning the meta layer.

To unpack the source code, run the following commands:

```
# unpack kernel
# checkout kernel source aligned to upstream LTS base with patches
applied from recipe
# Following would checkout kernel source in build-qcom-wayland/
workspace/sources/linux-qcom-base/
$  devtool modify linux-qcom-base
```

Note: For the custom BSP variant, use `linux-qcom-custom`.

6.4 Make kernel changes

Kernel changes or customizations are done in the `build-qcom-wayland/workspace/sources/linux-qcom-base` workspace.

To modify the kernel configuration, run the following commands:

```
# Modify sources in build-qcom-wayland/workspace/sources/linux-qcom-
base, or update kernel configuration by running makemenufonfig
  devtool menuconfig linux-qcom-base

# config fragment is updated in following location
  ls build-qcom-wayland/workspace/sources/linux-qcom-base/oe-local-
files/devtool-fragment.cfg
devtool-fragment.cfg

# you may do other changes to kernel and commit
```

Note: For the custom BSP variant, use `linux-qcom-custom`.

6.5 Commit kernel changes

To commit your changes, run the following Git commands:

```
cd build-qcom-wayland/workspace/sources/linux-qcom-base
git add .
git commit -s -m "my changes"
```

6.6 Build the kernel image

After the changes are done, to build the kernel and the image, run the following commands:

```
# to build kernel use
  devtool build linux-qcom-base
```

```
# and following to build the image
  devtool build-image qcom-console-image
```

```
# built images are produced in standard location
  ls build-qcom-wayland/tmp-glibc/deploy/images/<SoC>-<board>-
<variant>/
```

Note: To compile the custom BSP, use `linux-qcom-custom`.

6.7 Maintain kernel changes

Use Devtool to develop and export the patch and create an append file in the `meta-mylayer` layer. To create your own layer, first add it to the host kernel `bbappend` files and make changes.

To create and append kernel patches in the meta layer, run the following commands:

```
# create your own layer first and add it to host kernel bbappend and
changes
  bitbake-layers create-layer ~/meta-mylayer
# mkdir -p ~/meta-mylayer/recipes-kernel/linux/linux-qcom-base
  bitbake-layers add-layer ~/meta-mylayer

# following would update the meta-mylayer recipe and the change
  devtool finish linux-qcom-base ~/meta-mylayer

# devtool finish shall populate the meta-mylayer recipes along with
patches,

  ls -R meta-mylayer/recipes-kernel/
meta-mylayer/recipes-kernel/:
linux

meta-mylayer/recipes-kernel/linux:
linux-qcom-base  linux-qcom-base_%.bbappend

meta-mylayer/recipes-kernel/linux/linux-qcom-base:
0001-my-patch.patch  devtool-fragment.cfg

  less meta-mylayer/recipes-kernel/linux/linux-qcom-base%.bbappend
FILESEXTRAPATHS:prepend := "${THISDIR}/${PN}:"

SRC_URI += "file://devtool-fragment.cfg \
            file://0001-my-patch.patch"
```

After running the `devtool finish` command, the `~/meta-mylayer` layer is updated with the following:

- Corresponding kernel changes hosted as patches in

`meta-mylayer/recipes-kernel/linux/linux-qcom-base/*.patch`.

- Add and update `meta-mylayer/recipes-kernel/linux/linux-qcom-base_%.bbappend SRC_URI` with these patches.
- Remove `build-qcom-wayland/workspace` after the development is completed.

Note: For the custom BSP variant, use `linux-qcom-custom`.

6.8 Clean up the workspace

To clean up the workspace, use the following Devtool and BitBake commands:

```
# to remove layer
  bitbake-layers remove-layer ~/meta-mylayer

# to clean complete workspace
  bitbake -c cleanall <recipe-name>

# reset using Devtool
  devtool reset linux-qcom-base

# manually delete the workspace directory
  rm -rf build-qcom-wayland/workspace/sources/linux-qcom-base
```

7 Customize the kernel

The SoC and board device tree support are hosted in the kernel source at `arch/arm64/boot/dts/qcom`.

7.1 Develop the kernel

To customize the kernel and generate patches during kernel development, use the following kernel recipe:

```
# setup the workspace and get the sources following build guide
# setup the environment

  MACHINE=<SoC>-<board>-<variant> DISTRO=qcom-wayland source setup-
environment

# create your own layer to host changes
  bitbake-layers create-layer ~/meta-mylayer
  bitbake-layers add-layer ~/meta-mylayer

# use devtool to setup kernel source for development
  devtool modify linux-qcom-base

# do the development , commit changes, build and test
  devtool build linux-qcom-base
  devtool build-image qcom-console-image

# built images are produced in standard location
  ls build-qcom-wayland/tmp-glibc/deploy/images/<SoC>-<board>-
<variant>/

# generate the patches and update layer
  devtool finish linux-qcom-base ~/meta-mylayer
```

For more information about customizing the Qualcomm Linux kernel recipe, see Kernel recipe.

Note: For the custom BSP variant, use `linux-qcom-custom`.

For more information about Yocto provisions, host patches, and how to apply them, see Yocto Project Linux Kernel Development.

7.2 Kernel standalone development

Linux kernel can also be compiled without Yocto build system support.

Prerequisites

Provision the following dependencies to setup the kernel compilation process:

- aarch64 toolchain
- systemd-boot EFI stub to include it as a part of the UKI image
- systemd ukify tool to package the kernel image, initramfs, and DTB into the UKI image
- initramfs with UKI image
- ESP image that carries the updated UKI and must be flashed to boot the device

Download the aarch64 toolchain from ARM developer site.

Installing all the dependencies enables you to do incremental kernel builds without rebuilding the whole Yocto.

To build the Qualcomm® Linux Yocto base see, Getting started with Qualcomm Linux kernel. After Yocto base is built for Qualcomm Linux, find ukify, EFI boot stub, initramfs, and efi.bin at the following directories:

- ukify at `<build-path>/tmp-glibc/sysroots-components/x86_64/systemd-boot-native/usr/bin/ukify`
- EFI boot stub at `<build-path>/tmp-glibc/deploy/images/<SoC>/linuxaa64.efi.stub`
- initramfs at `<build-path>/tmp-glibc/deploy/images/<SoC>/initramfs-qcom-image-<SoC>.cpio.gz`
- `efi.bin` is generated out of build and is present at `<build-path>/tmp-glibc/deploy/images/<SoC>/qcom-<BUILD_TYPE>/efi.bin`

You can use the custom initramfs. Access the initramfs for arm64 hosted at Linaro Snapshots site, using the following command:

```
wget https://snapshots.linaro.org/member-builds/qcomlt/testimages/
arm64/latest/initramfs-test-image-qemuarm64-*.rootfs.cpio.gz -O
<download-path>/linaro-initramfs.cpio.gz
```

Get the Linux kernel source

Clone the Linux kernel source from the git repository hosted at CodeLinaro.

To clone the repository and get the source code, run the following commands:

```
  git clone https://git.codelinaro.org/clo/la/kernel/qcom kernel-src
  cd kernel-src/
  git checkout <released sha>
# e.g.
  git checkout origin/kernel.qclinux.1.0.r2-rel
```

Configure and build Linux kernel

Install the toolchain and set the following path:

```
export PATH=<PATH_TO_ARM_TOOLCHAIN>/arm-gnu-toolchain-13.3.rel1-x86_
64-aarch64-none-linux-gnu/bin:$PATH
```

Verify if the following command is functioning correctly:

```
aarch64-linux-gnu-gcc --version
```

After all the changes are completed, to build and configure the Linux kernel, run the following commands:

```
export CROSS_COMPILE=aarch64-linux-gnu-
export ARCH=arm64
make qcom_defconfig
make -j8 dir-pkg INSTALL_MOD_STRIP=1
```

For more information about building the kernel and the image, see build kernel image.

To build the kernel and generate the images, use the `make -j8 dir-pkg INSTALL_MOD_STRIP=1` command. The images get added to the `kernel-src/tar-install` directory. The compiled files are present in the following directories:

- Kernel image in the `kernel-src/arch/arm64/boot/Image` directory
- DTBs are deployed in the `kernel-src/tar-install/boot/dtbs/` directory
- Kernel modules in the `kernel-src/tar-install/lib/modules/` directory

Update initramfs with compiled kernel modules

Copy the initramfs file to the `kernel-src` directory and overlay the compiled modules on your initramfs.

Note:

- You can use your own custom or publicly available initramfs to build the kernel and boot into a shell.
- If you need a full-featured rootfs, you must rely on the Yocto build.

To use your ramdisk, run the following command:

```
cp <download-path>/linaro-initramfs.cpio.gz ./initramfs-qcom-image.
cpio.gz && (cd tar-install ; find lib/modules | cpio -o -H newc -R
+0:+0 | gzip -9 >> ../initramfs-qcom-image.cpio.gz)
```

Package UKI image

After the Linux kernel image, DTB, and initramfs with all the kernel modules are ready, package the UKI image.

Note:

The ukify tool supports Python (3.10) or later. To run the ukify tool, install `pip install pefile` to support ukify tool execution.

To build the ukify utility, run the following commands:

```
cd build-qcom-wayland/
```

```
STUB = $PWD/tmp-glibc/deploy/images/<SoC>/linuxaa64.efi.stub
```

```
KERNEL_IMAGE = <STANDALONE_kernel-src>/arch/arm64/boot/Image
```

```
INITRD = <STANDALONE_kernel-src>/initramfs-qcom-image.cpio.gz
```

```
DTB = <STANDALONE_kernel-src>/arch/arm64/boot/dts/qcom/qcs6490-
rb3gen2.dtb
```

```
KERNEL_VENDOR_CMDLINE = "root=/dev/disk/by-partlabel/system rw
rootwait console=ttyMSM0,115200n8 earlycon qcom_geni_serial.con_
enabled=1 kernel.sched_pelt_multiplier=4 mem_sleep_default=s2idle"
```

```
UKINAME = "uki.efi"
```

```
UKI_OUT=  <OUT_DIR>/$UKINAME
```

```
rm -f "${UKI_OUT}"
```

```
./tmp-glibc/sysroots-components/x86_64/systemd-boot-native/usr/bin/
ukify build --efi-arch=aa64 --stub="${STUB}" --linux="${KERNEL_IMAGE}
" --initrd="${INITRD}" --devicetree="${DTB}"  --cmdline="${KERNEL_
VENDOR_CMDLINE}" --output="${UKI_OUT}"
```

The ukify build command generates the `uki.efi` image with the built kernel and other images. Ignore the following warnings from ukify:

```
Kernel version not specified, starting autodetection
Real-Mode Kernel Header magic not found
+ readelf --notes $KERNEL_IMAGE
readelf: Error: Not an ELF file - it has the wrong magic bytes at the
start
Found uname version: 6.6.38-perf-gb09a1d09b89b-dirty
Wrote unsigned $UKI_OUT
```

Package ESP image

When the UKI image is ready, update the ESP image. Then, the updated image is flashed onto the board and booted. Use the following instructions to update `efi.bin` from the Yocto build with `uki.efi` from the standalone kernel.

Note: Overwrite the UKI image in EFI with the default image name in `efi.bin`.

To overwrite the UKI image, run the following commands:

```
sudo mount -t vfat efi.bin /mnt/
cp uki.efi /mnt/EFI/Linux/linux-<SoC>.efi
umount /mnt
```

To flash the EFI image and reboot, see Bring up the device.

Note: To build additional out-of-tree kernel modules, rely on the full Yocto build mechanism.

7.3 Configure the kernel

The Yocto build system is used for modifying the kernel configuration, while invoking `menuconfig`.

To modify the kernel configuration, run the following commands:

```
 MACHINE=<SoC>-<board>-<variant> DISTRO=qcom-wayland source setup-
environment
 bitbake linux-qcom-base -c menuconfig

# Above would update .config in kernel build directory build-qcom-
wayland/tmp-glibc/work/<SoC>-<board>-<variant>/linux-qcom-base/6.6-
r0/build/
# one can create a config fragment for modifications made by issuing
following

  bitbake linux-qcom-base -c diffconfig

# Above would create fragment.cfg in build directory build-qcom-
wayland/tmp-glibc/work/<SoC>-<board>-<variant>/linux-qcom-base/6.6-
r0/
```

Alternatively, use the Devtool to modify the kernel configuration:

```
 devtool modify linux-qcom-base
 devtool menuconfig linux-qcom-base
 devtool finish linux-qcom-base ~/meta-mylayer

# this would create a config fragment as a patch and update in your
meta layer
```

Note: For the custom BSP variant, use `linux-qcom-custom`.

For more information about Yocto-related details on kernel configuration, see Configuring the Kernel.

7.4 Create a debug build

To create a debug build, pass `DEBUG_BUILD=1` as an argument in the shell:

```
# setup the build environment
  export SHELL=/bin/bash
```

```
  MACHINE=<SoC>-<board>-<variant> DISTRO=qcom-wayland QCOM_SELECTED_
BSP=base source setup-environment

# build qcom linux console image
  DEBUG_BUILD=1 bitbake qcom-console-image
```

Note: For the custom BSP variant, use `QCOM_SELECTED_BSP=custom`.

7.5 Update the kernel command-line

To update the kernel command-line, modify the Yocto configuration variable `KERNEL_CMDLINE_EXTRA` in the corresponding SoC-specific machine inclusion file. For example, `meta-qcom-hwe/conf/machine/include/qcom-<SoC>.inc`.

To update the kernel command-line, modify the following variable:

```
KERNEL_CMDLINE_EXTRA = "root=/dev/disk/by-partlabel/system rw
rootwait console=ttyMSM0,115200n8 pcie_pme=nomsi earlycon"
```

7.6 Platform device tree and kernel configuration

To update the DTB support in the kernel and select a DTB on boot, use the following procedure:

DTB build support in kernel

To integrate the device tree of a new platform into the kernel build, update the `Makefile`.

The following example shows customizing DTB for the QCS6490 SoC. Replicate the following method to add new DTBs.

```
diff --git a/arch/arm64/boot/dts/qcom/Makefile b/arch/arm64/boot/dts/
qcom/Makefile
index 183aeba47193..a7815c774f7c 100644
--- a/arch/arm64/boot/dts/qcom/Makefile
+++ b/arch/arm64/boot/dts/qcom/Makefile
dtb-$(CONFIG_ARCH_QCOM)         += qcs6490-addons-rb3gen2.dtb
+dtb-$(CONFIG_ARCH_QCOM)          += qcs6490-my-board.dtb
dtb-$(CONFIG_ARCH_QCOM)         += qcs6490-rb3gen2.dtb
dtb-$(CONFIG_ARCH_QCOM)         += qcs404-evb-1000.dtb
```

DTB inclusion in machine configuration

The Yocto machine configuration is also updated to include the corresponding device tree blob. For example, to add the device tree for QCS6490 machine support, use the following file:

`meta-qcom-hwe/conf/machine/qcs6490-rb3gen2-core-kit.conf`:

```
OUT_OF_KERNEL_DTSO = qcs6490-rb3gen2-core-kit.conf
# List of dtbs for corresponding supported qcs6490 platforms
KERNEL_DEVICETREE = " \
                     qcom/qcs6490-my-board.dtb   \
                     qcom/qcs6490-addons-rb3gen2.dtb \
                     qcom/qcs6490-my-board.dtb   \
                     "


# Additional list of DTBOs to be overlaid on top of base kernel
devicetree
# See how existing boards are managing it in the following example:
# Format - KERNEL_TECH_DTBOS[<base-dtb-name>] = "<dtbo1 <dtbo2> ..."

KERNEL_TECH_DTBOS[qcs6490-addons-rb3gen2] = " \
qcm6490-graphics.dtbo qcm6490-wlan-rb3.dtbo \
qcm6490-display-rb3.dtbo qcm6490-bt.dtbo \
qcm6490-video.dtbo qcm6490-wlan-upstream.dtbo \
```

Note: See machine configuration files in the `meta-qcom-hwe/conf/machine/*.conf` directory for different SoCs. The DTB filename for the custom BSP variant contains `addons`.

DTB selection on boot

The custom DTB is packaged as a part of the UKI image that's updated in EFI to boot with the selected DTB.

Generate a UKI image using the ukify tool. The ukify tool is available as part of the Yocto build in the `tmp-glibc/sysroots-components/x86_64/systemd-boot-native/usr/bin/ukify` build directory.

To generate the UKI image, run the following:

```
# Note - ukify tool need python 3.10 version or above

   ukify build --efi-arch=aa64  \
            --stub=<build-path>/tmp-glibc/deploy/images/<SoC>/
linuxaa64.efi.stub \
            --linux=<build-path>/tmp-glibc/deploy/images/<SoC>/Image
\
```

```
            --initrd=<build-path>/tmp-glibc/deploy/images/<SoC>/
initramfs-qcom-image-<SoC>.cpio.gz \
            --cmdline="console=ttyMSM0,115200n8 earlycon qcom_geni_
serial.con_enabled=1 kernel.sched_pelt_multiplier=4 mem_sleep_
default=s2idle" \
            --devicetree=<build-path>/tmp-glibc/deploy/images/<SoC>/
<SoC>-my-board.dtb \
            --output=./uki.efi
```

The ukify build command generates the `uki.efi` image with a custom board DTB.

To update the `uki.efi` image in the ESP partition, do the following:

```
# Following may need sudo privilege

# Take the yocto build generated efi.bin and mount it locally
  mount <build-path>/tmp-glibc/deploy/images/<SoC>/efi.bin  /mnt --
options rw

# Overwrite the uki.efi with one packaged above
  cp uki.efi /mnt/EFI/Linux/uki.efi
  umount /mnt

# now efi.bin carries packaged uki.efi which can be flashed to the
target and booted
# UEFI shall now pick the <SoC>-my-board.dtb that is part of uki.efi
image

# reboot into fastboot and flash efi.bin
  fastboot flash efi <build-path>/tmp-glibc/deploy/images/<SoC>/efi.
bin
```

For more information about the device tree specification, see The Devicetree Specification.

For the Linux kernel documentation for device tree, see Linux and the Devicetree.

7.7 Update the ESP images

To compile the systemd-boot boot manager and kernel images into a packaged UKI type-2 image file, use the following:

Yocto build for hardware SoC generates all the required images and package boot images as `efi.bin` that's flashed into the EFI partition. The `efi.bin` file consists of systemd-boot boot manager and kernel images, which are packaged as UKI type-2 image format.

Rebuild the EFI image after updating the kernel source, configuration, or DTS and flash the generated `efi.bin` to the EFI partition.

```
MACHINE=<SoC>-<board>-<variant> DISTRO=qcom-wayland source setup-
environment

# build qcom linux console image
  DEBUG_BUILD=1 bitbake qcom-console-image

# build images are produced in following directory
  ls build-qcom-wayland/tmp-glibc/deploy/images/<SoC>-<qcom>-
<variant>/efi.bin
efi.bin

# reboot into fastboot
  fastboot flash efi efi.bin
  fastboot flash dtb_a dtb.bin
```

For more information about the UKI type-2 image format, see Type #2 EFI Unified Kernel Images.

For more information about ESP, see Boot and Bring up the device.

7.8 Customize the “initramfs” package

To update the initramfs package, modify the `PACKAGE_INSTALL` list in `meta-qcom-hwe/recipes-kernel/images/initramfs-qcom-image.bbappend` file:

```
  less meta-qcom-hwe/recipes-kernel/images/initramfs-qcom-image.
bbappend

# Add additional packages needed as part of initrd
PACKAGE_INSTALL += " \
    e2fsprogs \
    e2fsprogs-e2fsck \
    e2fsprogs-mke2fs \
    e2fsprogs-resize2fs \
    e2fsprogs-tune2fs \
    ${VIRTUAL-RUNTIME_dev_manager} \
    os-release-initrd \
    "
```

7.9 Add a kernel module

To compile out-of-tree kernel modules using the Yocto build system, use the following procedure:

1. Create a ``Makefile`` for out-of-tree kernel driver.

   The following is the sample `Makefile` for out-of-tree kernel driver:

```
all: modules
obj-m := hello.o

SRC := $(shell pwd)

modules:
  $(MAKE) -C $(KERNEL_SRC) M=$(SRC) modules $(KBUILD_OPTIONS)

modules_install:
  $(MAKE) -C $(KERNEL_SRC) M=$(SRC) modules_install
```

2. Integrate the module into the Yocto build system.

   See the following example to integrate the kernel modules using Yocto module class.

```
DESCRIPTION = "${SUMMARY}"
LICENSE = "GPL-2.0-only"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/${LICENSE};
md5=801f80980d171dd6425610833a22dbe6"

inherit module

SRC_URI += "file://Makefile \
            file://hello.c  \
            file://COPYING  \
            "
S = "${WORKDIR}"

EXTRA_OEMAKE += "MACHINE='${MACHINE}'"
MAKE_TARGETS = "modules"
MODULES_INSTALL_TARGET = "modules_install"

# Kernel module to be autoloaded
    KERNEL_MODULE_AUTOLOAD += "hello"

# The inherit of module.bbclass will automatically name module
packages with
# "kernel-module-" prefix as required by the oe-core build
```

```
environment.

RPROVIDES_${PN} += "kernel-module-hello"
```

For more information about the out-of-tree module, see Working with Out-of-Tree Modules.

7.10 Disable console log

Disable the kernel console logging in one of the following ways:

- Update the kernel command-line with `quiet`.

  For more information about kernel parameters, see The kernel's command-line parameters.

- Disable `CONFIG_SERIAL_EARLYCON` and `CONFIG_SERIAL_MSM_CONSOLE` in the kernel configuration.
- Add the `console=null` parameter to the kernel command-line arguments.

7.11 Configure the pinctrl driver

The Pinctrl subsystem in the Qualcomm Linux kernel manages and configures pins used for general-purpose input/output (GPIO), interintegrated circuit (I2C), serial peripheral interface (SPI), and other hardware interfaces.

Pinctrl configurations, such as **pin muxing** and **pin groupings**, are managed in the device-specific pinctrl drivers, where the drivers list all the available pins and functions.

For example, the corresponding driver for QCS6490 is available in the `kernel-src/drivers/pinctrl/qcom/pinctrl-sc7280.c` file.

Note: For more information about additional Qualcomm SoC pinctrl drivers, see Pinctrl Drivers.

The following are the pinctrl data objects:

Table: Pinctrl data objects

Variable	Description
`static const struct pinctrl_pin_desc sc7280_pins`	Enumerates all pins and their names

Variable	Description
`static const struct msm_pingroup sc7280_groups`	Defines the available muxed functions for the group of GPIO pins
`enum sc7280_functions`	Lists all the available functions as enum values

For more information about the supported functions of the respective SoC pinctrl binding documentation for QCS6490, see pinctrl binding documentation.

Function selection

For a `sc7280_functions` data object, one or multiple GPIO pins are used as a function and must be registered to the device tree and passed to the right device node.

During system boot, the kernel pinctrl infrastructure registers the functions.

The following example shows the kernel configuration infrastructure:

```
tlmm: pinctrl@f100000 {
    compatible = "qcom,sc7280-pinctrl";
    :
    :
    :
    :
    qup_spi0_data_clk: qup-spi0-data-clk-state {
        pins = "gpio0", "gpio1", "gpio2";
        function = "qup00";
    };

    qup_spi0_cs: qup-spi0-cs-state {
        pins = "gpio3";
        function = "qup00";
    };

    qup_spi1_data_clk: qup-spi1-data-clk-state {
        pins = "gpio4", "gpio5", "gpio6";
        function = "qup01";
    };

    qup_spi1_cs: qup-spi1-cs-state {
        pins = "gpio7";
        function = "qup01";
```

```
    };
    :
    :
    :
    :

};



    spi0: spi@980000 {
        compatible = "qcom,geni-spi";
        reg = <0 0x00980000 0 0x4000>;
        clocks = <&gcc GCC_QUPV3_WRAP0_S0_CLK>;
        clock-names = "se";
        pinctrl-names = "default";
        pinctrl-0 = <&qup_spi0_data_clk>, <&qup_spi0_cs>;
        interrupts = <GIC_SPI 601 IRQ_TYPE_LEVEL_HIGH>;
        #address-cells = <1>;
        #size-cells = <0>;
        power-domains = <&rpmhpd SC7280_CX>;
        operating-points-v2 = <&qup_opp_table>;
        interconnects = <&clk_virt MASTER_QUP_CORE_0 0 &clk_virt
SLAVE_QUP_CORE_0 0>,
                        <&gem_noc MASTER_APPSS_PROC 0 &cnoc2 SLAVE_QUP_0 0>;
        interconnect-names = "qup-core", "qup-config";
        dmas = <&gpi_dma0 0 0 QCOM_GPI_SPI>,
            <&gpi_dma0 1 0 QCOM_GPI_SPI>;
        dma-names = "tx", "rx";
        status = "disabled";
    };
```

7.12 Configure the GPIO usage

GPIO pin configuration requires the following two settings. The settings define a GPIO pin state and make those pins available for any input/output activity.

- Mux: The mux setting requires selecting the function name that's mapped from the set of available functions in the SoC-specific pinctrl driver.

  For more information about pinctrl, see Pinctrl configuration.

- Configuration: The configuration aspect requires setting the drive strength and bias property.

The following examples show how the two settings define the GPIO pin using the following procedures:

1. Define the pin configuration in the device tree:

```
bt_en: bt-en-state {
   pins = "gpio85";
   function = "gpio";
   output-low;
   bias-disable;
};
```

2. Configure the device node or intellectual property (IP) block in the device tree:

```
bluetooth: bluetooth {
   compatible = "qcom,wcn6750-bt";
   pinctrl-names = "default";
   pinctrl-0 = <&bt_en>, <&sw_ctrl>;
   enable-gpios = <&tlmm 85 GPIO_ACTIVE_HIGH>;
   swctrl-gpios = <&tlmm 86 GPIO_ACTIVE_HIGH>;
   vddaon-supply = <&vreg_s7b_0p9>;
   vddbtcxmx-supply = <&vreg_s7b_0p9>;
   vddrfacmn-supply = <&vreg_s7b_0p9>;
   vddrfa0p8-supply = <&vreg_s7b_0p9>;
   vddrfa1p7-supply = <&vreg_s1b_1p8>;
   vddrfa1p2-supply = <&vreg_s8b_1p2>;
   vddrfa2p2-supply = <&vreg_s1c_2p2>;
   vddasd-supply = <&vreg_l11c_2p8>;
   max-speed = <3200000>;
```

3. The driver code must use generic APIs to select and register their GPIO configurations within the pinctrl configurations.

   The following is an example of available APIs:

```
devm_gpiod_get_optional(&serdev->dev, "enable", GPIOD_
OUT_LOW);

/**
* devm_gpiod_get_optional - Resource-managed gpiod_get_
optional()
* @dev: GPIO consumer
* @con_id: function within the GPIO consumer
* @flags: optional GPIO initialization flags
*
* Managed gpiod_get_optional(). GPIO descriptors returned
```

```
from this function
* are automatically disposed on driver detach. See gpiod_
get_optional() for
* detailed information about behavior and return values.
*/

 gpiod_set_value_cansleep(qcadev->bt_en, 0);

/**
* gpiod_set_value_cansleep() - assign a gpio's value
* @desc: gpio whose value will be assigned
* @value: value to assign
*
* Set the logical value of the GPIO, i.e. taking its
ACTIVE_LOW status into
* account
*
* This function is to be called from contexts that can
sleep.
*/
```

GPIO as interrupt request (IRQ)

To set the GPIO as an IRQ, use the following procedure:

1. Configure the GPIO pin in the DTS file:
   a. Set the properties and the function for the GPIO pin.
   b. Set the pin to use GPIO 55 for the `qup_se_l3()` function with the following configurations:

```
qupv3_se3_rx: qupv3-se3-rx-state {
   pins = "gpio55";
   function = "qup03"; // To be taken from available from
functions.
   drive-strength = <2>;
   bias-disable;
};
```

2. Create a DT entry like the previous configuration for the device node where you want set the GPIO as an IRQ.

   In the following example, the GPIO55 is configured as an IRQ with the parent as a top-level mode multiplexer (TLMM) and the level is set to high.

```
interrupts-extended = <&tlmm 55 IRQ_TYPE_LEVEL_HIGH>;
```

3. The driver must read the value and register it as an interrupt to the generic interrupt controller (GIC) using the `request_irq` API specifying the interrupt service register (ISR) and IRQ flags.

```
irq_no = platform_get_irq(pdev, 1);
```

Configure GPIOS to generate clock or pulse width modulation

Configure any GPIO with `GP_CLK` as an alternalte functionality to get clock or pulse width modulation (PWM).

Note: The following procedure is applicable to QCS6490 SoCs.

For more information about how to find a GPIO with the `GP_CLK` function, see the Pin descriptions.

1. Add GPIO configuration node in `kernel/arch/arm64/boot/dts/qcom/sc7280.dtsi` file.

```
+gpio_pwm_default: gpio_pwm_default {
+       mux {
+               pins = "gpio42";
+               function = "gcc_gp1";    // search "gcc_gp"
in "kernel/drivers/pinctrl/qcom/pinctrl-sc7280.c", From this
we can find out which GPIO's has GP_CLK functionality
+       };
+
+       config {
+               pins = "gpio42";
+               bias-disable; /* No PULL */
+               drive-strength = <8>; /* 2 MA */
+       };
+};
```

2. Define devicetree node in `kernel/arch/arm64/boot/dts/qcom/sc7280.dtsi` file.

```
+beeper: beeper {
+       compatible = "gpio-beeper";
+       pinctrl-names = "default";
+       pinctrl-0 = <&gpio_pwm_default>;
+       clocks = <&clock_gcc GCC_GP1_CLK>; //clock_gcc is gcc
clk device node, GCC_GP1_CLK index which defined in "kernel/
```

```
include/dt-bindings/clock/qcom,gcc-sc7280.h"
+       clock-names = "gpio-pwm-clk";
+};
```

3. Add the following code in the device driver:

```
+#include <linux/clk.h>
+#include <linux/io.h>
...
+ struct clk *pclk;
+ struct rcg_clk *gp1_rcg_clk;
+ int ret;
+
+ pclk = devm_clk_get(&pdev->dev, "gpio-pwm-clk");
+ ret = clk_set_rate(pclk, 50000000); // please check the freq
table in kernel/drivers/clk/qcom/gcc-sc7280.c, the freq can be
found in the freq table of GCC_GP1_CLK.
+ if (ret)
+     printk("clk set rate fail, ret = %d\n", ret);
+
+ ret = clk_prepare_enable(pclk);  // By default this will
enable clock as PWM with 50% duty cycle.
+ if (ret)
+     printk("%s: clk_prepare error!!!\n", __func__);
+ else
+     printk("%s: clk_prepare success!\n", __func__);
+
```

4. If you don't use clock or PWM, call `clk_disable_unprepare()` to disable the clock to save power.

Note: Ensure to call `clk_prepare_enable()` first before calling `clk_disable_unprepare()`.

---

5. To generate the required duty cycle, call `clk_set_duty_cycle()` API after `clk_prepare_enable` API.

7.13 Configure ZRAM as a swap device

ZRAM is a compressed swap mechanism that creates a virtual block device in RAM. ZRAM is enabled as a module within the kernel defconfig.

ZRAM is configured in the Yocto build in the

`recipes-extended/zram/zram/zram-swap-init-update` file.

To enable or configure ZRAM, do the following:

```
# check if zram module is loaded
  lsmod | grep zram

# else load it
  modprobe zram

# Configure /dev/zram0 size according to your RAM size
  echo 128M > /sys/block/zram0/disksize

# activate swap
  mkswap /dev/zram0
  swapon /dev/zram0
```

To configure ZRAM as a swap device, see zram: Compressed RAM-based block devices.

7.14 Extend the memory map

To extend the memory map, adjust the addresses and sizes of memory regions in a DTSI file.

Use the following information to extend the carved out regions:

Add the carved out region in the `arch/arm64/boot/dts/qcom/<SoC>-<board>-<variant>.dts` file in the **reserved-memory** node using the following syntax:

```
my_carveout_mem: my_carveout_mem@address {
      reg= <0x0 0xbase_address 0x0 0xsize>;
      no-map;
}
```

For example:

```
my_carveout_mem: my_carveout_mem@d0800000 {
      reg= <0x0 0xd0800000 0x0 0x100000>;
      no-map;
}
```

Table: Syntax for carved out region

Variable	Description
`my_carveout_mem@d0800000`	Indicates the name of the device node. The convention mandates that you append the base address of the memory region to the name.
`my_carveout_mem`	Indicates the label assigned to this node that can be used to reference this node from within the other nodes in the device tree using phandles.
`reg`	Indicates the property, which is a 64-bit value that defines the base and size of the memory region.
`no-map`	Indicates that this region is carved out and the kernel should remove the mapping from its addressable range.

Note:

- None of the regions should overlap. If you must increase a region size, shift all other subsequent regions, and configure them in the device tree to avoid overlapping.
- The kernel expects all memory region boundaries defined for kernel usage to be 1 MB.
- Existing carved out regions are protected by trusted firmware from kernel access. Decreasing their size or deleting them may result in an external abort, leading to a kernel crash.

Add CMA region

To add the CMA region in the `arch/arm64/boot/dts/qcom/<SoC>-<board>-<variant>.dts` file under **reserved-memory** node, use the following syntax:

```
my_cma_mem: my_cma {
   compatible = "shared-dma-pool";
   alloc-ranges = <0x0 0x00000000 0x0 0xffffffff>;
   reusable;
   alignment = <0x0 0x400000>;
   size = <0x0 0x1400000>;
};
```

Table : Syntax for adding CMA region

Parameters	Description
`my_cma_mem`	Label for the CMA node and can be accessed as a phandle. The label `my_cma` is the name of the CMA region.
`shared-dma-pool`	Indicates that this region is a CMA area.
`alloc-ranges`	Indicates if the region should be within a certain memory limit, as some devices can't access memory regions beyond the 32-bit address limit.
`reusable`	Indicates that the kernel can use the memory in this region when it's free.
`alignment`	Indicates any alignment requirements in this region.
`size`	Indicates the size of the region.
`reg`	This is an optional attribute that indicates the fixed region for memory allocation, in the absence of which memory is dynamically allocated at a random address.

7.15 Add custom CMA heaps

To create custom CMA regions using DMA-BUF heaps, use the existing DMA-BUF framework in the Qualcomm Linux kernel.

The Qualcomm Linux kernel exports the `cma_heap_add()` API for the custom DMA-BUF heaps.

```
/**
* cma_heap_add - adds a CMA heap to dmabuf heaps
* @cma:        pointer to the CMA pool to register the heap for
* @data:       unused
*
* Returns 0 on success. Else, returns errno.
*/

int cma_heap_add(struct cma *cma, void *data);
```

The DMA-BUF heap exported to the user space, has the same name as the phandle of the CMA region in the device tree.

To add a custom CMA heap, do the following:

1. To create a custom CMA region see Contiguous memory allocator.
2. In the driver to which you are adding a custom heap:
   a. Parse the device tree for the newly added CMA region.
   b. Add a DMA-BUF heap associated with the driver.

```
int create_my_cma_heap(struct device *dev)
{
   int rc = 0, idx = 0;

   rc = of_reserved_mem_device_init_by_idx(dev, dev->of_node,
0);  // Parse the devicetree for the cma region

   if (rc) {
            pr_err("No reserved DMA memory, ret=%d\n", rc);
            rc = -EINVAL;
            goto err;
   }

   rc = cma_heap_add(dev->cma_area, NULL);  // Add a dmabuf
heap associated with the cma region

   if (rc) {
            pr_err("cma_heap_add failed, ret=%d\n", rc);
            rc = -EINVAL;
            goto err;
   }

err:
   return rc;
}
```

Note: Align CMA regions to the 4 MB address base and size to support page migration. The migration happens at 2 $^{pageblock\_order}$ pages at the page-block level. The page-block order is 10 in the Qualcomm Linux kernel.

The Qualcomm Linux kernel supports the standard Linux scheduler solution. The kernel uses the scheduler to choose the right CPU for task placement based on the CPU energy consumption.

PELT multiplier

In the Linux kernel, the PELT half-life multiplier influences how the system adapts to changing workloads and adjusts CPU frequencies based on historical utilization data. It's a configuration that speeds up the PELT clock and reduces the half-life time, resulting in faster system response.

Configure the PELT multiplier through the kernel command-line argument:

```
kernel.sched_pelt_multiplier=[1, 2, 4] Default value: 1 (half life
32 msec), 2 (half life 16msec), 4(half life 8 msec)]
```

Utilization clamping

Utilization clamping is a scheduler feature that allows user space to manage the performance requirements of tasks.

To perform clamping, configure the following parameters:

- `UCLAMP_MIN`: If it's set to any value `> 0`, the task demand is always greater than or equal to this value. If the actual task demand is more than this value, then the actual demand signal is used, but if the actual task demand is lower than this value, then `UCLAMP_MIN` is reported as the task demand.
- `UCLAMP_MAX`: If it's set to any value `> 0`, the task demand is always less than or equal to this value. If the actual task demand is less than this value, then the actual demand signal is used, but if actual task demand is more than this value, then `UCLAMP_MAX` is reported as task demand.

Note: `UCLAMP_MAX > UCLAMP_MIN`

Use the UCLAMP to influence the scheduler's task placement decisions.

On a heterogeneous system, the scheduler uses the task demand or utilization signal (PELT signal) to classify a task as small or big. Based on the input from the task classification, the scheduler selects small (less computing power) or big (more computing power) CPU cores for task placement with a probable impact on power consumption.

For example, clamp nonimportant (trivial/background/housekeeping) tasks to lower values (`lower UCLAMP_MAX`) to influence the scheduler to place the tasks on the little cluster. Similarly, clamp the important/foreground/active tasks to higher values (`higher UCLAMP_MIN`) to influence the scheduler to place the tasks on the big cluster.

UCLAMP also allows the scheduler to control frequency guidance.

To meet the quick frequency ramp for a task, clamp the task to a higher demand (`UCLAMP_MIN`) value. The higher clamping helps to ramp up the frequency of the cluster to meet the performance requirements of the task.

For more information about the interface and configuration, see Utilization Clamping.

7.16 Change the default CPU frequency governor

The CPU frequency governor can be changed at runtime or set statically during build compilation.

1. Kconfig configuration option: To set the CPU frequency governor, enable the corresponding driver in the kernel defconfiguration file.

   To set `PERFROMANCE` governor as the default CPU frequency governor, set `CONFIG_CPU_FREQ_DEFAULT_GOV_PERFORMANCE=y` in the defconfiguration file.

2. Kernel command-line option: To override the kernel configuration option, add a `cpufreq.default_governor=performance` parameter to the kernel command-line in the `meta-qcom-hwe/conf/machine/include/qcom-<SoC>.inc` file to set the appropriate CPU frequency governor.

7.17 Customize cache and memory DVFS

The mapping between **CPU frequency** and **L3/DDR frequency** is adjusted based on the power or performance requirements.

In DTSI, for each CPUx node, there is an `operating-points-v2 = <&cpux_opp_table>` entry. The `cpux_opp_table` holds a static mapping between CPU, L3, and DDR frequencies.

For example:

```
cpu0_opp_300mhz: opp-300000000 {
   opp-hz = /bits/ 64 <300000000>;
   opp-peak-kBps = <800000 9600000>;
};
```

When CPU 0 operates at 300 MHz, it votes for 9600000 to L3, which translates to 300,000 Hz (9600000/w) L3 frequency. If the vote is for 800,000 Hz to DDR, this results in 200,000 Hz (800000 / w) DDR frequency.

In the equation, ‘w’ represents how many bytes can be written in a single cycle:

- For DDR, ‘w’ is 4 (each channel performs two transactions per cycle, with each transaction being 2 bytes).
- For L3, ‘w’ is 32 (one transaction per cycle at 32 bytes per transaction).

Note: These values are set per channel for DDR, and the mapping relates CPU frequency to the memory controller (MC) channel bandwidth. Adjusting this map table can impact power and

performance characteristics.

For more information about the operating performance points (OPP) framework and syntax, see Generic OPP (Operating Performance Points) Bindings.

7.18 Configure the postboot settings

Postboot settings comprise completing initial setup, configuring memory, and CPU frequency parameters. These steps ensure proper configuration and functionality after the system boots up.

Postboot framework

A postboot script file is used to configure system parameters on the Qualcomm Linux distribution.

Postboot settings are managed through a *systemd service* file.

For more information about systemd, see systemd(1) - Linux manual page.

What's a systemd service?

A systemd service is a background process that starts or stops based on specific conditions.

A `systemd service` file is written to allow systemd to parse, understand, and run as instructed.

For more information about the background process, see background process.

Basic structure of systemd service file

To modify the behavior of systemd based on the requirement, use the following procedure:

The following is the example of the systemd service file:

```
[Unit]
SourcePath=/etc/initscripts/log_restrict.sh
Description=QTI logging service

[Service]
ExecStart=/etc/initscripts/post_boot.sh

[Install]
WantedBy=multi-user.target
```

The `[Unit]` section

In systemd, a unit refers to any resource that the system knows how to operate and manage. It encompasses services, sockets, devices, mount points, and groups of externally created processes.

Each unit is defined using a configuration file called a `unit` file, which has metadata and configuration details. The `[Unit]` section within a unit file provides information about the description and relationships with other units.

The `[Unit]` section has the following fields:

Table: Supported fields for [Unit] section

Fields	Description
`Description`	Human-readable title of the systemd service.
`After`	Set dependency on a service. For example, if you are configuring a WCN3960 web server, you want the server to start after the network is online.
`Before`	Start current service before specified service. In this example, the WCN3960 web server is running before the service for the next cloud is started because the next cloud server depends on the WCN3960 web server.

Fields	Description

The `[Service]` section

The `[Service]` section contains details of the execution and termination of service.

The `[Service]` section has the following fields:

Table: Supported fields for [Service] section

Fields	Description
`ExecStart`	The command that you must run when the service starts. For example, a web server service to start.
`ExecReload`	The optional field that it specifies how a service is restarted. For services that perform disk I/O, it's recommended to kill them and restart the service. Use the `ExecReload` field if you want to have a specific restart mechanism.
`Type`	Indicates the start-up type of a process for a specified systemd service. The options available are `simple`, `exec`, `forking`, `oneshot`, `dbus`, `notify`, and `idle`.

For more information, see Options.

The `[Install]` section

The `[Install]` section handles the installation of a systemd service or unit file. This is used when you run either the `systemctl enable` or `systemctl disable` command for enabling or disabling a service.

The `[Install]` section has the following fields:

- `WantedBy`: Similar to the `After` and `Before` fields of the The [Unit] section. However, it's used to specify systemd-equivalent **run levels**.

  The `default.target` is used when all the system initialization is complete and you are asked to log in. Most user-facing services (like WCN3960, cron, GNOME-stuff) use this target.

  The `shutdown.target` is to run the service just before the device shuts down.

  The `multi-user.target` is to run the service at the time of system boot, as the root user.

  It can be either a target or a service, for example `network.target`.

- `RequiredBy`: This field is similar to `WantedBy`. However, the field specifies **hard dependencies**. If a dependency fails, the service also fails.

Postboot script

The postboot script file is present in the core recipe of the `meta-qcom-hwe` layer.

Run the postboot script as a part of the previous systemd service. All postboot settings are hosted in a `meta-qcom-hwe/recipes-core/initscripts/files/post_boot.sh` script.

```
#!/bin/sh
# Copyright (c) 2023 Qualcomm Innovation Center, Inc. All rights
reserved.
# SPDX-License-Identifier: BSD-3-Clause-Clear

# Apply postboot settings
```

Recipe for postboot systemd service

Use the `initscripts_1.0.bbappend` file to understand the recipe configuration procedure to install the `post_boot` script.

The recipe to install the postboot bash script in `/etc/init.d` is present in the `meta-qcom-hwe/recipes-core/initscripts/initscripts_1.0.bbappend` file.

```
# postboot

inherit systemd externalsrc

FILESEXTRAPATHS:prepend := "${THISDIR}/files:"
SRC_URI:append = " \
   file://post_boot.sh \
   file://logging-restrictions.sh \
   file://log-restrict.service \
   file://post-boot.service \
   "

do_install:append() {
   # postboot
   install -m 0755 ${WORKDIR}/post_boot.sh ${D}${sysconfdir}/
initscripts/post_boot.sh
   install -m 0644 ${WORKDIR}/post-boot.service -D ${D}${systemd_
unitdir}/system/post-boot.service
   ln -sf ${systemd_unitdir}/system/post-boot.service ${D}${systemd_
unitdir}/system/multi-user.target.wants/post-boot.service
```

```
}

S = "${WORKDIR}"

INITSCRIPT_PACKAGES =+ "${PN}-post-boot"
INITSCRIPT_NAME:${PN}-post-boot = "post_boot.sh"

PACKAGES =+ "${PN}-post-boot"
FILES:${PN}-post-boot += "${systemd_unitdir}/system/post-boot.service
${systemd_unitdir}/system/multi-user.target.wants/post-boot.service $
{sysconfdir}/initscripts/post_boot.sh"
```

Scheduler DCVS settings

Scheduler DCVS settings are defined in the `post_boot script` file.

All postboot settings are present in the `meta-qcom-hwe/recipes-core/initscripts/files/post_boot.sh` script.

```
#!/bin/sh
# Copyright (c) 2023 Qualcomm Innovation Center, Inc. All rights
reserved.
# SPDX-License-Identifier: BSD-3-Clause-Clear
```

8 Debug the kernel

Use the following methods to understand the high-level categorization of ways to debug the kernel, and report issues identified with the inherent debug features in the Qualcomm Linux kernel.

For more information about debug, see Qualcomm Linux Debug Guide.

8.1 Capture the serial console logs

Standard boot and kernel logs are accessible using a serial console. The serial console allows live debugging on a live device. The Qualcomm Linux kernel allows the `CONFIG_SERIAL_QCOM_GENI` driver to support UART and console.

Add the following line to the kernel command-line parameter:

```
console=ttyMSM0,115200n8

# as in following line in meta-qcom-hwe/conf/machine/include/qcom-
<SoC>.conf
KERNEL_CMDLINE_EXTRA ?= "root=/dev/disk/by-partlabel/system rw
rootwait console=ttyMSM0,115200n8 pcie_pme=nomsi earlycon"
```

When you rebuild and load the kernel:

- Connect a Micro USB to USB-A type cable as follows:
  - Device > Micro USB====USB-TypeA > host computer
- To connect to a device serial port on the host, use a serial console client on a host computer.
- You can see serial console logs and other kernel logs on serial client.

For more information about the serial console, see Linux Serial Console.

8.2 Configure the console log level

Configure the kernel console log level for extensive or minimal logging to comply with debug or release software requirements.

Use `/proc/sys/kernel/printk` to configure log level to control log messages appearing on the console. Set the log level from 1 to 7.

When you set the log level value to 1, it filters the log to the lowest level. When the log level value is set to 1, only `pr_emerg/KERN_EMERG` printk logs are printed.

When you set the log level value to 7, it enables the highest log level `pr_info/KERN_INFO` and prints all printk logs to the console.

```
echo "1" > /proc/sys/kernel/printk
```

8.3 Display kernel logs

To display the kernel logs, run one of the following commands:

```
dmesg
```

```
cat /proc/kmsg
```

8.4 Display kernel logs since bootup

To display the kernel logs since bootup, run one of the following commands:

```
cat /var/log/messages
```

```
cat /var/log/kern.log
```

8.5 Debug with printk

Use the `printk` technique to debug in the Linux kernel for printing messages and tracing.

The wrappers around printk defined in `include/linux/printk.h` support adding a log level to your log statements. For example:

```
pr_emerg("At line %d: Func: %s\n", __LINE__, __func__);
pr_info("At line");
```

For more information, see Message logging with printk.

8.6 Debug device tree

To debug the device tree, do the following:

1. Clone the kernel code using `devtool modify linux-qcom-custom` command.
2. Make changes in the device tree.
3. Build DTB using `bitbake dtb-qcom-image` command.
4. Flash `build-qcom-wayland/tmp-glibc/deploy/images/qcs6490-rb3gen2-vision-kit/dtb-qcom-image-qcs6490-rb3gen2-vision-kit.rootfs.vfat` (which is same as dtb.bin) to `dtb_a` and `dtb_b` partitions using the `fastboot` command.
5. Restart the device.

8.7 Debug kernel modules

To debug the kernel modules, do the following:

1. Clone the kernel code using the `devtool modify linux-qcom-custom` command.
2. Disable kernel module signing configuration, `CONFIG_MODULE_SIG` using the following `menuconfig` commands:

```
CONFIG_MODULE_SIG
```

```
CONFIG_MODULE_SIG_FORCE
```

Note: This is a one-time step.

3. Make changes in kernel or modules.
4. Re-build kernel modules using `bitbake esp-qcom-image` command.
5. Push `tmp-glibc/deploy/images/qcs6490-rb3gen2-vision-kit/linux-qcs6490-rb3gen2-vision-kit.efi` kernel image to the device at `/tmp` directory, and rename it as `vmlinuz` (kernel binary).
6. Update kernel using `cp /tmp/vmlinuz /boot/ostree/poky-<SHA>/vmlinuz-6.6.52-03343-gdfbbed24a2c5-dirty` command.
7. Push `build-qcom-wayland/tmp-glibc/deploy/images/qcs6490-rb3gen2-vision-kit/modules--6.6-r0-qcs6490-rb3gen2-vision-kit-<DATE>.tgz` kernel modules to the device at `/tmp` directory.

8. Remount root using `mount -o remount,rw /` and `mount -o remount,rw /usr` commands.
9. Unzip using `tar -xzf modules--6.6-r0-qcs6490-rb3gen2-vision-kit-<DATE>.tgz` command.
10. Update the kernel modules using, `cp -rf /tmp/lib/modules/6.6.65-debug-04076-g029659012248-dirty/ /lib/modules/` command.
11. Restart the device.

8.8 Use the log levels

You can print messages using any other log levels defined in `/linux/printk.h`. The console logs are controlled using the log level used in printk and the log level chosen on the device.

Choose the log level according to your use case. You may encounter numerous logs if a lower log level is chosen in a frequently called function.

The following example shows the logs for kernel print levels:

```
pr_info("At func %s\n", __func__);
pr_notice("At func %s\n", __func__);
pr_warn("At func %s\n", __func__);
pr_err("At func %s\n", __func__);
pr_crit("At func %s\n", __func__);
pr_alert("At func %s\n", __func__);
pr_emerg("At func %s\n", __func__);
```

8.9 Enable SSH

Enable a secure shell (SSH) in permissive mode to securely access your host device. For instructions, see Sign in using SSH.

8.10 Retrieve the information from device

You can retrieve the information from your device to debug the kernel issues.

The following is the list of kernel commands:

Table: List of kernel commands

Commands	Description
`$ zcat /proc/config.gz`	Kernel configuration
`$ cat /proc/cmdline`	Kernel command-line parameters
`$ cat /proc/version`	Kernel version
`$ ls /proc/device-tree`	Device tree configuration on device
`$ lsmod`	Loaded modules
`$ cat /proc/interrupts`	Status of interrupts
`$ cat /sys/devices/system/cpu/cpufreq/policyX/scaling_available_frequecies`	Available CPU frequencies where X = cluster
`$ cat /sys/kernel/debug/qcom_socinfo/chip_id`	chip_id using debugfs

For more information about `procfs`, see The /proc Filesystem.

8.11 Enable debugfs

Debugfs is a file system that allows kernel developers to provide kernel information to the user space.

Debugfs is used to access kernel data structures and variables, trace events, debug messages, and statistics.

- Ensure that your kernel configuration has `CONFIG_DEBUG_FS` set (enabled by default).
- If not, you can enable it in your kernel configuration using `menuconfig`.

- Mount debugfs if it's not mounted by default.

```
mount -t debugfs none /sys/kernel/debug
```

8.12 Debug kernel and modules with KGDB

Kernel GNU debugger (KGDB) provides live debugging support for the kernel on a device.

You can configure KGDB to debug core kernel and module issues. KGDB is a source-level debugger used with GDB to debug the Qualcomm Linux kernel.

For more information about KGDB kernel debugging, see Debugging kernel and modules via gdb.

Use KGDB with serial COM port

Enable the following kernel drivers in the kernel configuration file to support KGDB with serial COM port:

```
CONFIG_FRAME_POINTER
CONFIG_KGDB
CONFIG_KGDB_SERIAL_CONSOLE
CONFIG_HAVE_ARCH_KGDB
CONFIG_CONSOLE_POLL
CONFIG_MAGIC_SYSRQ
```

Update the command-line for debug:

```
# Append  KERNEL_CMDLINE_EXTRA in meta-qcom-hwe/conf/machine/include/
qcom-<SoC>.conf with following string
"kgdboc=ttyMSM0,115200n8 kgdbwait nokaslr"

# If waiting during the boot of kernel for gdb connection is not
desired,
# the parameter 'kgdbwait' can be skipped.
```

Disable watchdog

- Disable the watchdog when debugging using KGDB. The SoC resets when the watchdog is enabled.
- To disable the watchdog, add the following kernel command-line parameter:

```
echo 1 > /sys/bus/platform/devices/hypervisor:qcom,gh-watchdog/
disable
```

Methods to enter debug mode

Method 1 - Use `kgdbwait`:

If `kgdbwait` is passed as the command-line parameter, the kernel does the following: 1. Pauses the execution during boot 2. Enters the Debug mode 3. Waits for the connection from remote GDB, with the following message:

```
[ 0.239669] printk: console [ttyMSM0] enabled
[ 1.535669] random: fast init done
[ 1.541411] KGDB: Registered I/O driver kgdboc
[ 2.224804] KGDB: Waiting for connection from remote gdb...
```

Disable the watchdog from the configuration.

Method 2 - Use `Magic SysRq`:

To enter the Debug mode, run `SysRq-G` in the shell, and run the following command:

```
echo g > /proc/sysrq-trigger
```

Method 3 - Whenever you want to hit a KGDB breakpoint in the source code, add the following code snippet to the intended driver file:

```
#include <linux/kgdb.h>   (you may need to include this file )
kgdb_breakpoint(); //call this for adding a break point at compile
time
```

Method 4 - Triggering panic: The kernel automatically enters Debug mode when an exception occurs. To trigger a crash, use the `Magic SysRq` feature.

```
echo c > /proc/sysrq-trigger
```

Connect to the Qualcomm SoC through GDB over COM port

Install GDB `gdb-multiarch: mingw64\bin\gdb-multiarch.exe`.

- GDB for Windows–Install Mingw-w64 through MSYS2.
- GDB for Linux–Install GDB using How to Install GDB.
- Close all other connections to a serial port, while using the COM Port with GDB to connect to the SoC.
- On a Windows host, set the COM port number to a number greater than 16. You can get the port number from the device manager.
- On a Windows host, pass the COM Port as `\\.\com<#>`.
- In the following example, establish a connection from a serial port on a Windows host. The serial port number used is COM17. GDB is disconnected from COM port before connecting PuTTY again. To attach and de-attach GDB commands, run the following commands:

```
gdb-multiarch.exe -b 115200 <path_to_vmlinux>
(gdb) target remote \\.\com17
   Remote debugging using \\.\com17
   warning: multi-threaded target stopped without sending a
thread-id, using first non-exited thread
   [Switching to Thread -2]
   arch_kgdb_breakpoint () at arch/arm64/include/asm/kgdb.h:21
   21      arch/arm64/include/asm/kgdb.h: No such file or
directory.
```

To set the breakpoints, view them, or check CPU registers, run the following commands:

```
# backtrace:
> bt

# set a break point:
break <function_name>
or
break <filename>: line no

# View all current break points:
info break

# View CPU registers:
info reg

# step over:
s

# continue execution:
c
```

Use KGDB to debug kernel modules

KGDB is a kernel patch that allows source-level debugging of a running Qualcomm Linux kernel using the GDB interface.

You must have the `.ko` files and location of the `.text` kernel module files.

To get the address of the `.text` section of the loaded module, run the following command:

```
# from live device
# cat /sys/module/<module>/sections/.text
```

To add a module symbol file, run the following command:

```
(gdb) add-symbol-file <path_to_.ko> <.text_addr>
```

For example, `add-symbol-file /sys/modules/linux/linux.ko 0xc0ae22d0`.

8.13 Ftrace

The ftrace provides tracing utilities that enable system-wide profiling and runtime tracing.

For more information about ftrace, see Function trace.

Enable memory-mapped input/output traces

The memory-mapped I/O (MMIO) traces in the Qualcomm Linux kernel provide information to understand how drivers interact with MMIO devices and their associated hardware states.

Generic MMIO traces use `__raw_{read,write}{b,l,w,q}` accessors to read/write from/to memory-mapped registers.

In the following cases, the device hangs or some undefined behavior leads to device crashes:

Table: Reasons of device crashes

Reasons	Explanation
Unclocked access	If the access to the register space is unclocked, the device crashes.
Protected register space	If the register space is protected and not set for access to the nonsecure world exceptions happen. For example, only exception level (EL3) access is allowed and any EL2/EL1 access is forbidden.
xPU control	xPU memory protection units control access to certain memory or register regions for specific clients.

The previous scenarios result in instant reboot, synchronous errors (SErrors)/network on chip (NoC) issues, or interconnect hangs. `CONFIG_TRACE_MMIO_ACCESS` provides ftrace trace events to log such MMIO register accesses, offering early enablement of trace events, filtering capability, and the ability to dump the ftrace logs on console.

The following is the sample output from the trace buffer when the MMIO traces are enabled:

```
# List all rwmmio trace events
  cat /sys/kernel/tracing/available_events | grep rwmmio

# Enable all rwmmio trace events
  cat /sys/kernel/tracing/available_events | grep rwmmio >> /sys/
kernel/tracing/set_event
```

```
# List all traces
  cat /sys/kernel/tracing/trace
rwmmio_read: gic_peek_irq+0xd0/0xd8 readl addr=0xffff800010040104
rwmmio_write: gic_poke_irq+0xe4/0xf0 writel addr=0xffff800010040184
rwmmio_read: gic_do_wait_for_rwp+0x54/0x90 readl
addr=0xffff800010040000
rwmmio_write: gic_set_affinity+0x1bc/0x1e8 writeq
addr=0xffff800010046130
```

Kprobes

Kprobes allows you to break into any kernel routine and collect debugging and performance information nondisruptively.

The kernel code address is trapped using a handler routine to invoke when the breakpoint is hit. Kprobes is useful during scheduler debug when you generate traces at different scheduling events to understand the system behavior.

For more information, see Kernel Probes.

8.14 Configure GPIOS from the user space

Use the `libgpiod` library from the user space to control the GPIOS for better performance.

1. To compile and push `libgpiod` library from the host computer, do the following:
   a. To install Arm® (Arm64) toolchain, run the following commands:

   ```
   sudo apt install gcc-aarch64-linux-gnu
   ```

   ```
   sudo apt install binutils-aarch64-linux-gnu
   ```

   b. To download and extract the libgpiod source code from libgpiod 1.6.4.tar.xz, run the following commands:

   ```
   wget https://www.kernel.org/pub/software/libs/libgpiod/
   libgpiod-1.6.4.tar.xz
   ```

   ```
   tar xvf libgpiod-1.6.4.tar.xz
   ```

   ```
   cd libgpiod-1.6.4
   ```

   c. To configure the sources for static linking, run the following command:

```
./configure --enable-tools=yes --build x86_64-pc-linux-gnu --
host aarch64-linux-gnu CFLAGS="-static -static-libgcc -Wl,-
static,--start-group,/usr/lib/gcc-cross/aarch64-linux-gnu/7.
5.0/libgcc.a,/usr/lib/gcc-cross/aarch64-linux-gnu/7.5.0/
libgcc_eh.a,/usr/aarch64-linux-gnu/lib/libc.a,--end-group"
```

d. To compile the library, run the following command:

```
make
```

Note: Compiling creates linked binaries.

e. To build statically linked binaries, run the following commands:

```
aarch64-linux-gnu-gcc -static -o tools/gpiodetect tools/
gpiodetect.o tools/tools-common.o -Wl,-L<ABSOLUTE_PATH_TO_
LIBGPIOD>/libgpiod-1.6.4/lib/.libs,-lgpiod,-lpthread,-static
```

```
aarch64-linux-gnu-gcc -static -o tools/gpioget tools/gpioget.
o tools/tools-common.o -Wl,-L<ABSOLUTE_PATH_TO_LIBGPIOD>/
libgpiod-1.6.4/lib/.libs,-lgpiod,-lpthread,-static
```

```
aarch64-linux-gnu-gcc -static -o tools/gpioset tools/gpioset.
o tools/tools-common.o -Wl,-L<ABSOLUTE_PATH_TO_LIBGPIOD>/
libgpiod-1.6.4/lib/.libs,-lgpiod,-lpthread,-static
```

```
aarch64-linux-gnu-gcc -static -o tools/gpiofind tools/
gpiofind.o tools/tools-common.o -Wl,-L<ABSOLUTE_PATH_TO_
LIBGPIOD>/libgpiod-1.6.4/lib/.libs,-lgpiod,-lpthread,-static
```

```
aarch64-linux-gnu-gcc -static -o tools/gpioinfo tools/
gpioinfo.o tools/tools-common.o -Wl,-L<ABSOLUTE_PATH_TO_
LIBGPIOD>/libgpiod-1.6.4/lib/.libs,-lgpiod,-lpthread,-static
```

```
aarch64-linux-gnu-gcc -static -o tools/gpiomon tools/gpiomon.
o tools/tools-common.o -Wl,-L<ABSOLUTE_PATH_TO_LIBGPIOD>/
libgpiod-1.6.4/lib/.libs,-lgpiod,-lpthread,-static
```

2. To push the binaries to your device after compilation, run the `scp` command. For example:

```
scp gpioinfo root@<IP_address>:/path/to/directory/on/device
```

After pushing the binaries to your device, run the following commands on the device to interact with the GPIOs:

1. Use the `gpiodetect` and `gpioinfo` commands to list the GPIO chips and lines. The following example shows the GPIO chip information:

```
sh-5.2# ./gpiodetect
gpiochip0 [c440000.spmi:pmic@8:pinctrl@c00] (12 lines)
gpiochip1 [c440000.spmi:pmic@1:gpio@8800] (10 lines)
gpiochip2 [c440000.spmi:pmic@2:gpio@8800] (9 lines)
gpiochip3 [c440000.spmi:pmic@0:gpio@b000] (4 lines)
gpiochip4 [f100000.pinctrl] (176 lines)
gpiochip5 [33c0000.pinctrl] (15 lines)
```

   The following example shows the GPIO lines:

```
sh-5.2# ./gpioinfo gpiochip5
gpiochip5 - 15 lines:
      line    0:      unnamed       unused   input   active-high
      line    1:      unnamed       unused   input   active-high
      line    2:      unnamed       unused   input   active-high
      line    3:      unnamed       unused   input   active-high
      line    4:      unnamed       unused   input   active-high
      line    5:      unnamed       unused   input   active-high
      line    6:      unnamed       unused   input   active-high
      line    7:      unnamed       unused   input   active-high
      line    8:      unnamed       unused   input   active-high
      line    9:      unnamed       unused   input   active-high
      line   10:      unnamed       unused   input   active-high
      line   11:      unnamed       unused   input   active-high
      line   12:      unnamed       unused   input   active-high
      line   13:      unnamed       unused   input   active-high
      line   14:      unnamed       unused   input   active-high
```

2. Use the `gpioset` command to set GPIO values. For example, to set `GPIO line 0` on `gpiochip4`, do the following:

```
sh-5.2# ./gpioset gpiochip4 0=1
sh-5.2# ./gpioinfo gpiochip4
gpiochip4 - 176 lines:
   line    0:      unnamed       unused   output  active-high
   line    1:      unnamed       unused   input   active-high
```

3. Use the `gpioget` command to read GPIO values. For example, to read the value of `GPIO line 0` on `gpiochip4`, do the following:

```
sh-5.2# ./gpioget gpiochip4 0
1
sh-5.2# ./gpioinfo gpiochip4
gpiochip4 - 176 lines:
        line    0:      unnamed       unused   input   active-
high
        line    1:      unnamed       unused   input   active-
high
```

8.15 Troubleshoot kernel issues

To troubleshoot kernel issues, use the following methods:

Boot failure due to incorrect or no DTB picked

When troubleshooting boot failure due to DTB load issues, verify authentication and DTB availability, and follow the steps to fix the problem.

The following scenarios lead to boot failure:

- **Authentication failure**

  The following example shows the logs for failure due to incorrect authentication:

```
Platform Subtype : 0
DtPlatformLoadDtb qcs6490-rb3gen2.dtb is loaded
Platform Subtype : -2090817768
DtPlatformLoadSign qcs6490-rb3gen2.sgn is loaded
failed to authenticate image !
```

- **Failure due to unavailability of DTB**

  The following example shows the logs for failure due to the missing DTB:

```
DtPlatformLoadDtb qcs6490-rb3gen2.dtb is loading failed with
Status = E
DtPlatformDxeEntryPoint: no DTB blob could be loaded, defaulting
to ACPI (Status == Not Found)
```

  To fix the unavailability of DTB issues, do the following:

  – Verify if the corresponding DTB file is a part of a packaged `efi.bin` file.

  – Mount the `efi.bin` file locally on the host development machine to verify the following:

```
mount -t vfat efi.bin /mnt/

ls -lR /mnt/
```

Serial console not working

To troubleshoot serial console log failure, enable specific drivers in the kernel configuration and add relevant parameters to the kernel boot arguments.

- Enable the following driver in the kernel configuration file:
  - CONFIG_SERIAL_QCOM_GENI=y
  - CONFIG_SERIAL_QCOM_GENI_CONSOLE=y
- Add the following parameter to the kernel boot arguments:

```
console=ttyMSM0,115200n8
```

- To get early boot messages from the kernel, add the following parameter to the kernel boot arguments:

```
earlycon
```

Debug Remoteproc failure

To troubleshoot remoteproc failure, capture kernel logs, use matching firmware signature files, and verify firmware locations in the device.

- Ensure that all the matching subsystem images are flashed without any errors.
- Verify if there are any errors in the kernel log.

  The following example shows a sample kernel log:

```
0x000000000A27652C |   5198.790423:   qcom_q6v5_pas 3000000.
remoteproc: fatal error received: err_inject_crash.c:413:Crash
injected via Diag
0x000000000A276689 |   5198.801061:   remoteproc remoteproc2:
crash detected in 3000000.remoteproc: type fatal error
0x000000000A2767A1 |   5198.809602:   remoteproc remoteproc2:
handling crash #1 in 3000000.remoteproc
0x000000000A27688E |   5198.816837:   remoteproc remoteproc2:
recovering 3000000.remoteproc
0x000000000A276971 |   5198.823784:   qcom_q6v5_pas 8a00000.
remoteproc: subsystem event rejected
```

- Disable the following subsystem restart feature to ensure that the remoteproc crash signature is visible in the kernel log:

```
echo disabled > /sys/kernel/debug/remoteproc/remoteprocN/
recovery
```

- Confirm if all the necessary firmware files are present in the `/lib/firmware/qcom/<SoC> in rootfs` file system.

Debug configurations or symbols not reflecting in images

To debug kernel configuration change issues where configurations or symbols aren't reflecting in the images, follow these steps:

- Add the debug configuration driver to the `arch/arm64/configs/qcom_debug.config` file.
- Export the `DEBUG_BUILD=1` macro before running the `bitbake` command.

Note: The previous debug configuration file pertains to a custom BSP variant. Add the changes to the `qcom.cfg` file for the base BSP variant.

System memory is too low

To debug the cases where a system runs out of free memory, see Out of memory.

Identify DTB during boot

When the device boots, the boot loader verifies the following IDs and loads the corresponding DTB file:

```
qcom,msm-id = <x z>;
qcom,board-id = <y y'>;
```

- x = ID for SoC
- z = ID for SoC revision (reserved field)
- y = ID for CDP, MTP (hardware variants), and platform
- y' = ID for subtype (assumed 0 if absent)

Troubleshooting virtio issues

Use the following methods to troubleshoot virtio issues:

- Ensure that the kernel is compiled with the required configurations.
- Verify the QEMU command-line options or libvirt XML configuration.
- Verify the system logs for errors.
- Enable traces in the QEMU command-line for virtio using the available trace back ends. For more information about KVM traces, see KVM traces.

9 References

9.1 Related documents

Title	Number
Qualcomm Technologies, Inc.	
Qualcomm Linux Debug Guide	80-70018-12
Qualcomm Linux Build Guide	80-70018-254
Qualcomm Linux Boot Guide	80-70018-4
Qualcomm Linux Yocto Guide	80-70018-27
Resources	
Linux kernel-EFI	https://docs.kernel.org/admin-guide/efi-stub.html
Linux kernel API doc	https://www.kernel.org/doc/html/latest/driver-api/gpio/index.html
Memory	https://docs.kernel.org/core-api/index.html#memory-management
Scheduler	https://docs.kernel.org/scheduler/sched-energy.html
CPU topology	https://docs.kernel.org/scheduler/sched-capacity.html
DT configuration setting	https://www.kernel.org/doc/Documentation/devicetree/bindings/gpio/gpio.txt
DT pin configuration	https://www.kernel.org/doc/Documentation/devicetree/bindings/pinctrl/pincfg-node.yaml
PELT	https://lwn.net/Articles/531853/
SchedUtil governor	https://docs.kernel.org/scheduler/schedutil.html
UCLAMP	https://docs.kernel.org/scheduler/sched-util-clamp.html
DVFS	https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt
Remoteproc framkework	https://www.kernel.org/doc/Documentation/remoteproc.txt
Serial console	https://docs.kernel.org/admin-guide/serial-console.html
Debugging with printk	https://www.kernel.org/doc/html/next/core-api/printk-basics.html
File systems	https://docs.kernel.org/filesystems/proc.html
Debugfs	https://www.kernel.org/doc/html/next/admin-guide/dynamic-debug-howto.html

Title	Number
GDB	https://docs.kernel.org/dev-tools/gdb-kernel-debugging.html
Debug memory	https://www.kernel.org/doc/html/v5.0/vm/page_owner.html
Slub Debug	https://www.kernel.org/doc/Documentation/vm/slub.txt
Kernel configuration	https://docs.yoctoproject.org/4.3.1/kernel-dev/common.html#configuring-the-kernel

9.2 Acronyms and terms

Acronym or term	Definition
ASMP	Asymmetric multiprocessing
APSS	Applications processor subsystem
aDSP	Audio digital signal processor
BSP	Board support package
CID	Context identifier
CMA	Contiguous memory allocator
cDSP	Compute digital signal processor
Chardev	Character device
COFF	Common object file format
crosvm	The ChromeOS virtual machine monitor
DDR	Double data rate
DMA	Direct memory access
DMIPS	Dhrystone million instructions per second
DTSI	Device tree source inclusion
DTBO	DTB overlays
DTSO	Device tree source overlay
DVFS	Dynamic voltage and frequency scaling
EAS	Energy-aware scheduling
EFI	Extensible firmware interface
ELF	Executable and linking format
EM	Energy model
ESP	EFI system partition
FAT	File allocation table
GCC	GNU compiler collection
GIC	Generic interrupt controller
GPIO	General-purpose input/output
HYP	Hypervisor
I2C	Interintegrated circuit
Initrd	Initial RAM disk

Acronym or term	Definition
IRQ	Interrupt request
IPC	Interprocess communication
ISR	Interrupt service routine
KVM	Kernel-based virtual machine
L3	Level 3 cache
LLCC	Last level cache controller
LPASS	Low-power audio subsystem
LTS	Long-term support
MPSS	Modem peripheral subsystem software
MTP	Mobile test platform
OS	Operating system
PAS	Peripheral authentication service
PBL	Primary boot loader
PCI	Peripheral component interconnect
PE	Portable executable file format
PELT	Per entity load tracking
PD	Protection domain
PIL	Peripheral image loader
PSCI	Power state coordination interface
PWM	Pulse width modulation
QCAP	Qualcomm crash analysis portal
QPST	Qualcomm Product Support Tool
QXDM	Qualcomm extensible diagnostic monitor
QEMU	Quick emulator
RPE	Reset, power, error
RTOS	Real-time operating system
SLPI	Sensor low-power island
SMD	Shared memory driver
SMP2P	Shared memory point to point
SoC	System on chip
SPI	Serial peripheral interface
SSR	Subsystem restart
SVM	Secondary virtual machine
TLMM	Top-level mode multiplexer
TTY	Teletype
UCLAMP	Utilization clamping
UEFI	Unified extensible firmware interface
UKI	Unified kernel image
URI	Uniform resource identifier
VHE	Virtual host extensions

Acronym or term	Definition
VFIO	Virtual function I/O
VM	Virtual machine
VMM	Virtual machine monitor
WCNSS	Wireless connect subsystem
WDOG	Watchdog service
WLAN	Wireless local area network
WPSS	Wireless processor subsystem
XBL	eXtensible Boot Loader

LEGAL INFORMATION

Your access to and use of this material, along with any documents, software, specifications, reference board files, drawings, diagnostics and other information contained herein (collectively this "Material"), is subject to your (including the corporation or other legal entity you represent, collectively "You" or "Your") acceptance of the terms and conditions ("Terms of Use") set forth below. If You do not agree to these Terms of Use, you may not use this Material and shall immediately destroy any copy thereof.

1) Legal Notice.

This Material is being made available to You solely for Your internal use with those products and service offerings of Qualcomm Technologies, Inc. ("**Qualcomm Technologies**"), its affiliates and/or licensors described in this Material, and shall not be used for any other purposes. If this Material is marked as "**Qualcomm Internal Use Only**", no license is granted to You herein, and You must immediately (a) destroy or return this Material to Qualcomm Technologies, and (b) report Your receipt of this Material to qualcomm.support@qti.qualcomm.com. This Material may not be altered, edited, or modified in any way without Qualcomm Technologies' prior written approval, nor may it be used for any machine learning or artificial intelligence development purpose which results, whether directly or indirectly, in the creation or development of an automated device, program, tool, algorithm, process, methodology, product and/or other output. Unauthorized use or disclosure of this Material or the information contained herein is strictly prohibited, and You agree to indemnify Qualcomm Technologies, its affiliates and licensors for any damages or losses suffered by Qualcomm Technologies, its affiliates and/or licensors for any such unauthorized uses or disclosures of this Material, in whole or part.

Qualcomm Technologies, its affiliates and/or licensors retain all rights and ownership in and to this Material. No license to any trademark, patent, copyright, mask work protection right or any other intellectual property right is either granted or implied by this Material or any information disclosed herein, including, but not limited to, any license to make, use, import or sell any product, service or technology offering embodying any of the information in this Material.

THIS MATERIAL IS BEING PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, WHETHER EXPRESSED, IMPLIED, STATUTORY OR OTHERWISE. TO THE MAXIMUM EXTENT PERMITTED BY LAW, QUALCOMM TECHNOLOGIES, ITS AFFILIATES AND/OR LICENSORS SPECIFICALLY DISCLAIM ALL WARRANTIES OF TITLE, MERCHANTABILITY, NON-INFRINGEMENT, FITNESS FOR A PARTICULAR PURPOSE, SATISFACTORY QUALITY, COMPLETENESS OR ACCURACY, AND ALL WARRANTIES ARISING OUT OF TRADE USAGE OR OUT OF A COURSE OF DEALING OR COURSE OF PERFORMANCE. MOREOVER, NEITHER QUALCOMM TECHNOLOGIES, NOR ANY OF ITS AFFILIATES AND/OR LICENSORS, SHALL BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY EXPENSES, LOSSES, USE, OR ACTIONS HOWSOEVER INCURRED OR UNDERTAKEN BY YOU IN RELIANCE ON THIS MATERIAL.

Certain product kits, tools and other items referenced in this Material may require You to accept additional terms and conditions before accessing or using those items.

Technical data specified in this Material may be subject to U.S. and other applicable export control laws. Transmission contrary to U.S. and any other applicable law is strictly prohibited.

Nothing in this Material is an offer to sell any of the components or devices referenced herein.

This Material is subject to change without further notification.

In the event of a conflict between these Terms of Use and the *Website Terms of Use* on www.qualcomm.com, the *Qualcomm Privacy Policy* referenced on www.qualcomm.com, or other legal statements or notices found on prior pages of the Material, these Terms of Use will control. In the event of a conflict between these Terms of Use and any other agreement (written or click-through, including, without limitation any non-disclosure agreement) executed by You and Qualcomm Technologies or a Qualcomm Technologies affiliate and/or licensor with respect to Your access to and use of this Material, the other agreement will control.

These Terms of Use shall be governed by and construed and enforced in accordance with the laws of the State of California, excluding the U.N. Convention on International Sale of Goods, without regard to conflict of laws principles. Any dispute, claim or controversy arising out of or relating to these Terms of Use, or the breach or validity hereof, shall be adjudicated only by a court of competent jurisdiction in the county of San Diego, State of California, and You hereby consent to the personal jurisdiction of such courts for that purpose.

2) Trademark and Product Attribution Statements.

Qualcomm is a trademark or registered trademark of Qualcomm Incorporated. Arm is a registered trademark of Arm Limited (or its subsidiaries) in the U.S. and/or elsewhere. The Bluetooth® word mark is a registered trademark owned by Bluetooth SIG, Inc. Other product and brand names referenced in this Material may be trademarks or registered trademarks of their respective owners.

Snapdragon and Qualcomm branded products referenced in this Material are products of Qualcomm Technologies, Inc. and/or its subsidiaries. Qualcomm patented technologies are licensed by Qualcomm Incorporated.