Qualcomm Linux Yocto Guide

80-70018-27 AA

April 9, 2025

Contents

1 Overview

This Qualcomm® Linux® Yocto Guide provides information about how Qualcomm Linux uses Yocto to build an embedded system software image for Qualcomm reference hardware.

For an introduction to Yocto, see the following pages created by the Yocto Project®:

- The Yocto Project
- The Yocto Project 5.0.6 documentation

The Yocto Project provides tools, software, configurations, and best practices to create customized Linux images for embedded and IoT devices, or anywhere a customized Linux OS is needed.

The Qualcomm Linux release and this documentation are built on the principles of the Yocto Project to help you further customize Qualcomm Linux.

Qualcomm Linux offers two build variants: `base` and `custom`.

- The `base` variant allows you to build a system software image using upstream software components. However, there are a few exceptions for Qualcomm software that's still being upstreamed or is necessary to boot the reference hardware.
- The `custom` variant allows you to build a system software image that includes Qualcomm value-added software.

For more information about these variants, see Use of BitBake OVERRIDES in Qualcomm Linux metadata layers. The machine configurations defined in meta-qcom-hwe can use these two variants, and these configurations are mapped to the development kits.

Note: To find the mapping of supported overrides and machine configurations to a development kit, see Section 4.2 of the Qualcomm Linux Release Notes.

For information about the supported features, see Qualcomm Linux features.

For information about how to customize Qualcomm Linux, see User customizations.

For diagnosis and troubleshooting procedures for commonly encountered problems with Yocto workspaces, see Debug.

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

Note: This guide uses the QCM6490 and QCS6490 hardware SoCs interchangeably. The `qcs6490-rb3gen2-core-kit.conf` and `qcs6490-rb3gen2-vision-kit.conf` machine configuration files defined in the `meta-qcom-hwe/conf/machine/` directory support the QCM6490, QCS6490, and QCS5430 hardware SoCs.

2 Qualcomm Linux features

Qualcomm Linux features include:

- Metadata layers
- Recipes and configurations in the metadata layers
- Software features to enhance Qualcomm Linux
- Tools introduced by Qualcomm
- Instructions on how to build Qualcomm Linux images

For a comprehensive document on the Yocto Project Scarthgap release, see Yocto Project documentation.

Note:

- Qualcomm Linux is based on the Yocto Scarthgap release.
- When you read this document, the Yocto Project may have released additional Scarthgap LTS point releases. This document refers to the version that was current at the time of this release.

If you are new to the Yocto Project, see the build instructions in the Yocto Project documentation.

The Qualcomm Linux environment consists of several community-maintained metadata layers that deliver recipes for software packages, package groups, image recipes, and configurations. The community layers stack the Qualcomm Linux metadata layers to provide additional software components required for Qualcomm development kits.

2.1 Qualcomm Linux metadata layers overview

This section introduces the layers included in the Qualcomm manifest. This manifest includes all the layers required to reproduce the reference build. The subsequent sections introduce the layers maintained by the Yocto Project. Qualcomm maintains the layers specific to Qualcomm development kits, which depend on the community layers to realize full functionality.

The following figure shows the layers included in the Qualcomm Linux release:

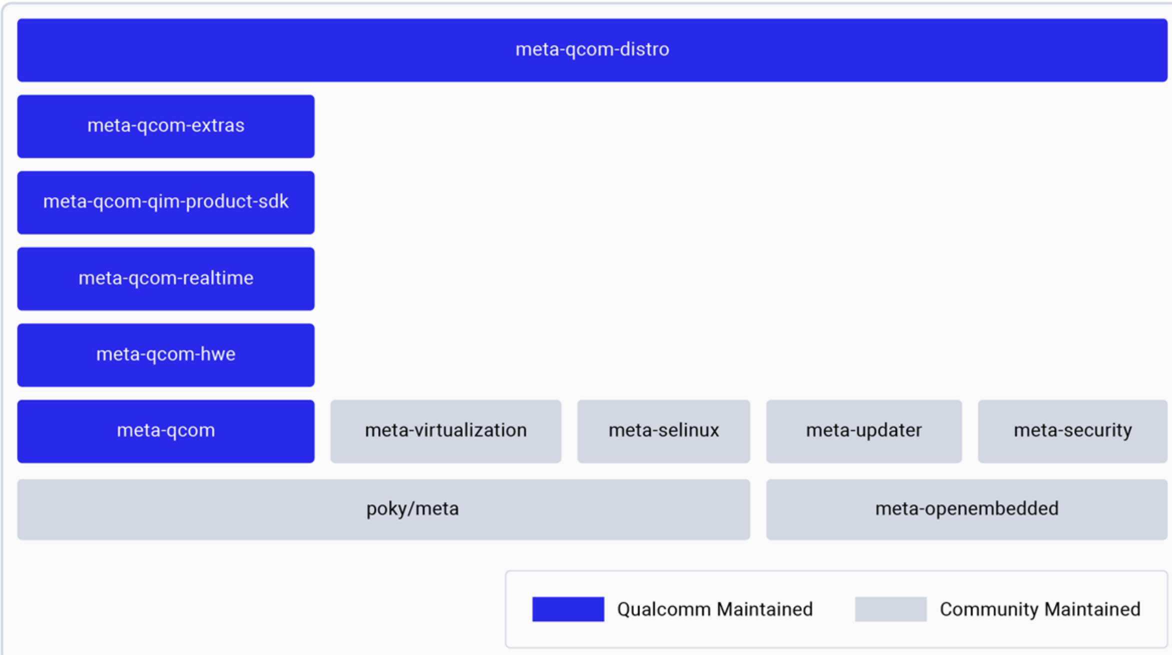


Figure : Qualcomm Linux metadata layers

Table : Qualcomm Linux metadata layers and descriptions

Metadata layer	Description
`meta-qcom-hwe`	Contains recipes that build software components for Qualcomm development kits and provides value-added software features applicable to Qualcomm SoCs.
`meta-qcom-distro`	Provides a reference distribution configuration for Qualcomm development kits. Image recipes and package groups are defined in this layer.
`meta-qcom-extras`	Optional metadata layer for registered users. This layer enables source compilation of select components, which are otherwise provided as binaries in `meta-qcom-hwe`.
`meta-qcom-qim-product-sdk`	Provides Qualcomm® Intelligent Multimedia SDK (IM SDK) and AI SDKs based on the GStreamer framework. This includes a set of GStreamer plug-ins, sample applications for multimedia, and AI use cases.
`meta-qcom-realtime`	Provides patches and configurations for the Qualcomm Linux kernel to enable real-time operations.

Metadata layer	Description
`meta-qcom`	Contains Qualcomm device support and upstream open-source software (OSS) components.
`meta-virtualization` For more information, see meta-virtualization.	Contains packages for constructing OpenEmbedded virtualized solutions and virtualization stacks, such as Docker and Kubernetes.
`meta-selinux` For more information, see meta-selinux.	Enables SELinux support. This layer includes reference SELinux policies and provides necessary tooling. To enable SELinux for Qualcomm Linux, set the variable `DEFAULT_ENFORCING` to `enforcing` in `meta-qcom-distro/conf/distro/include/qcom-base.inc`.
`poky/meta` For more information, see poky/meta.	Provides build tools and recipe files that provide various software components needed for an embedded OS distribution.
`meta-openembedded` For more information, see meta-openembedded.	Collection of layers for the OpenEmbedded build system.
`meta-security` For more information, see meta-security.	Provides security tools and hardening tools for Qualcomm Linux kernels and libraries for implementing security mechanisms.
`meta-updater` For more information, see meta-updater.	Enables over-the-air updates (OTA) with OSTree. OSTree is a tool for atomic full file system upgrades with rollback capability.

For information about robotics layers, see Qualcomm Intelligent Robotics Product (QIRP) SDK layers.

Qualcomm Linux metadata layers

The following layers represent the Qualcomm BSP metadata:

- meta-qcom
- meta-qcom-hwe
- meta-qcom-realtime

The following layer defines the `qcom-wayland` reference distribution:

- meta-qcom-distro

The following layer defines the optional Qualcomm IM SDK:

- meta-qcom-qim-product-sdk

The following metadata layer defines the optional BSP:

- meta-qcom-extras

Use of BitBake OVERRIDES in Qualcomm Linux metadata layers

The Qualcomm Linux metadata layers use the OVERRIDES mechanism of BitBake to implement two distinct BSP variants referred to as `base` and `custom`.

When initiating Qualcomm Linux builds, set the `QCOM_SELECTED_BSP` variable to either `custom` or `base`. This variable is defined in `conf/machine/include/qcom-base.inc` and is set to `custom` by default as follows:

```
QCOM_SELECTED_BSP ??= "custom"
```

The value set for `QCOM_SELECTED_BSP` translates into two BitBake OVERRIDES. The following is a mapping table of `QCOM_SELECTED_BSP` to the corresponding BitBake OVERRIDE:

`QCOM_SELECTED_BSP` value	Corresponding BitBake OVERRIDE
`base`	`qcom-base-bsp`
`custom`	`qcom-custom-bsp`

The effective value of the BitBake OVERRIDE sets variables selectively, resulting in inclusion of packages in the image.

Note: To familiarize yourself with BitBake OVERRIDES, see Conditional syntax (Overrides) and Yocto Project documentation.

To utilize the `base` variant of the BSP and how to set `QCOM_SELECTED_BSP` at build time, see Build base image.

Note: In this release, the `base` BSP supports Qualcomm reference devices for building with `qcs8300-ride-sx.conf`, `qcs9075-ride-sx.conf`, `qcs9100-ride-sx.conf`.

The final image composition depends on whether you choose `base` or `custom` value for the `QCOM_SELECTED_BSP` variable, see Overview.

To find examples of how these OVERRIDES select the software components to be built, search the `meta-qcom-hwe` codebase using `qcom-base-bsp` and `qcom-custom-bsp`.

meta-qcom

The meta-qcom metadata layer is hosted at git.yoctoproject.org and provides recipes to build the Qualcomm OSS. Build the software image using the following recipes from the meta-qcom layer.

recipes-devtools/qdl/qdl_git.bb	The Qualcomm Download (QDL) flashing tool communicates with the USB devices displaying ID 05c6:9008 to upload a flash loader and uses this recipe to flash the images.
recipes-support/pd-mapper/pd-mapper_git.bb	The Qualcomm pd-mapper is an implementation for the protection domain mapper service. This service configures and manages protection domains, ensuring secure communication between applications and various remote processors.
recipes-support/qrtr/qrtr_git.bb	The Qualcomm Router (QRTR) is an inter-process communication mechanism used in Qualcomm SoCs. It allows communication between different processors in the system, like the application processor and the modem, using a socket-style programming interface in the user space.
recipes-support/initrdscripts/initramfs-module-copy-modules_1.0.bb	initramfs-framework module for copying kernel modules from initramfs to rootfs.

File	Description

meta-qcom-hwe

The `meta-qcom-hwe` metadata layer is available on GitHub. It provides additional software support for enabling Qualcomm devices.

- **BitBake classes**

  For an introduction to the BitBake classes, see Classes.

File	Description
`classes/qprebuilt.bbclass`	Implements logic to use a prebuilt package instead of fetching and compiling the source. For any recipe inheriting **qprebuilt**, this class unpacks the binary packed in a *tar.gz* archive and provides these binaries for packaging using a BitBake task.
`classes/qmodule.bbclass`	By default, the Qualcomm BSP enforces signing the kernel module by enabling `CONFIG_MODULE_SIG_FORCE` in the kernel. However, some out-of-tree modules may not be signed. To avoid module loading issues, `qmodule.bbclass` inspects all packages providing kernel modules and signs them if they're not already signed.
`classes/image_types_ota_sdboot.bbclass`	This BitBake class implements image creation according to requirements of OSTree over-the-air upgrade system.

- **Machine configurations**

  The Qualcomm Linux machine configuration files are available on GitHub.

  The files located in the `meta-qcom-hwe/conf/machine/include` directory define and set the required BitBake variables, which can be commonly used by machine configurations defined by the Qualcomm BSP. The following table provides an overview of these files.

File	Description
`meta-qcom-hwe/conf/machine/include/qcom-base.inc`	This file sets the BitBake variables that are commonly shared by all machine configurations defined in the Qualcomm BSP. For example, `SOC_ARCH`, `PREFERRED_PROVIDER` and `IMAGE_FSTYPES`.

File	Description
`meta-qcom-hwe/conf/machine/include/qcom-qcs6490.inc`	This file sets the `SOC_FAMILY` variable to `qcm6490`. This allows Qualcomm Linux recipes to use the `qcm6490` OVERRIDE to implement changes specific to the Qualcomm BSP. This file also defines configuration variables that are shared by all Qualcomm development kits based on QCS6490.
`meta-qcom-hwe/conf/machine/include/qcom-qcs9100.inc`	This file sets the `SOC_FAMILY` variable to `qcs9100`. This allows Qualcomm Linux recipes to use the `qcs9100` OVERRIDE to implement changes specific to the Qualcomm BSP. This file also defines configuration variables that are shared by all Qualcomm development kits based on QCS9100.
`meta-qcom-hwe/conf/machine/include/qcom-qcs8300.inc`	This file sets the `SOC_FAMILY` variable to `qcs8300`. This allows Qualcomm Linux recipes to use the `qcs8300` OVERRIDE to implement changes specific to the Qualcomm BSP. This file also defines configuration variables that are shared by all Qualcomm development kits based on QCS8300.
`meta-qcom-hwe/conf/machine/include/qcom-qcs615.inc`	This file sets the `SOC_FAMILY` variable to `qcs615`. This allows Qualcomm Linux recipes to use the `qcs615` OVERRIDE to implement changes specific to the Qualcomm BSP. This file also defines configuration variables that are shared by all Qualcomm development kits based on QCS615.

Machine configuration files for development kits based on QCS6490

Note: For Qualcomm Linux 1.2 and subsequent releases, you must use the new machine configuration files listed in the following table. Qualcomm Linux 1.2 has replaced the previous machine configuration file, `qcm6490.conf`, from Qualcomm Linux 1.1.

Configuration file	Description
`conf/machine/qcm6490-idp.conf`	This file is for the integrated development platform (IDP) with QCM6490.
`conf/machine/qcs6490-rb3gen2-core-kit.conf`	This file is for the QCS6490-based Qualcomm® RB3 Gen 2 Core development kit.

Configuration file	Description
`conf/machine/qcs6490-rb3gen2-vision-kit.conf`	This file is for the QCS6490-based Qualcomm® RB3 Gen 2 Vision development kit with low-/high-resolution CSI cameras.
`conf/machine/qcs6490-rb3gen2-industrial-kit.conf`	This file is for the QCS6490-based Qualcomm® RB3 Gen 2 Industrial development kit.

Machine configuration files for development kits based on QCS9075 and QCS9100

Configuration file	Description
`conf/machine/qcs9100-ride-sx.conf`	This file is for the QCS9100-based Qualcomm® IQ9 Beta Evaluation Kit (EVK).
`conf/machine/qcs9075-ride-sx.conf`	This file is for the QCS9075-based Qualcomm IQ9 Beta EVK.
`conf/machine/qcs9075-rb8-core-kit.conf`	This file is for the QCS9075-based Qualcomm Dragonwing™ IQ-9075 EVK.
`conf/machine/qcs9075-rb8-core-kit-interface-plus-mezz.conf`	This file is for the QCS9075-based Dragonwing IQ-9075 EVK.

Machine configuration files for development kits based on QCS8300

Configuration file	Description
`conf/machine/qcs8300-ride-sx.conf`	This file is for the QCS8300-based Qualcomm® IQ8 Beta EVK.

Machine configuration files for development kits based on QCS615

Configuration file	Description
`conf/machine/qcs615-adp-air.conf`	This file is for the QCS615-based Qualcomm® IQ6 Beta EVK.

Use of OVERRIDES in `meta-qcom-hwe`

The Qualcomm BSP uses BitBake OVERRIDES to define two different methods for building the BSP.

The metadata layer derives `MACHINEOVERRIDES` from `QCOM_SELECTED_BSP` in the `meta-qcom-hwe/conf/machine/include/qcom-base.inc` file as follows:

```
MACHINEOVERRIDES =. "qcom-${QCOM_SELECTED_BSP}-bsp:"
```

The two MACHINEOVERRIDES in `meta-qcom-hwe` are `qcom-custom-bsp` and `qcom-base-bsp`. These OVERRIDES are used by machine configuration files, recipes, and other configuration files within the `meta-qcom-hwe` metadata layer. The configuration files and recipes use these OVERRIDE constructs to conditionally set variables and append tasks as needed.

Recipes and configuration files use MACHINEOVERRIDES to determine whether the final image is composed of Qualcomm custom BSP software components or upstream software components, based on the effective OVERRIDE at build time.

For example, PREFERRED_PROVIDER for `egl`, `libgl`, `libgles1`, and `libgles2` is set as follows:

Effective OVERRIDE	PREFERRED_PROVIDER for egl, libgl, libgles1, libgles2
qcom-custom-bsp	adreno
qcom-base-bsp	mesa

In the BitBake code snippet, the OVERRIDES conditionally set the variables as follows:

```
# Provider for Graphics Library.
# qcom-base-bsp uses 'mesa' as GL provider
GL_PROVIDER ?= "adreno"
GL_PROVIDER:qcom-base-bsp ?= "mesa"

PREFERRED_PROVIDER_virtual/egl      = "${GL_PROVIDER}"
PREFERRED_PROVIDER_virtual/libgl    = "${GL_PROVIDER}"
PREFERRED_PROVIDER_virtual/libgles1 = "${GL_PROVIDER}"
PREFERRED_PROVIDER_virtual/libgles2 = "${GL_PROVIDER}"

PREFERRED_PROVIDER_virtual/egl-native      = "mesa-native"
PREFERRED_PROVIDER_virtual/libgl-native    = "mesa-native"
PREFERRED_PROVIDER_virtual/libgles1-native = "mesa-native"
PREFERRED_PROVIDER_virtual/libgles2-native = "mesa-native"
```

Note: In this release, the `qcom-base-bsp` OVERRIDE is built only for `qcs9100-ride-sx.conf` and `qcs8300-ride-sx.conf`.

– **Kernel cmdline**

The `meta-qcom-hwe/conf/machine/include/qcom-qcs6490.inc`, `meta-qcom-hwe/conf/machine/include/qcom-qcs9100.inc` and `meta-qcom-hwe/conf/machine/include/qcom-qcs8300.inc` include files use the `KERNEL_CMDLINE_EXTRA` variable to set the kernel command-line parameters as follows:

qcom-qcs6490.inc

The following snippet is from the `qcom-qcs6490.inc` file:

```
# Additional Kernel cmdline parameters for debug builds
DBG_CMDLINE = "${@oe.utils.conditional('DEBUG_BUILD','1',
'earlycon page_owner=on qcom_scm.download_mode=1 slub_
debug=FZP,zs_handle,zspage;FZPU','',d)}"

KERNEL_CMDLINE_EXTRA ?= "pcie_pme=nomsi kernel.sched_pelt_
multiplier=4 rcupdate.rcu_expedited=1 rcu_nocbs=0-7 kpti=off
kasan=off kasan.stacktrace=off no-steal-acc ${DBG_CMDLINE}
swiotlb=128 mitigations=auto net.ifnames=0"
```

qcom-qcs9100.inc

The following snippet is from the `qcom-qcs9100.inc` file:

```
# Additional Kernel cmdline parameters for debug builds
DBG_CMDLINE = "${@oe.utils.conditional('DEBUG_BUILD','1',
'earlycon reboot=panic_warm page_owner=on qcom_scm.download_
mode=1 slub_debug=FZP,zs_handle,zspage;FZPU','',d)}"

KERNEL_CMDLINE_EXTRA ?= "pcie_pme=nomsi net.ifnames=0
pci=noaer kpti=off kasan=off kasan.stacktrace=off swiotlb=128
${DBG_CMDLINE} mitigations=auto kernel.sched_pelt_multiplier=4
rcupdate.rcu_expedited=1 rcu_nocbs=0-7 no-steal-acc vfio_
iommu_type1.allow_unsafe_interrupts=1 fw_devlink.strict=1"
```

qcom-qcs8300.inc

The following snippet is from the `qcom-qcs8300.inc` file:

```
# Additional Kernel cmdline parameters for debug builds
DBG_CMDLINE = "${@oe.utils.conditional('DEBUG_BUILD','1',
'earlycon reboot=panic_warm page_owner=on qcom_scm.download_
mode=1 slub_debug=FZP,zs_handle,zspage;FZPU','',d)}"

KERNEL_CMDLINE_EXTRA ?= "pcie_pme=nomsi net.ifnames=0
pci=noaer kpti=off kasan=off kasan.stacktrace=off swiotlb=128
${DBG_CMDLINE} mitigations=auto kernel.sched_pelt_multiplier=4
rcupdate.rcu_expedited=1 rcu_nocbs=0-7 no-steal-acc arm64.
nopauth  fw_devlink.strict=1"
```

– **Include DTB**

Set the `KERNEL_DEVICETREE` variable conditionally, using the OVERRIDES mechanism to ensure that the correct device tree binary (DTB) is included. This is managed within each individual machine configuration file in `meta-qcom-hwe/conf/machine`.

Set the `KERNEL_DEVICETREE` variable differently for `custom` and `base` variants in the machine configuration file. The following example from `conf/machine/qcs9100-ride-sx.conf` shows how the machine configuration file selects a DTB:

linux-qcom-base

The following code block shows how `KERNEL_DEVICETREE` is set for `linux-qcom-base`, resulting in the `base` variant using the upstream DTB.

```
KERNEL_DEVICETREE:pn-linux-qcom-base = " \
                     qcom/qcs9100-ride.dtb \
                     qcom/qcs9100-ride-r3.dtb \
                     qcom/sa8775p-ride.dtb \
                     qcom/sa8775p-ride-r3.dtb \
                     "
```

linux-qcom-custom

The following code block shows how `KERNEL_DEVICETREE` is set for `linux-qcom-custom`, resulting in the `custom` variant using the upstream DTB.

```
KERNEL_DEVICETREE:pn-linux-qcom-custom = " \
                    qcom/qcs9100-addons-ride.dtb \
                    qcom/qcs9100-addons-ride-r3.
dtb \
                    qcom/sa8775p-addons-ride.dtb \
                    qcom/sa8775p-addons-ride-r3.
dtb \
                    "
```

– **Include additional DTBOs**

To include an additional device-tree overlay (DTBO) to be overlaid on the kernel device-tree, use the `KERNEL_TECH_DTBOS` variable to list the DTBO names.

Note: Qualcomm Linux supports device tree binary overlay only for `custom` variant.

The following example from `qcs9100-ride-sx.conf` shows how DTBOs are used.

```
KERNEL_TECH_DTBOS[sa8775p-addons-ride] = " \
   sa8775p-video.dtbo qcs9100-graphics.dtbo \
   qcs9100-ride-sx-camera.dtbo \
   "
KERNEL_TECH_DTBOS[sa8775p-addons-ride-r3] = " \
   sa8775p-video.dtbo qcs9100-graphics.dtbo \
   qcs9100-ride-sx-camera.dtbo \
   "

KERNEL_TECH_DTBOS[qcs9100-addons-ride] = " \
   sa8775p-video.dtbo qcs9100-graphics.dtbo \
   qcs9100-ride-sx-camera.dtbo \
   "

KERNEL_TECH_DTBOS[qcs9100-addons-ride-r3] = " \
   sa8775p-video.dtbo qcs9100-graphics.dtbo \
   qcs9100-ride-sx-camera.dtbo \
   "

KERNEL_TECH_DTBO_PROVIDERS = "\
```

```
    qcom-graphicsdevicetree \
    qcom-videodtb \
    cameradtb \
    "
```

- **Firmware recipes**

  Qualcomm Linux firmware recipe files are available on GitHub. When the Qualcomm Linux source code is synced, the firmware recipes are available in the following directory: `<workspace>/layers/meta-qcom-hwe/recipes-firmware/firmware`.

  - **Critical boot binaries**

    Critical boot firmware images are required to boot the kernel on the device. The following firmware recipe provides the hardware SoC-specific boot firmware.

`firmware-qcom-bootbins_1.0.bb`	Handles fetch, unpack, and deploy for critical boot firmware binaries for compatible targets. The `QCM6490_bootbinaries.zip`, `QCS9100_bootbinaries.zip`, and `QCS8300_bootbinaries.zip` files provide the boot firmware required for QCS6490, QCS9075, and QCS8300 based machines.

    After generating the build, the firmware binaries in these zip files are available for flashing in the following directory:

    `<workspace>/build-qcom-wayland/tmp-glibc/deploy/images/<machine-name>/<image-name>/`

  - **Subsystem firmware binaries**

    Qualcomm Linux includes firmware binaries that are loaded and run on the corresponding subsystems. As the Qualcomm hardware SoC boots up, individual subsystems run the firmware as they come out of reset.

`firmware-qcom-hlosfw_1.0.bb`	Handles fetch, unpack, and install for subsystem firmware binaries, such as aDSP, cDSP, modem, and WLAN. The `QCM6490_fw.zip`, `QCS9100_fw.zip`, and `QCS8300_fw.zip` files pack the firmware files for QCM6490, QCS9075, and QCS8300 based machines.

    The `firmware-qcom-hlosfw_1.0.bb` recipe does the following:

1. Fetches the subsystem firmware binaries from the remote server based on SRC_URI.
2. Unpacks the zip file.
3. Installs the firmware in `rootfs`.

– **DSP libraries**

User space utilities refer to the DSP libraries, which must be available in the `rootfs` image. The following firmware recipes provide the hardware SoC-specific DSP libraries:

`firmware-qcom-dspso_1.0.bb`	Handles fetch, unpack, and install for DSP libraries. The `QCM6490_dspso.zip`, `QCS9100_dspso.zip`, and `QCS8300_dspso.zip` zip files pack the libraries for QCM6490, QCS9075, and QCS8300 based machines.

The `firmware-qcom-dspso_1.0.bb` recipe does the following:

1. Fetches the DSP libraries from the remote server based on SRC_URI.
2. Unpacks the zip file.
3. Installs the DSP libraries in `rootfs`.

– **Installation of boot, subsystem, and dspso**

When Qualcomm Linux is built, the build system uses the firmware recipes to deploy the prebuilt firmware based on the `MACHINE_EXTRA_RDEPENDS` configuration variable, which is set in the machine configuration file. For example, in `qcom-qcs6490.inc`, see the inclusion of `packagegroup-firmware-qcm6490` in the `MACHINE_EXTRA_RDEPENDS` variable:

```
MACHINE_EXTRA_RDEPENDS += " \
    packagegroup-firmware-qcm6490 \
    "
```

Note: The `packagegroup-firmware-qcm6490` recipe is present in the `<workspace>/layers/meta-qcom-hwe/recipes-firmware/packagegroups/` directory. It groups the firmware recipes to generate the image.

When Qualcomm Linux is built, based on the configuration in the machine configuration and the package group recipe file, the respective firmware recipes from the `<workspace>/layers/meta-qcom-hwe/recipes-firmware/firmware` directory are built.

- **Kernel recipes**

The Qualcomm Linux kernel recipes used by Qualcomm Linux are in `<workspace>/layers/meta-qcom-hwe/recipes-kernel/linux`.

Qualcomm Linux supports the long-term support (LTS) Qualcomm Linux kernel v6.6.x. In the `meta-qcom-hwe` layer, there are two distinct kernel recipes:

- The `linux-qcom-custom_6.6.bb` recipe supports the `custom` BSP and fetches the kernel sources from `qcom.git` hosted at `git.codelinaro.org`.
- The `linux-qcom-base_6.6.bb` recipe supports the `base` BSP and retrieves kernel sources from `linux.git` hosted at `git.kernel.org`.

linux-qcom-custom_6.6.bb

```
inherit kernel sota

COMPATIBLE_MACHINE = "(qcom)"

SRCPROJECT = "git://git.codelinaro.org/clo/la/kernel/qcom.git;
protocol=https"
SRCBRANCH  = "kernel.qclinux.1.0.r1-rel"
SRCREV     = "d3ed32bf7ee64db22653833d4c3d9a80dd76896d"

SRC_URI = "${SRCPROJECT};branch=${SRCBRANCH};destsuffix=kernel \
         ${@bb.utils.contains('DISTRO_FEATURES', 'selinux', '
file://selinux.cfg', '', d)} \
         ${@bb.utils.contains('DISTRO_FEATURES', 'selinux', '
file://selinux_debug.cfg', '', d)} \
         "

S = "${WORKDIR}/kernel"

KERNEL_CONFIG ??= "qcom_defconfig"
```

linux-qcom-base_6.6.bb

```
inherit kernel sota

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 \
```

```
              file://vm-configs/qcom_vm.cfg \
              file://qcom_debug.cfg \
              "
# Apply qcom patches
require ${BPN}-${PV}/configs.inc
require ${BPN}-${PV}/devicetree.inc
require ${BPN}-${PV}/drivers.inc
require ${BPN}-${PV}/dt-bindings.inc
require ${BPN}-${PV}/tools.inc

KERNEL_CONFIG_FRAGMENTS:append = " ${WORKDIR}/qcom.cfg"
KERNEL_CONFIG_FRAGMENTS:append = " ${@oe.utils.vartrue('DEBUG_
BUILD', '${WORKDIR}/qcom_debug.cfg', '', d)}"

S = "${WORKDIR}/git"
```

The machine configuration files use OVERRIDES to select the appropriate kernel recipe variant. The `qcom-base.inc` file chooses `linux-qcom-base` when you select the `base` variant. If you select the `custom` variant, the kernel is built using the `linux-qcom-custom` recipe as shown in the following snippet.

```
PREFERRED_PROVIDER_virtual/kernel ?= "linux-qcom-custom"
PREFERRED_PROVIDER_virtual/kernel:qcom-base-bsp ?= "linux-qcom-
base"
```

Kernel configuration

The Qualcomm Linux kernel recipe uses a different set of kernel configuration and fragments for the `base` and `custom` variants.

Variant	Configuration and fragments files
`base`	`defconfig`, `qcom.cfg`, `qcom_vm.cfg`, `qcom_debug.cfg`
`custom`	`qcom_defconfig`, `qcom_addons.config`, `selinux.cfg`, `qcom_debug.config`, `qcom_addons_debug.config`, `selinux_debug.cfg`

The following table provides the description of `defconfig` and `fragments` used for the `custom` variant:

Kernel configuration fragments	Description
`<kernel_src>/arch/arm64/configs/qcom_defconfig`	Default configuration that's aligned to product/performance needs

Kernel configuration fragments	Description
`<kernel_src>/arch/arm64/configs/qcom_debug.config`	Debug configuration fragment
`<kernel_src>/arch/arm64/configs/qcom_addons.config`	Additional Qualcomm value-added additions on top of upstream aligned base
`<kernel_src>/arch/arm64/configs/qcom_addons_debug.config`	Qualcomm debug enablement

The Qualcomm Linux `custom` variant recipe `linux-qcom-custom_6.6.bb` further supports the `perf` and `debug` variations. The default method to build the kernel using `linux-qcom-custom_6.6.bb` is `perf`.

Build variant	Defconfig/config fragments
`Perf`	– `arch/arm64/configs/qcom_defconfig` – `arch/arm64/configs/qcom_addons.config`
`Debug`	– `arch/arm64/configs/qcom_defconfig` – `arch/arm64/configs/qcom_debug.config` – `arch/arm64/configs/qcom_addons.config` – `arch/arm64/configs/qcom_addons_debug.config`

To build the debug kernel image with `linux-qcom-custom_6.6.bb`, set DEBUG_BUILD to 1 in the shell where you are using BitBake comamnds to build the image.

This selection is effective in the following code:

```
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 autoload kernel modules for Qualcomm platforms, update the KERNEL_MODULE_AUTOLOAD variable in the Qualcomm Linux kernel recipe. For example, the CoreSight and STM modules are autoloaded as follows:

```
KERNEL_MODULE_AUTOLOAD += "coresight coresight-tmc coresight-
funnel"
KERNEL_MODULE_AUTOLOAD += "coresight-replicator coresight-etm4x
coresight-stm"
KERNEL_MODULE_AUTOLOAD += "coresight-cti coresight-tpdm
coresight-tpda coresight-dummy"
KERNEL_MODULE_AUTOLOAD += "coresight-remote-etm coresight-tgu"
KERNEL_MODULE_AUTOLOAD += "stm_core stm_p_ost stm_console stm_
heartbeat stm_ftrace "
```

- **Licenses**

  The licenses for recipes in `meta-qcom-hwe` are listed at `<workspace>/meta-qcom-hwe/files/common-licenses`.

```
common-licenses/
├── BSD-3-Clause-Clear
├── GPLv2.0-with-linux-syscall-note
└── Qualcomm-Technologies-Inc.-Proprietary
```

  Yocto can automatically create SPDX SBOM documents based on image creation. To enable this feature, inherit the `create-spdx` class in `local.conf` as follows:

```
INHERIT += "create-spdx"
```

  After you inherit the class, rebuild the image with the BitBake command:

```
bitbake qcom-multimedia-image
```

  You can find the SPDX output in the following directories:

  – For each recipe, the generated files are available in the `tmp/deploy/spdx/<machine>` directory.

  – The top-level SPDX output file is in the `tmp/deploy/images/MACHINE/<image-recipe>-<MACHINE>.spdx.json` directory.

meta-qcom-distro

This layer provides a reference distribution configuration for Qualcomm Linux. This layer defines the image recipes and package groups.

- **BitBake classes**

  The following table provides an introduction to the BitBake classes, which are available at Classes.

  Qualcomm Linux supports both SSH and UART serial shell for device access. You can choose either SSH or UART to access the device. You can also use ADB to debug issues when IP interfaces are down or to transfer large files.

`image-adbd.bbclass`	The `image-adbd.bbclass` class in `meta-qcom-distro` installs adbd in the image. The adbd daemon remains disabled unless `IMAGE_FEATURES` contains the `enable-adbd` feature. You can disable adbd by manually removing `/etc/usb-debugging-enabled` from `rootfs`.
`image-qcom-deploy.bbclass`	Deploys the image files available in `<workspace>/build-<distro>/tmp-glibc/deploy/images/<machine>/<image-name>`. The generated images are deployed in the `<image-name>` subdirectory.

- **Distribution configuration**

  The following table provides an introduction to the distribution configurations, which are available on GitHub.

`conf/distro/qcom-wayland.conf`	This distribution configuration file defines the `qcom-wayland` distribution. You can use the `qcom-wayland` distribution in the following example command. `MACHINE=qcs6490-rb3gen2-core-kit DISTRO=qcom-wayland QCOM_SELECTED_BSP=base source setup-environment` The `meta-qcom-distro/conf/distro/include/qcom-base.inc` configuration defines common DISTRO_FEATURES. The `meta-qcom-distro/conf/distro/qcom-wayland.conf` configuration adds the following features: – `wayland` – `vulkan` – `opengl` The Yocto Project documentation defines these distribution features at Distribution features.
`conf/distro/include/qcom-base.inc`	The INIT_MANAGER is set to `systemd`. For Yocto Project documentation on INIT_MANAGER, see INIT_MANAGER. Other DISTRO_FEATURES enabled are: `DISTRO_FEATURES:append = " pam overlayfs acl xattr selinux ptest security virtualization tpm usrmerge sota"` To understand the purpose of these DISTRO_FEATURES, see Distribution features. This file selects `systemd` as INIT_MANAGER and `udev` as the DEV_MANAGER.
`conf/distro/include/qcom-security_flags.inc`	This file includes the security flags as defined in security flags.

- **Package groups**

  Package groups are defined in `meta-qcom-hwe` and `meta-qcom-distro`. These package groups help you understand the features defined by the Qualcomm BSP.

`packagegroup-qcom.bb`	Package group that contains all the basic packages.
`packagegroup-qcom-multimedia.bb`	Package group that contains packages to enable multimedia support: – `packagegroup-container` – `packagegroup-qcom-audio` – `packagegroup-qcom-camera` – `packagegroup-qcom-display` – `packagegroup-qcom-fastcv` – `packagegroup-qcom-graphics` – `packagegroup-qcom-k8s` – `packagegroup-qcom-opencv` – `packagegroup-qcom-video` – `python3-docker-compose` – `packagegroup-qcom-iot-base-utils` – `packagegroup-qcom-location` **Note:** The `packagegroup-qcom-location` is defined in `meta-qcom-extras` metadata layer.
`packagegroup-qcom-test-pkgs.bb`	Package group that contains test packages.

- **Image recipes**

  The `meta-qcom-distro` Qualcomm Linux metadata layer defines image recipes, which are available on GitHub. The following table lists various images, their `IMAGE_FEATURES`, and the functions that the images serve:

Image recipe	Description of the image
`qcom-minimal-image.bb`	Defines a small `rootfs` to boot to the shell. The `IMAGE_FEATURES` enabled are as follows: `IMAGE_FEATURES += "splash tools-debug allow-root-login post-install-logging enable-adbd"` For more information about `IMAGE_FEATURES`, see Image features.

Image recipe	Description of the image
`qcom-console-image.bb`	Extends `qcom-minimal-image` by adding more packages and enabling more `IMAGE_FEATURES`: `IMAGE_FEATURES += "package-management ssh-server-openssh"`
`qcom-multimedia-image.bb`	Requires `DISTRO_FEATURE` wayland and it includes all the multimedia packages in `rootfs`.
`qcom-multimedia-test-image.bb`	Includes test packages in `rootfs` to test `qcom-multimedia-image`.
`qcom-multimedia-crossesdk-image.bb`	Generates eSDK for `qcom-multimedia-image`.
`qcom-guestvm-image.bb`	A minimal kernel-based virtual machine (KVM) image with boot to shell support.

- **QDL flashing tool**

  QDL is a flashing tool that communicates with the USB devices to upload flash loader to the device. The flash loader flashes the images to universal flash storage (UFS) or embedded multimedia card (eMMC) built into the device. For more information about `QDL flashing`, see QDL.

Build Qualcomm Linux

The `meta-qcom`, `meta-qcom-hwe`, and `meta-qcom-distro` sections describe the machine configurations, distribution configurations, image recipes, and OVERRIDES.

The supported values for `MACHINE`, `DISTRO`, and `QCOM_SELECTED_BSP` are listed in the table. To setup the environment, use these values and run the following command:

```
MACHINE=<machine configuration name> DISTRO=<Distro name> QCOM_
SELECTED_BSP=<variant name> source setup-environment
```

The following table lists the possible image recipes to select and generate an image according the selected `MACHINE`, `DISTRO`, and `QCOM_SELECTED_BSP`, run the following command:

```
bitbake <image recipe name>
```

QCOM_SELECTED_BSP (selected by you)	Effective BitBake OVERRIDE (derived from QCOM_SELECTED_BSP)	MACHINE configuration (selected by you)	Reference DISTRO configuration (selected by you)	Image recipe (selected by you)
custom	qcom-custom-bsp	• qcs6490-rb3gen2-core-kit.conf • qcs6490-rb3gen2-vision-kit.conf • qcs6490-rb3gen2-industrial-kit.conf • qcs6490-idp.conf • qcs8300-ride-sx.conf • qcs9100-ride-sx.conf • qcs9075-ride-sx.conf • qcs8300-ride-sx.conf	• qcom-wayland • qcom-robotics-ros2-humble	• qcom-minimal-image • qcom-console-image • qcom-multimedia-image • qcom-multimedia-test-image
base	qcom-base-bsp			

To learn more about the supported combinations for building Qualcomm Linux, see Qualcomm Linux Release Notes.

For detailed build instructions, see GitHub workflow for unregistered users.

meta-qcom-realtime

The `meta-qcom-realtime` metadata layer is available on GitHub. This layer provides additional software support for building a real-time kernel for Qualcomm devices.

- **Kernel recipes**

  Qualcomm Linux supports the LTS Qualcomm Linux kernel v6.6.x and real-time extensions. It's maintained through the `linux-qcom-custom-rt_6.6.bb` and `linux-qcom-base-rt_6.6.bb` Yocto recipes at `recipes-kernel/linux` under the `meta-qcom-realtime` layer. The pending pre-empt RT patches can be found at realtime. These patches are fetched and applied on top of the `linux-qcom-custom-rt_6.6.bb`, which is publicly hosted at Codelinaro.

  To compile a real-time kernel for Qualcomm devices:

  – If you chose the `custom` OVERRIDE, `conf/layer.conf` selects `linux-qcom-custom-rt`.

  – If you chose the `base` OVERRIDE, `conf/layer.conf` selects `linux-qcom-base-rt`.

  Kernel configuration

  Both recipes append the `qcom_rt.cfg` fragment as follows:

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

- **Enable** `meta-qcom-realtime` **in build**

  To include `meta-qcom-realtime` in the build, export the `meta-qcom-realtime` layer to EXTRALAYERS in `bblayers.conf`, as described in the following steps:

  1. Source the environment.

     The following is an example to source the environment for a QCS6490-based machine and the `qcom-wayland` distribution:

     ```
     MACHINE=qcs6490-rb3gen2-core-kit DISTRO=qcom-wayland source
     setup-environment
     ```

  2. Open the `build-qcom-wayland/conf/bblayers.conf` file and update the EXTRALAYERS variable as follows.

     ```
     EXTRALAYERS ?= " \
       ${WORKSPACE}/layers/meta-qcom-realtime \
      "
     ```

  3. Run the build command to rebuild with `meta-qcom-realtime` as follows:

```
bitbake qcom-multimedia-image
```

meta-qcom-extras

This layer is an optional metadata layer for registered users. This layer allows source compilation of select components, which are otherwise present as binaries in `meta-qcom-hwe`. If you are entitled to receive this metadata layer, you can use the steps shared in the Qualcomm Linux Build Guide.

- **Firmware recipes**

  The `meta-qcom-extras` layer provides recipe append files for the firmware recipes defined in `meta-qcom-hwe` layer at `meta-qcom-hwe/recipes-firmware/firmware`. The following code snippet shows these recipe append files, with extension `.bbapend`. The firmware recipe append files have the `SRC_URI` set to zip files that you may want to use instead of the default zip files distributed with this Qualcomm Linux release.

  Follow the Qualcomm Linux Build Guide to build cDSP, aDSP, Boot firmware, and generate the zip file that you can integrate using the provided recipe append files.

`recipes-firmware`
`└── firmware`
`    ├── firmware-qcom-bootbins_1.0.bbappend`
`    ├── firmware-qcom-dspso_1.0.bbappend`
`    ├── firmware-qcom-hlosfw_1.0.bbappend`
`    └── firmware-qcom-partconf_1.0.bbappend`
	The recipes in `meta-qcom-extras` override the `SRC_URI` in recipes from `meta-qcom-hwe`. The `meta-qcom-hwe` layer uses pre-built firmware binaries by default, while the `meta-qcom-extras` layer builds the firmware zip provided by the user.
	The `meta-qcom-extras` layer ignores the pre-built binaries from the default .zip files. These recipes instead search for user-provided .zip files set in FWZIP_PATH as follows:
	– `firmware-qcom-bootbins_1.0.bbappend`: Searches for the zip archive under the path defined by the variable `FWZIP_PATH`.
	– `firmware-qcom-hlosfw_1.0.bbappend`: Searches for the zip archive under the path defined by the variable `FWZIP_PATH`.
	– `firmware-qcom-dspso_1.0.bbappend`: Searches for the zip archive under the path defined by the variable `FWZIP_PATH`.
	– `firmware-qcom-partconf_1.0.bbappend`: Searches for the zip archive under the path defined by the variable `FWZIP_PATH`.

meta-qcom-qim-product-sdk

- **BitBake classes**

  The following table lists the BitBake classes defined in the `meta-qcom-qim-product-sdk` metadata layer:

BitBake class	Description
qim-prod-sdk-pkg.bbclass	– Provides a packaging task to pack the Qualcomm Intelligent Multimedia Product (QIMP) SDK artifacts into an archive. It's invoked by the `qim-product-sdk` recipe. – The easy-to-install artifact archives are available in the `<workspace>/build-qcom-wayland/tmp-glibc/deploy/qim_prod_sdk_artifacts` directory after the recipe build is complete.
qimsdk-pkg.bbclass	– Provides a task to package the Qualcomm Multimedia SDK packages into an archive along with an easy-to-use install script. The archives are generated with packages to develop, deploy, and debug separately. – The easy-to-install artifact archives are available in the `<workspace>/build-qcom-wayland/tmp-glibc/deploy/qimsdk_artifacts` directory. – Invoked by the `qim-sdk` recipe during the build.
tflitesdk-pkg.bbclass	– Provides a packaging task to pack the Lite Runtime SDK artifacts into several archives to develop, deploy, and debug. It's invoked by the `tflite-sdk` recipe during the build. – The easy-to-install artifact archives are available in the `<workspace>/build-qcom-wayland/tmp-glibc/deploy/qim_prod_sdk_artifacts` directory after the recipe build is complete.

- **Distribution configuration**

`layer.conf`	Configures the project layers with the following information: – Recipe file path information – Supported Yocto version – Supported Qualcomm® Hexagon™ Processor version – Supported Qualcomm® Neural Processing SDK version – Supported Qualcomm Neural Network (QNN) SDK version

- **Image recipes**

Recipe	Description
`recipes-gst`	Consists of the upstream GStreamer recipe changes (`.bbapend`) along with Qualcomm recipes: – `gstreamer1.0-plugins-good_1.20%.bbappend` – `gstreamer1.0-plugins-qcom-oss-base.bb` – `gstreamer1.0-plugins-qcom-oss-batch.bb` – `gstreamer1.0-plugins-qcom-oss-metamux.bb` – `gstreamer1.0-plugins-qcom-oss-mldemux.bb` – `gstreamer1.0-plugins-qcom-oss-mlmeta.bb` – `gstreamer1.0-plugins-qcom-oss-mlmuxer.bb` – `gstreamer1.0-plugins-qcom-oss-mlqnn.bb` – `gstreamer1.0-plugins-qcom-oss-mlsnpe.bb` – `gstreamer1.0-plugins-qcom-oss-mltflite.bb` – `gstreamer1.0-plugins-qcom-oss-mlvclassification.bb` – `gstreamer1.0-plugins-qcom-oss-mlvconverter.bb` – `gstreamer1.0-plugins-qcom-oss-mlvdetection.bb` – `gstreamer1.0-plugins-qcom-oss-mlvpose.bb` – `gstreamer1.0-plugins-qcom-oss-mlvsegmentation.bb` – `gstreamer1.0-plugins-qcom-oss-overlay.bb` – `gstreamer1.0-plugins-qcom-oss-qmmfsrc.bb` – `gstreamer1.0-plugins-qcom-oss-socket.bb` – `gstreamer1.0-plugins-qcom-oss-tools.bb` – `gstreamer1.0-plugins-qcom-oss-vcomposer.bb` – `gstreamer1.0-plugins-qcom-oss-vsplit.bb` – `gstreamer1.0-plugins-qcom-oss-vtransform.bb` – `gstreamer1.0-qcom-oss-sample-apps.bb`

Recipe	Description
`recipes-qcom-ml`	Consists of two recipes: – `qnn.bb`: Used to package QNN SDK – `snpe.bb`: Used to package Qualcomm Neural Processing SDK
`recipes-qim-product-sdk`	Recipe to install QIM product SDK that has QIM, Qualcomm Neural Processing, QNN, and Lite Runtime SDKs
`recipes-tensorflow-lite`	Lite Runtime recipes build and install Lite Runtime for the following versions: – 2.12.1, 2.13.1, 2.14.1, and 2.15.0 – Default version: 2.15.0

- **Package groups**

Package group	Description
packagegroup-qcom-gstrecipes-gst	Package group to enable upstream basic GStreamer along with Qualcomm plug-ins: – gstreamer1.0-plugins-qcom-oss-base – gstreamer1.0-plugins-qcom-oss-tools – gstreamer1.0-plugins-qcom-oss-batch – gstreamer1.0-plugins-qcom-oss-metamux – gstreamer1.0-plugins-qcom-oss-mldemux – gstreamer1.0-plugins-qcom-oss-mlmeta – gstreamer1.0-plugins-qcom-oss-mlvconverter – gstreamer1.0-plugins-qcom-oss-mlvclassification – gstreamer1.0-plugins-qcom-oss-mlvdetection – gstreamer1.0-plugins-qcom-oss-mlvpose – gstreamer1.0-plugins-qcom-oss-mlvsegmentation – gstreamer1.0-plugins-qcom-oss-overlay – gstreamer1.0-plugins-qcom-oss-qmmfsrc – gstreamer1.0-plugins-qcom-oss-socket – gstreamer1.0-plugins-qcom-oss-vcomposer – gstreamer1.0-plugins-qcom-oss-vsplit – gstreamer1.0-plugins-qcom-oss-vtransform – gstreamer1.0-qcom-oss-sample-apps – gstreamer1.0-plugins-qcom-oss-mlsnpe – gstreamer1.0-plugins-qcom-oss-mlqnn – gstreamer1.0-plugins-qcom-oss-mltflite It also packs the upstream GStreamer packages: – cairo – gdk-pixbuf – liba52 – libdaemon – libgudev – lame – libpsl – librsvg – libsoup-2.4 – libtheora – libwebp – mpg123 – orc – sbc – speex – taglib

Package group	Description
packagegroup-qcom-gst-basic	Package group to enable upstream GStreamer: – gstreamer1.0 – gstreamer1.0-plugins-base – gstreamer1.0-plugins-good – gstreamer1.0-plugins-bad – gstreamer1.0-rtsp-server
packagegroup-qcom-qim-product	Package group to pack the following packages along with an install script: – packagegroup-qcom-gst – packagegroup-qcom-ml – install.sh
packagegroup-qcom-ml	Package group to pack the Qualcomm ML framework: – tensorflow-lite – qnn – snpe – libgomp-dev

2.2 Qualcomm Linux software components

This section covers key software components like initialization scripts, debugging tools, systemd-boot, partitioning tools, and support for containers and Kubernetes. It also describes features, such as logging and secondary virtual machines.

System initscripts	Description

System initscripts

meta-qcom-hwe added the system initscripts to the image as follows:

System initscripts	Description
var-persist.mount	Mounts the /dev/disk/by-partlabel/persist disk partition to /var/persist.
android-tools-adbd.service	Provides the adbd daemon on the device.
logrotate.service	Archives old logs. Modify the rsyslog.logrotate configuration file in Qualcomm Linux to manage on-device logs. The modified rsyslog.logrotate file is in the meta-qcom-hwe/dynamic-layers/openembedded-layer/recipes-devtools/rsyslog/rsyslog/rsyslog.logrotate directory. This file overrides the default configuration file provided by meta-openembedded/meta-oe/recipes-extended/rsyslog/rsyslog/rsyslog.logrotate.

pd-mapper.service	Configures and manages protection domains. pd-mapper_git.bbappend in the Qualcomm Linux BSP layer updates the pd-mapper.service.in file to run the service as a system user instead of as the root user.
	`do_install:prepend() {`
	`    # convert the service from root user to system user`
	`    sed -i "/ExecStart=/i\User=system\nGroup=system" pd-mapper.service.in`
	`}`

property-vault.service	Provides the property_get and property_set functionalities. For more information about this service, see Properties.
persist-property-vault.service	Runs set-persist-prop.sh, which sets the le.persistprop.enable flag to true. This property allows use of persist properties, which are stored in the filesystem and persist across reboots.

System initscripts	Description
`resize-partition@.service`	Resizes the file system at bootup time according to the size of the partition. `ExecStart=/bin/sh -c "/sbin/e2fsck -n /dev/disk/by-partlabel/%i; if [ $? -gt 1 ]; then /sbin/mkfs.ext4 /dev/disk/by-partlabel/%i; fi; /sbin/resize2fs /dev/disk/by-partlabel/%i"`
`rsyslog.service`	Redirects logs according to a specified configuration.
`sys-kernel-debug.mount`	Mask the `sys-kernel-debug.mount` unit when building the `perf` variant. This conditional masking of this systemd unit is done in `do_install:append:qcom` task of `meta-qcom-hwe/recipes-core/systemd/systemd_%.bbappend`.

Debug tools

The `packagegroup-core-tools-debug` defined in the `<workspace>/layers/poky/meta` directory adds debug tools as part of `rootfs`. The `<workspace>/layers/meta-qcom-hwe/recipes-devtools` contains the appended package group recipe as `packagegroup-core-tools-debug.bbappend`. This append file adds `ltrace`, `perf`, `sysstat`, and `valgrind` tools to this package group. For more information, see Debug Linux user space issues.

Configure and secure boot with systemd-boot and UKI

The systemd-boot unified extensible firmware interface (UEFI) boot manager provides options to control the boot flow and loads the user-selected boot loader. The configuration files, kernel images, initrd images, and other EFI images must reside on the EFI partition.

To run the Qualcomm Linux kernel directly as EFI images, build them with `CONFIG_EFI_STUB`. The systemd-boot supports two configurations:

- Type1:

  The Type1 configuration uses boot loader specification (BLS) description files. You can find these files in the `/loader/entries/` directory on the EFI.

- Type2:

  The Type2 configuration uses unified kernel images (UKI). These images combine the kernel, initrd, and kernel command-line into a single EFI executable. Type2 offers better security because the UKI contains all the necessary information for the device to boot. Signing a UKI image secures all included entities. If UEFI secure boot is enabled, the system only loads signed images, making signing a requirement.

For more details, see systemd-boot.

Note: To use a secure boot enabled device, signing is required.

- **UKI**

  UKI is a combination of a UEFI boot stub program, a Qualcomm Linux kernel image, an initrd, and other resources in a single UEFI portable executable (PE) file. The UEFI boot stub looks for various resources for the kernel invocation inside the UEFI PE binary. This allows combining various resources inside a single UKI image, which may then be signed using sbsign. Qualcomm Linux uses sbsign to sign PE files, while non-PE files like DTB are signed using OpenSSL.

  For more details about UKI, see unified_kernel_image. The following table shows the `uki.efi` content:

Components of uki.efi file	**Contents**
Initrd = Init ramdisk	`initramfs-ostree-image-qcs6490-rb3gen2-vision-kit.cpio.gz`
Linux = Kernel Image	`Image` (as systemd-boot expects uncompressed kernel)
Uname = Kernel Release	`6.6.52`
Efi-arch = Architecture	`aa64`
Stub = System-boot efi stub	`linuxx64.efi.stub`
OS-release = OS-release	– `ID = qcom-wayland` – `Name = "QCOM Reference Distro with Wayland"` – `VERSION = "1.0"` – `VERSION_ID = 1.0` – `PRETTY_NAME = "QCOM Reference Distro with Wayland 1.0"`

Image recipes

`meta-qcom-hwe/recipes-kernel/images` contains the following recipes:

– `linux-qcom-uki.bb` generates `uki.efi`.

– `esp-qcom-image.bb` generates a VFAT image, `efi.bin`, which contains `uki.efi` and `systemd-boot`.

The `meta-qcom-distro/classes/image-qcom-deploy.bbclass` invokes the `esp-qcom-image`.

- **EFI image**

  The EFI image, `efi.bin`, is a VFAT file system image stored in the EFI partition of the flash. This VFAT file system contains the images necessary for the UEFI to load and transfer execution control to systemd-boot. To transfer execution control to the systemd-boot manager, UEFI mounts `efi.bin`, loads `bootaa64.efi`, and executes it. The systemd-boot manager parses the `loader.conf` and loads the kernel image, and transfers the control to it.

  For more information about the structure of EFI, see EFI system partition.

  Following is the sample structure of `efi.bin` from Qualcomm Linux. It contains systemd-boot `bootaa64.efi` and Qualcomm Linux kernel `vmlinuz-<version>` under `/ostree/poky-<sha256-sum>` directory.

```
├── EFI
│   ├── BOOT
│   │   └── bootaa64.efi
├── loader
│   └── loader.conf
├── ostree
│   ├── poky-e7a2237ecd4319dae26f498925c89d892b11e9fb0f526adce78775502550bbf8
│   │   └── initramfs-6.6.38.img
│   │   └── vmlinuz-6.6.38
```

  efi.bin file generated with OSTree support

- **Signing**

  Secure boot is a feature in the UEFI standard, but it's not enabled by default in Qualcomm Linux. When enabled, secure boot adds a layer of protection to the preboot process by maintaining a cryptographically signed list of binaries that are run at device boot-up if successfully authenticated. This ensures that the device's boot firmware and Linux OS boot components, such as the boot manager, kernel, and initramfs, haven't been tampered with.

  UEFI secure boot uses a digital signature to validate the authenticity and integrity of the binary code that it loads. The UEFI secure variables store all the keys. Achieving UEFI secure boot involves using the platform key (PK), key exchange key (KEK), database (DB), and forbidden signatures database (DBX).

Using secure boot requires the keys PK, KEK, and DB. While multiple KEK, DB, and DBX are allowed, only one PK is allowed.

Enabling UEFI secure boot requires registering the PK in the system. It's advised to provision PK at the last step of the secure boot enabling process. For more information about how Qualcomm has implemented the UEFI secure boot feature, see Secure boot.

Host tool signing_tool.py to sign Linux OS images generated by Qualcomm Linux builds

Enabling UEFI secure boot requires signing the EFI and DTB images. Use the `signing_tool.py` host signing tool, to streamline this process. This command-line Python script runs on a Linux host computer (Ubuntu 20.04 or later versions). It automates the signing of EFI and DTB images in two separate operations. The tool also supports the feature to combine the DTB files.

The host signing tool is available for download on GitHub.

Host signing tool overview

The host signing tool runs on a Linux machine with Python3 installed. It can sign either the EFI image or the DTB image in a single operation. To sign both the EFI and DTB images, you must invoke the tool twice with different inputs.

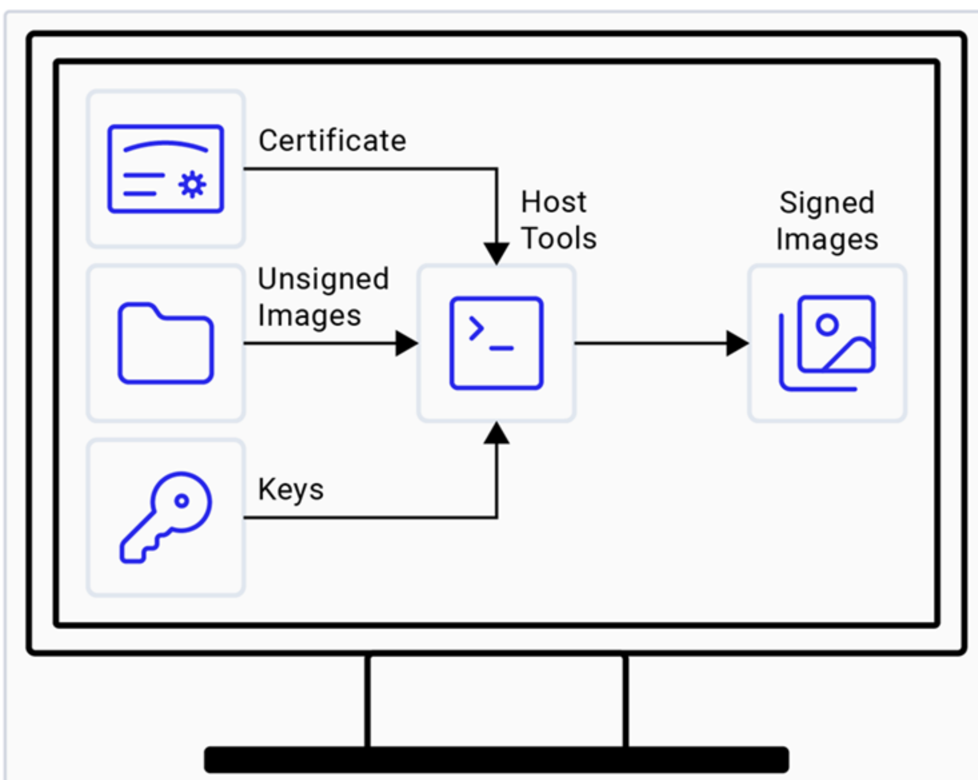


Figure : Linux machine with OpenSSL and sbsign

The host tool expects unsigned EFI or DTB files, along with certificates and keys, as input. After invoking, the tool unpacks the unsigned image, signs the available items using the provided key and certificate, and then repacks the images, replacing the unsigned version with the signed one.

To combine DTB files, you must follow a different process than the signing process. Use the tool to either combine a new DTB file with a pre-existing `dtb.bin` or create a new concatenated `dtb.bin` file from a list of available DTB files.

Working of host signing tool

– **Prerequisites to run the tool**

  To run this tool, install the following on the Linux host computer:

  - OpenSSL, sbsign, and mtools utilities
  - Python3
  - pip, subprocess, shlex, socket, glob, and shutil Python modules

– **Host signing tool configuration**

  You must configure the host signing tool before starting the operation.

  - `config.ini` **file**

    The host tool requires providing the necessary information in a `config.ini` configuration file. The tool reads this file and signs the image accordingly. The following code snippet shows the variables in the configuration file:

```
[common]
# Section - 1: Common Selection
# Select operation: 1. sign_image or 2. combine_dtb
operation = sign_image
# This option is useful for both operations(sign_image &
combine_dtb). Possible values for file_path are 1. remote or
2. local
file_path = local
# This option is required for both operations(sign_image &
combine_dtb) if file_path == remote
local_machine_private_key_path = /usr2/<user_name_for_
machine>/.ssh/id_rsa

# Section - 2: operation == sign_image related common
selection
# Possible values for image_type are 1. efi or 2. dtb
image_type = efi
# This option is required if operation == sign_image & image_
type == efi
```

```
loader_conf_timeout = 20

# Section - 3: operation == combine_dtb related common
selection
# Possible values for combine_dtb_type are 1. combine_with_
old_dtb, 2. combine_without_old_dtb
combine_dtb_type = combine_with_old_dtb

# Below options are required to fetch file from remote linux
machine in the same network(i.e. if file_path == remote)

# This option is useful if operation == sign_image & image_
type == efi
[efi_config]
efi_remote_hostname = <remotemachine_ip_or_hostname_where_
efi.bin_available>
efi_remote_username = <username_on_remote_machine_where_efi.
bin_available>
efi_remote_filepath = <full_path_of_efi.bin_file_on_
remotemachine>

# This option is useful if operation == sign_image. Both
image_type requires this option
[keys_config]
keys_remote_hostname = <remotemachine_ip_or_hostname_where_
keys_available>
keys_remote_username = <username_on_remote_machine_where_
keys_available>
keys_remote_filepath = <full_path_of_keys_directory_on_
remotemachine>

# This option is useful if operation == sign_image & image_
type == dtb
[dtb_config]
dtb_remote_hostname = <remotemachine_ip_or_hostname_where_
dtb_available>
dtb_remote_username = <username_on_remote_machine_where_dtb_
available>
dtb_remote_filepath = <full_path_of_dtb_on_remotemachine>

# This option is useful if operation == combine_dtb.
[combine_dtb_config]
combine_dtb_remote_hostname = <remotemachine_ip_or_hostname_
where_combined-dtb.dtb_available>
```

```
combine_dtb_remote_username = <username_on_remote_machine_
where_combined-dtb.dtb_available>
combine_dtb_remote_filepath = <full_path_of_combined-dtb.dtb_
on_remotemachine>
```

Table : Variables in config.ini file

Variable in config.ini	Values	Description
`operation`	`sign_image/combine_dtb`	Use this configuration to select either signing the image or combining DTB files.
`image_type`	`efi/dtb`	If `operation == sign_image`, use this configuration to select `efi` or `dtb` to sign separately.
`combine_dtb_type`	`combine_with_old_dtb/combine_without_old_dtb`	If `operation == combine_dtb`, use this configuration to select the type of combine DTB operation you want to perform. · `combine_with_old_dtb`: combine with DTB from old `dtb.bin` · `combine_without_old_dtb`: combine a set of DTB files
`file_path`	`local/remote`	· local: Keys and efi.bin/dtb.bin are present in the same path as the script. · remote: Copy efi.bin/dtb.bin and the keys from a remote Linux machine to the current path.
`local_machine_private_key_path`	`<path of id_rsa file in local machine>`	This file establishes an SSH connection with a remote machine if `file_path = remote`.
`loader_conf_timeout`	`<timeout in seconds>`	The systemd-boot wait time to let you choose to authenticate the binaries. This option is required to sign `efi.bin`.
`efi/keys/dtb/combine-dtb_remote_hostname`	`<ip or hostname of the remote Linux machine>`	If `file_path = remote`, then the host tool selects the host name of the remote machine to copy the `efi/keys/dtb/combine-dtb` file from the remote machine using SCP.

Variable in config.ini	Values	Description
efi/keys/dtb/combine-dtb_remote_username	<username_on_remote_machine>	If file_path = remote, then the host tool selects the user name of the remote machine to copy the efi/keys/dtb/combine-dtb file from the remote machine using SCP, provided the username is created on the remote machine.
efi/keys/dtb/combine-dtb_remote_filepath	<full_path_of_file_on_remote_machine>	If file_path = remote, then the host tool selects the path of a efi/key/dtb/combine-dtb file on the remote machine to copy that file from the remote machine using SCP.

- **Configure using config.ini file**

  1. Operation selection: Set the operation variable to specify which operation must be performed. The options are either sign_image or combine_dtb.
  2. Image selection: If you select operation == sign_image, specify which image to sign by setting the image_type variable. The options are either efi or dtb.
  3. File location: Indicate the location of the unsigned EFI/DTB image, keys, and certificates using the file_path variable.

     If you select local in the configuration file, copy the EFI/DTB image, keys, and certificate files manually to the local working directory.

     a. Create an unsigned_binaries directory in the same path as the script, and then copy the efi.bin/ dtb.bin image into that directory.

     b. Create a keys directory in the same path as the script and then copy the db.auth, db.crt, db.key, KEK.auth, and PK.auth files into that directory.

     If you want the script to copy the required files automatically from a remote Linux machine on the same network, select remote in the configuration file.

     In the configuration file, provide information for the following variables:

     - local_machine_private_key_path (mandatory)
     - [efi_config] section (if operation is sign_image and if image_type is efi)
     - [keys_config] section (if operation is sign_image)
     - [dtb_config] section (if operation is sign_image and if image_type is dtb)
     - [combine_dtb_config] section (if operation is combine_dtb)

Note: The script supports copying from another Linux machine over SCP within the same network.

4. Loader configuration timeout: When `image_type` is set to efi in the configuration file, update the `loader_conf_timeout` variable.
5. Combining DTB selection: When you set `operation == combine_dtb`, specify the type of DTB combination operation by setting the `combine_dtb_type` variable. The options are either `combine_with_old_dtb` or `combine_without_old_dtb`.
   a. If you select `combine_dtb_type == combine_with_old_dtb`, create an `unsigned_binaries` directory in the same path as the script and copy the `dtb.bin` image to that directory.
   b. For both options, create a `dtb_files` directory in the same path as the script and copy all the DTB files that must be combined into that directory (either with the old combined DTB from `dtb.bin` or with each other only).
6. Handling missing configuration: If you missed any configuration information, the script runs and prompts you for the missing details through the command line.

- **Run host signing tool**

1. Run the host tool: After completing the code build process and obtaining the unsigned `efi.bin` and `dtb.bin` images, run the host signing tool.
2. Prepare a host computer: Store the host signing tool files (`signing_tool.py` and `config.ini`) on a Linux machine. Ensure that both the files are in the same working directory.
3. Configure the tool: Set up the host signing tool according to the configuration instructions.
4. Run the tool: Run the following command to launch the host tool from the command line: `$python3 signing_tool.py`
5. Interactive process: The host signing tool displays your selections and operational commands on the screen. It also displays errors in the command line.
6. Signed images: After the tool completes its process, it creates a directory called `signed_binaries` in the same working directory. The signed `efi.bin` or `dtb.bin` image is stored in the directory. The tool deletes other user-created directories after signing.
7. Repeat for both images: Follow this process twice, once for `efi.bin` and once for `dtb.bin`. After each signing operation, delete the `signed_binaries` directory before starting a new operation.

– **Host signing tool workflow**

The following figure shows the workflow of the host signing tool:

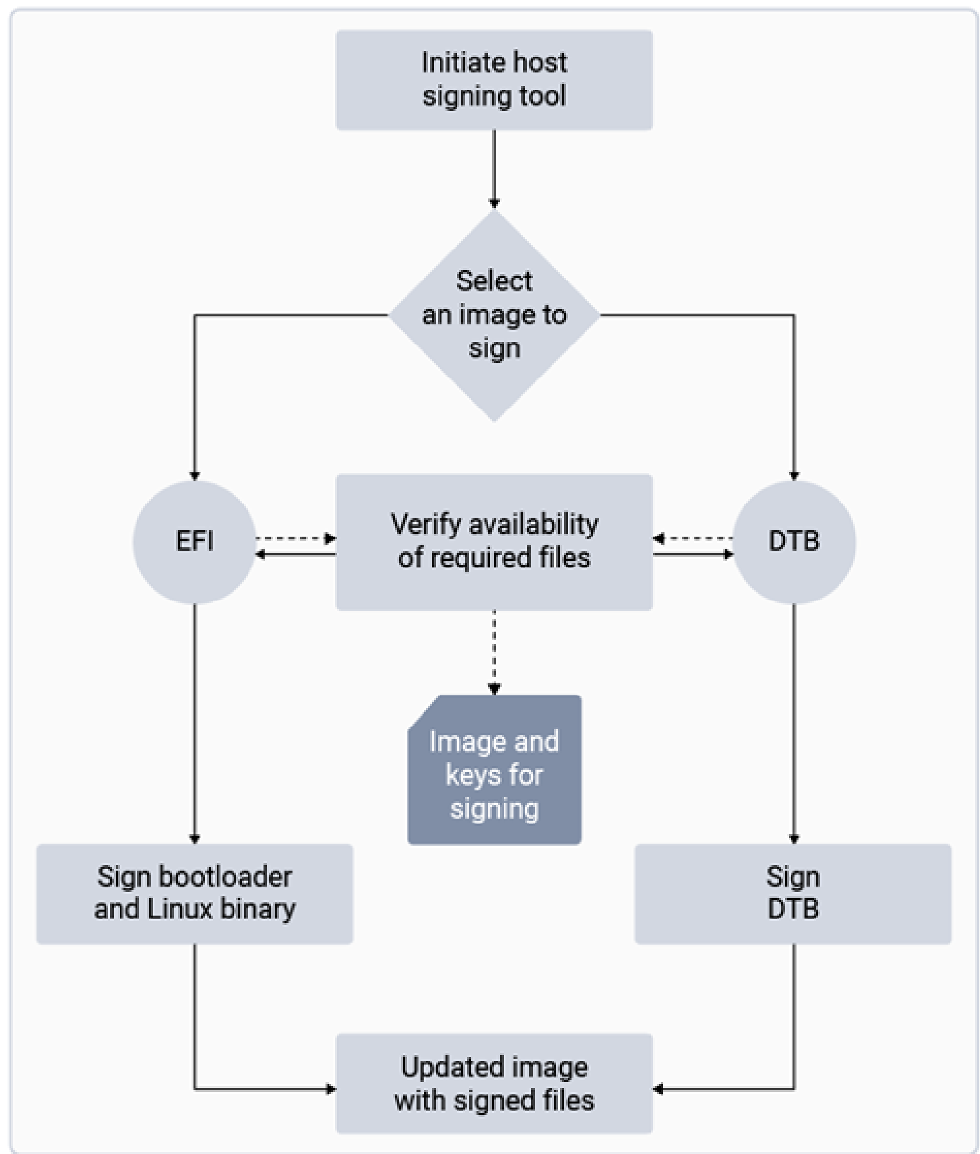


Figure : Host signing tool workflow

- The host tool requires the `efi.bin` and `dtb.bin` paths (absolute path or network path).
  - `efi.bin` with OSTree support contains `vmlinuz-x.y.z` (Qualcomm Linux kernel image) and `bootaa64.efi` (boot loader image).
  - `dtb.bin` contains `combined-dtb.dtb`.

- The host tool requires the path of `certificate` and `key` (absolute path or network path) to sign the images.
- The host tool mounts `efi.bin`/`dtb.bin` on the FAT partition, which provides the following directory structure and follows its separate signing process:

```
├─── EFI
|     ├─── BOOT
|     |     └─── bootaa64.efi
|     └─── Linux
|           └─── uki.efi
└── loader
        └── loader.conf
```

efi.bin

```
├── EFI
|  ├── BOOT
|  |  └── bootaa64.efi
├── loader
|  └── loader.conf
├── ostree
|  ├── poky-e7a2237ecd4319dae26f498925c89d892b11e9fb0f526adce78775502550bbf8
|  |  └── initramfs-6.6.38.img
|  |  └── vmlinuz-6.6.38
```

efi.bin with OSTree support

```
.
└── combined-dtb.dtb
```

dtb.bin

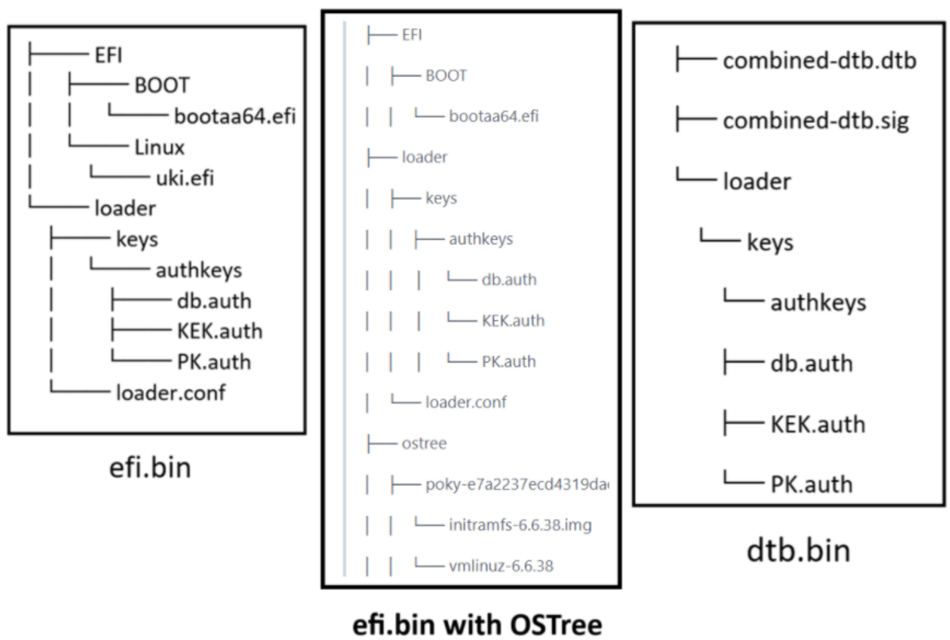
- After signing the images, the host tool copies the *Auth* files to the `/loader/keys/authkeys` directory for both `efi.bin` and `dtb.bin`.
- The host tool must configure the wait time in the `systemd-boot` loader

configuration. This wait time stops the Kernel loading and allows you to review and select the `systemd-boot` menu options. The `loader.conf` file must be available in an updated `efi.bin` file.

Note: The signing process isn't followed for the `dtb.bin` file.

- The host tool configures `/loader/loader.conf`.
- The syntax for `loader.conf` is `timeout x`, where x = timeout in seconds.

- After the image is signed, the host tool must unmount the `efi.bin/dtb.bin` from the FAT partition. Store the signed `efi.bin/dtb.bin` on the host computer on the similar path as the host tool in the `signed_binaries` directory.
- Following is the directory structure for signed `efi.bin` and `dtb.bin`:



Directory structures of efi.bin and dtb.bin files

`efi.bin` **signing process**

- The host tool uses the `sbsign` utility to sign the `uki.efi` or vmlinuz.x.y.z and `bootaa64.efi` images separately.
- `sbsign` requires `certificate` and `key` for the signing process. Verify the following syntax where `dsk1.key` is key, `dsk1.crt` is certificate, and the output filename is the same as the input file:

```
sbsign --key <key file> --cert <cert file> <efi file>
<output file name>
```

Examples:

- ```
  sbsign -key dsk1.key -cert dsk1.crt bootaa64.efi
  bootaa64.efi
  ```

- ```
  sbsign --key dsk1.key --cert dsk1.crt uki.efi uki.
  efi
  ```

- ```
  sbsign --key dsk1.key --cert dsk1.crt vmlinuz.x.y.z
  vmlinuz.x.y.z
  ```

`dtb.bin` **signing process**

- The host tool requires the path of the `dtb.bin` file.
- The host tool requires the path of `key` and `certificate` (absolute path or network path) to sign the images.
- UEFI secure boot requires PE format files for verification. Non-PE files, such as `dtb`, can't be signed using `sbsign` as this signing tool requires PE format files as input.
- The host tool uses the `openssl` utility to sign the `dtb` file. Verify the following syntax, where `dsk1.key` is key and `dsk1.crt` is certificate:

```
   openssl cms -sign -inkey <.key file> -signer <.
crt file> -binary -in <dtb file> --out <output .
dtb.sig file> -outform DER
```

Example:

```
   openssl cms -sign -inkey dsk1.key -signer dsk1.crt
-binary -in <foo.dtb file> --out <foo.dtb.sig file >
-outform DER
```

This command adds the signature for the DTB file in a separate file (`foo.dtb.sig`) and doesn't modify the original file (`foo.dtb`). Hence, the host tool must keep both the files where the `*.dtb.sig` file is used during the UEFI secure boot verification.

– **Host signing tool workflow to combine DTB**

To combine DTB files using the host tool, do the following:
```

1. Prepare DTB files: Place the new DTB files in the `dtb_files` directory for combination.
2. Select operation: Set `operation=combine_dtb` in `config.ini`.
3. Combine with the old DTB:
   a. To combine the new DTB files with the existing `combined-dtb.dtb` in `dtb.bin`, set `combine_dtb_type=combine_with_old_dtb`.
   b. Ensure that the old `dtb.bin` is in the `unsigned_binaries/` directory.
   c. The host tool takes `combined-dtb.dtb` from the old `dtb.bin` and appends all DTB files from the `dtb_files` directory.
   d. The host tool then creates a `dtb.bin` (vfat), copies all old files/directories from the old `dtb.bin`, and includes the newly created `combined-dtb.dtb` with the appended DTB files.
   e. The host tool places the updated `dtb.bin` in the `unsigned_combined_dtb_bi` directory.
4. Combine without old DTB:
   a. To combine only the new DTB files in the `dtb_files` directory, set `combine_dtb_type=combine_without_old_dtb`.
   b. The host tool takes all DTB files from the `dtb_files` directory and creates a `combined-dtb.dtb` file.
   c. The host tool then creates a `dtb.bin` (vfat) and includes the newly created `combined-dtb.dtb`.
   d. The host tool places the updated `dtb.bin` in the `unsigned_combined_dtb_bin` directory.

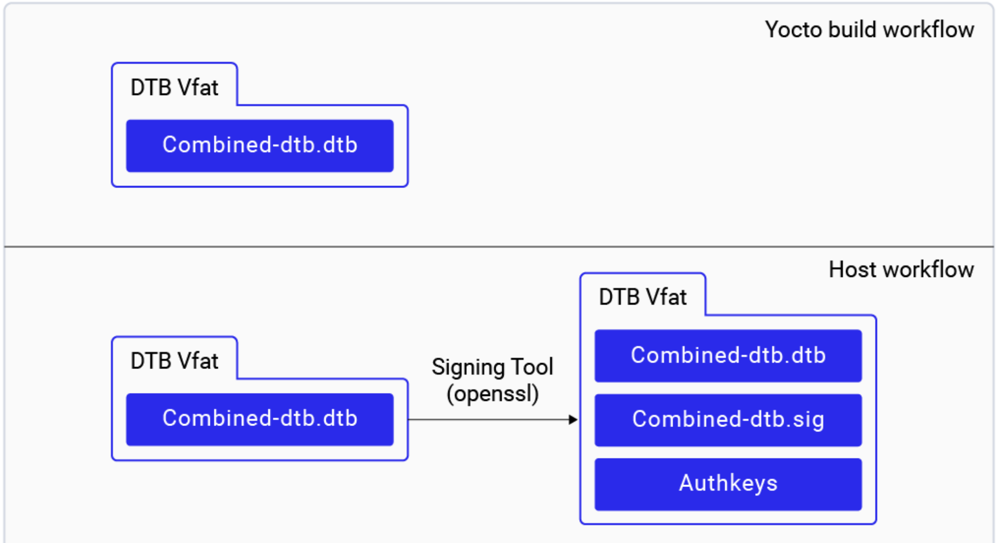


Figure : Create/sign DTB

- **Multi-DTB support**

  Qualcomm supports multiple Qualcomm development kits based on the same hardware SoC. For example, the QCS6490 development kit variants include the RB3 Gen 2 Core development kit and RB3 Gen 2 Vision development kit.

  Each Qualcomm development kit variant has its own DTB in the kernel. During bootup, UEFI selects the appropriate DTB based on the specific Qualcomm development kit variant. To facilitate this, combine and store all DTBs for Qualcomm development kits sharing the same hardware SoC in the DTB partition.

  Generate combined DTB

  To enable multi-DTB support, append all supported DTBs one after the other to generate a combined DTB.

DTB1	**DTB2**	**DTB3**	**DTB4**	**......**	**......**

  For example, to generate the `combined-dtb.dtb` for the RB3 Gen 2 Vision development kit, combine the following DTBs. The following code snippet is from the `meta-qcom-hwe/conf/machine/qcs6490-rb3gen2-vision-kit.conf` file:

```
KERNEL_DEVICETREE:pn-linux-qcom-custom = " \
                    qcom/qcs6490-addons-rb3gen2-video-mezz.dtb \
                    qcom/qcs6490-addons-rb3gen2-vision-mezz.dtb \
                    qcom/qcs6490-addons-rb3gen2-vision-mezz-hsp.dtb
\
                    qcom/qcs6490-addons-rb3gen2-ptz-mezz.dtb \
```

```
                    qcom/qcs6490-addons-rb3gen2-ia-mezz.dtb \
                    qcom/qcs5430-fp1-addons-rb3gen2-vision-mezz.dtb
\
                    qcom/qcs5430-fp1-addons-rb3gen2-vision-mezz-hsp.
dtb \
                    qcom/qcs5430-fp2-addons-rb3gen2-vision-mezz.dtb
\
                    qcom/qcs5430-fp2-addons-rb3gen2-vision-mezz-hsp.
dtb \
                    qcom/qcs5430-fp2p5-addons-rb3gen2-vision-mezz.
dtb \
                    qcom/qcs5430-fp2p5-addons-rb3gen2-vision-mezz-
hsp.dtb \
                    qcom/qcs5430-fp3-addons-rb3gen2-vision-mezz.dtb
\
                    qcom/qcs5430-fp2p5-addons-rb3gen2-vision-mezz-
hsp.dtb \
                    "
```

DTB partition

– The generated vfat image named dtb.bin contains the combined DTB image. A dedicated partition named `dtb` is present on the Qualcomm development kits. Flash the `dtb.bin` on this partition.

– UEFI parses the combined DTB present in the `dtb` partition and selects a matching DTB for the hardware.

Managing partitions in Qualcomm Linux

This section describes how you can add, delete, modify, and rename partitions.

Qualcomm Linux uses a pre-generated `partition.xml` file by default. The pre-generated file is delivered as part of the `QCM6490_bootbinaries.zip` file. When you initiate a build, the `partition.xml` file from `QCM6490_bootbinaries.zip` is used and steps 3 to 5 are run from Ptool workflow figure.

To add, delete, modify, or rename partitions, Qualcomm Linux provides configuration files, which define partitions for the UFS device. The configuration files are in the `meta-qcom-hwe/recipes-devtools/partition-utils/qcom-partition-confs/` directory.

By modifying these configuration files and integrating these in the build, you can generate a custom partition layout. When a configuration file from `meta-qcom-hwe/recipes-devtools/partition-utils/qcom-partition-confs/` is used in Qualcomm Linux builds, the workflow spans all the steps from the Ptool workflow figure.

The following table lists the files and tools used for partitioning:

File or tool	Description
Partition XML (mandatory)	Defines the Qualcomm internal XML format.
Ptool (mandatory)	Qualcomm Ptool that converts information in partition XML to GUID Partition Table (GPT) binaries.
Partition layout file (optional)	Defines all partitions for storage in JSON format.
gen_partition tool (optional)	Reads the partition layout file and generates an internal XML format handled by the Qualcomm Ptool.

The partition definitions in JSON format (used by the .conf files), existing partitions, and the tool that generates the GUID partition table for Qualcomm development kits are as follows:

Note: The following partition layout and examples are specific to the UFS.

- **Partition layout file**

  The device partitions the UFS to store various images of the bootchain and the Linux operating system. The configuration file details the partitions configured for a specific Qualcomm development kit. The following configuration file in the workspace defines the UFS partitions for the RB3 Gen 2 development kit: `meta-qcom-hwe/recipes-devtools/partition-utils/qcom-partition-confs/qcm6490-partitions.conf`.

  – **Partition layout file syntax**

  The following examples from the `qcm6490-partitions.conf` file show the syntax of the partition layout file.

  The first entry in the configuration file defines the disk type, disk size, logical block addressing (LBA) size (sector size), and whether the last partition should grow till the last usable LBA.

```
# select disk type emmc | nand | ufs mandatory
# disk size in bytes is mandatory

--disk --type=ufs --size=137438953472 --write-protect-
boundary=0 --sector-size-in-bytes=4096 --grow-last-partition
```

  The `qcm6490-partitions.conf` file specifies the following examples of partitions. A few partitions are defined in LUN0, with each line representing an individual partition.

  Note: In the following examples, the `firmware-qcom-bootbins_1.0.bb` recipe downloads the `partition.xml` file from Qualcomm software center and takes the GUIDs from it.

- Example 1: The `ssd` partition of 8 KB is defined for LUN0. This partition doesn't require a file to be flashed, hence `--filename=` parameter isn't used.

```
--partition --lun=0 --name=ssd --size=8KB --type-
guid=2C86E742-745E-4FDD-BFD8-B6A7AC638772
```

A couple of partitions are defined in LUN1. Each line defines an individual partition:

- Example 2: The `xbl_a` partition of 3604 KB is defined for LUN1. Flash the `xbl.elf` file to this partition.

```
#This is LUN 1 - Boot LUN A
--partition --lun=1 --name=xbl_a --size=3604KB --type-
guid=DEA0BA2C-CBDD-4805-B4F9-F428251C3E98 --filename=xbl.elf
```

- Example 3: The `xbl_config_a` partition of 512 KB is defined for LUN1. Flash the `xbl_config.elf` file to this partition.

```
#This is LUN 1 - Boot LUN A
--partition --lun=1 --name=xbl_config_a --size=512KB --type-
guid=5A325AE4-4276-B66D-0ADD-3494DF27706A --filename=xbl_
config.elf
```

A partition defined in LUN4:

- Example 4: The `aop_a` partition of 512 KB is defined for LUN4. Flash the `aop.mbn` file to this partition.

```
--partition --lun=4 --name=aop_a --size=512KB --type-
guid=D69E90A5-4CAB-0071-F6DF-AB977F141A7F --filename=aop.mbn
```

The options used for each partition entry are as follows:

- Mandatory options:

```
--lun (mandatory for UFS, optional for emmc. Expressed as
number)
--name (name for the partition, a string)
--size (size of the partition, generally expressed in KB)
--type-guid (GUID for the partition)
```

- Optional options:

```
--attributes  (Optional 64 bit attribute, e.g.,
1000000000000004)
--filename    (Name of the file that is flashed to the
partition)
```

```
--readonly    (whether the partition should be read-only,
values true or false)
--sparse      (whether the partition is for a sparsed image,
values true or false)
```

- **Linux operating system partitions**

Partitions	Description
EFI partition	The EFI system partition (ESP) contains `Esp.bin` and a `vfat` file. It contains all the details necessary for the UEFI to enable systemd-boot. For more information on this image, see EFI image.
Rootfs partition	This partition contains the `system.img` image file. This image consists of all the user space libraries and binaries.

- **Partition tool (Ptool)**

  Ptool generates a GUID partition table binary that the QDL tool uses to partition the storage.

  The following figure shows the Ptool workflow:

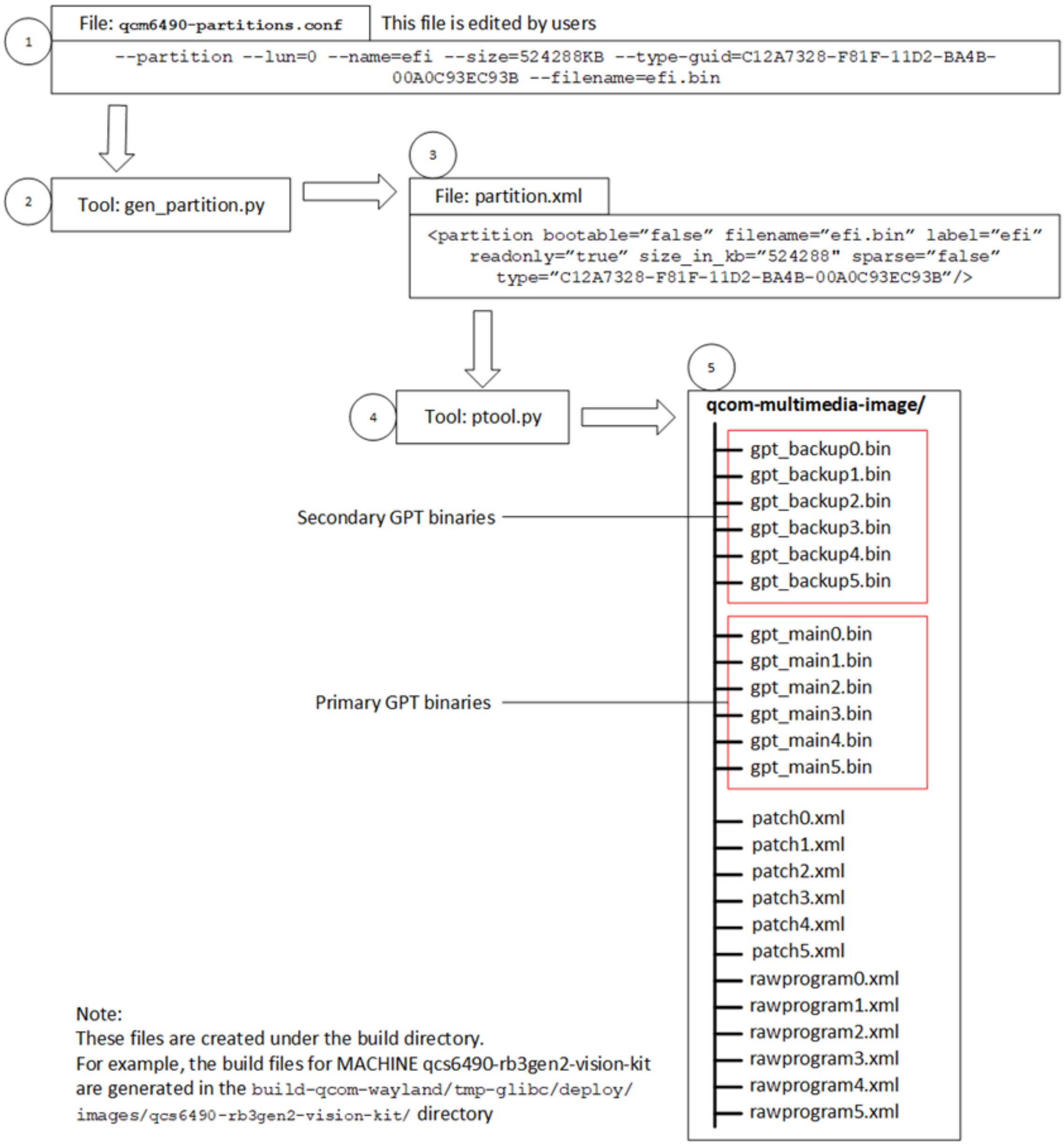


Figure : Ptool workflow

– **Modify partition**

The `qcm6490-partitions.conf` file present in the `meta-qcom-hwe/recipes-devtools/partition-utils/qcom-partition-confs` directory defines all the partitions. The `gen_partition.py` tool processes and generates `partition.xml`, which is a mandatory input for `ptool.py`. As a final step, Ptool generates the `rawprogram.xml`, `patch.xml`, `gpt_main*.bin`, and `gpt_backup*.bin` files, which are required by the QDL tool to flash the system image to the device.

To add a partition, make a partition entry in the configuration file; for example `qcm6490-partitions.conf` with a universal unique identifier (UUID).

After updating the conf file, to switch the preferred provider of virtual/partconf to `qcom-partition-confs_1.0.bb`, add the following line to `local.conf`.

```
PREFERRED_PROVIDER_virtual/partconf = "qcom-partition-confs"
```

After adding the entry, run the BitBake command to generate all necessary files:

```
bitbake <image-name>
```

After the image build completes, if MACHINE selected is qcs6490-rb3gen2-core-kit and the image recipe selected is qcom-console-image, run the following command to flash the image.

```
cd <workspace>/build-qcom-wayland/tmp-glibc/deploy/images/
qcs6490-rb3gen2-core-kit/qcom-console-image
./qdl prog_firehose_ddr.elf rawprogram*.xml patch*.xml
```

Use of Docker containers

Qualcomm Linux enables Docker containers. To use Docker containers, ensure that Qualcomm BSP has enabled the virtualization features. To use containers on Qualcomm Linux devices, ensure that the `meta-virtualization` layer is enabled as part of the workspace.

- **Docker**

  The `qcom-multimedia-image` enables Docker:

  1. For the `qcom-multimedia-image` image recipe, `packagegroup-qcom-containers` is included using `meta-qcom-distro/recipes-products/packagegroups/packagegroup-qcom-multimedia.bb`.
  2. The `packagegroup-qcom-containers` is defined in the `meta-qcom-hwe` layer at `dynamic-layers/virtualization-layer/recipes-containers/packagegroups/packagegroup-qcom-containers.bb`.
  3. After the image is built and flashed, Docker is available on the device.
  4. To verify kernel compatibility, run the `check-config.sh` script on the device. To verify the configuration, enable the required kernel configurations and rebuild the image. For more information, see Kernel compatibility.
  5. To verify the status of the Docker daemon, run the following command and verify the output:

Command	Description	Output
`systemctl status docker`	If Docker commands have been run at least once after boot-up	Active (running)
	If the device has not run any Docker commands after boot-up	Inactive (dead)

The images pulled from the Docker repository for the `linux/arm64` platform and tested to run successfully on the device are as follows:

– `busybox`

– `python`

– `alpine`

– `ubuntu`

– `postgres`

– `mongo`

– `nginx`

– `redis`

The following use cases are verified with Docker enabled:

– Load Docker images from tar files using the Docker load utility.

– Build a Docker image from the Docker file.

– Save Docker images that were pulled/built into a `.tar` file.

– Load Docker images, which are saved as `.tar` files.

– Run multiple container instances of multiple images pulled/built.

– Verify the list of Docker images loaded on the device.

– Verify the list of active and total containers on the device.

– Inspect Docker images using the Docker inspect utility.

– Obtain logs from a container using the Docker logs utility.

– Stop and kill the container using the Docker stop and Docker kill utilities.

– Remove containers and Docker images from the device.

- **Hardware node access from Docker**

  The applications running inside a Docker container may need access to device nodes and files on the device (that is, device is the host for the Docker container). Pass the respective device nodes as an option with the run command for Docker to perform this task:

```
docker run -it --rm --device=<device-1> --device=<device-2>
<docker-image-name>
```

Pass the device nodes located in the `/dev` directory on the device. Depending on the use case, for example, in a graphics scenario, pass `/dev/kgsl-3d0` to Docker using `--device=/dev/kgsl-3d0`.

Applications running inside a Docker container may also need access to storage on the device (target-host). To expose files/directories as bind mounts, run the following command:

```
docker run -it --rm --mount type=bind,source=<source-path-on-
device>,target=<destination-path-inside-container> <docker-
image-name>
```

Run a docker container exposing multiple directories, files, and device nodes to the container using the following command:

```
docker run -it --rm --device=<device-1> --device=<device-2> --
mount type=bind,source=<source-path-on-device-1>,target=
<destination-path-inside-container-1>  --mount type=bind,source=
<source-path-on-device-2>,target=<destination-path-inside-
container-2> <docker-image-name>
```

- **Docker Compose**

  Docker Compose is a tool for defining and running multicontainer applications. It streamlines managing the application stack by defining services, networks, and volumes in a single, comprehensible YAML configuration file. By default, the Qualcomm Linux release includes Docker Compose in the `qcom-multimedia-image`.

  The `docker-compose` package is added to the `packagegroup-qcom-multimedia` package group in the `meta-qcom-distro` metadata layer through `packagegroup-qcom-containers`.

  To run Docker Compose on the device:

  1. Create or copy the Docker Compose YAML files under the writable path on a device such as `/var` (for example `/var/docker-compose-yaml`) to be used with Docker Compose on the device.

  2. Ensure that the Docker images listed in your Docker Compose YAML files are available on the device. If not, the device should be connected to a valid Internet connection to pull the docker images.

  3. To run Docker Compose for multiple YAML files, run the following command:

     ```
     docker-compose -f <docker-compose-file-1>.yml -f <docker-
     compose\u0002file-2>.yml up
     ```

Docker Compose runs and initiates the Docker containers as configured in the Docker Compose YAML file. After the Docker Compose command returns, run `docker ps -a` and `docker logs <conatiner-id>` to verify if the docker containers are running as expected.

To know more about Docker Compose, see Docker Compose overview.

Setup Kubernetes with Qualcomm Linux

Kubernetes is an open-source platform used to automate the deployment, scaling, and management of applications running in a container. It helps organize containers across multiple devices, ensuring efficient resource utilization, high availability, and scalability of applications. Kubernetes is useful not only for large-scale servers and cloud systems but also for small-scale Linux embedded devices. It helps manage and orchestrate containers and applications across IoT devices efficiently. For more information about Kubernetes, see the official website of Kubernetes.

- **Enable Kubernetes**
  - The Qualcomm Linux release enables Kubernetes by default in `qcom-multimedia-image`, through the following changes:
    - The `packagegroup-qcom-k8s` is defined in the `meta-qcom-hwe` layer at `dynamic-layers/virtualization-layer/recipes-containers/packagegroups/packagegroup-qcom-k8s.bb`.
    - The `packagegroup-qcom-k8s` is included in the `recipes-products/packagegroups/packagegroup-qcom-multimedia.bb` image recipe in the `meta-qcom-distro` layer.
  - To verify the status of kubelet after bootup, run the following command and verify the output:

Command	Description	Output
`systemctl status kubelet`	If the device has been set up at least once as a Kubernetes node using kubeadm after bootup	Active (running)
	If the device has not been set up as a Kubernetes node using kubeadm after bootup	Inactive (dead)

Changes made in Docker and Kubernetes in contrast to the `meta-virtualization` solution

- The `meta-qcom-hwe/dynamic-layers/virtualization-layer/recipes-containers/docker/docker-moby_git.bbappend` file modifies `/${systemd_unitdir}/system/docker.service` to add option `--exec-opt native.cgroupdriver=systemd` to the `dockerd` command that runs as part of `ExecStart` of the service. This maintains parity with the `cgroupdriver` between Docker and Kubernetes.

- The `meta-qcom-hwe/recipes-containers/kubernetes/kubernetes_git.bbappend` file modifies `/lib/systemd/system/kubelet.service.d/10-kubeadm.conf` to add `--fail-swap-on=false` as part of `KUBELET_EXTRA_ARGS`. By default, Kubernetes expects swap to be **OFF**, but Qualcomm Linux has swap set to **ON** with the zram feature enabled. Setting `--fail-swap-on=false` allows the kubelet to run even if swap memory is enabled.
- The upstream `kubelet.service` attempts to start automatically upon device bootup, even if the device isn't configured as a Kubernetes node. However, this behavior consumes resources and impacts power. To address this, modify `kubelet.service` by removing `WantedBy=multi-user.target` in the `meta-qcom-hwe/recipes-containers/kubernetes/kubernetes_git.bbappend` file. With `WantedBy=multi-user.target` deleted, the upstream `kubelet.service` starts when the device is set up as a Kubernetes node using `kubeadm`, rather than automatically at bootup as part of `multi-user.target`.

Configure properties

Properties (`property-vault`) provide functionality to store and share key-value pairs stored as strings across the system. Any software component can share any value for a particular key and any other process can access the value by using the same key. `property-vault` allows you to define persistent properties across reboots. Components can use `property-vault` to share specific information, which might be relevant to any other modules on the device. These key-value pairs can be accessed using the command-line interface (CLI).

Note: `property-vault` is supported only in `custom` variant.

- **Configure properties using CLI on Qualcomm device**

  To set a property using the CLI on the Qualcomm device, use the following command:

  ```
  setprop "my-key" "my-value"
  ```

  To access a property using the CLI, use the following command:

  ```
  getprop "my-key"
  ```

  Output of the previous command:

  ```
  my-value
  ```

  To set a property using the CLI and make it persistent across reboots on the device, use the following command:

```
setprop "persist.my-key" "my-value"
```

Note: For a property to persist across reboots, the property key must start with `persist`.

- **Use Properties in Qualcomm Linux C/C++ source files**

  To use the properties in C/C++ source code modules, do the following:

  1. To use properties in a program, for example, in `example-recipe.bb`, add a dependency on `property-vault`:

     ```
     DEPENDS += "glib-2.0 property-vault"
     RDEPENDS:${PN} += "property-vault"
     ```

  2. Change build configuration files:

     a. If you are using `autotools`:

        `configure.ac`

        ```
        ...

        PKG_CHECK_MODULES(GLIB, glib-2.0 >= 2.16, dummy=yes, AC_
        MSG_ERROR(GLib >= 2.16 is required))
        GLIB_CFLAGS="$GLIB_CFLAGS"
        GLIB_LIBS="$GLIB_LIBS"
        AC_SUBST(GLIB_CFLAGS)
        AC_SUBST(GLIB_LIBS)

        AC_CONFIG_FILES([Makefile])
        ...
        ```

        `Makefile.am`

        ```
        root_sbindir        = "/sbin"
        root_sbin_PROGRAMS = property-test

        property_test_SOURCES   = source.c
        property_test_CFLAGS = @GLIB_CFLAGS@
        property_test_LDFLAGS = @GLIB_LIBS@ -lpropertyvault
        ```

     b. If you are using `cmake`:

        `CMakeList.txt`

```
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -std=c99 -lpthread -
lrt -lm -lglib-2.0 -ldl -latomic")
target_link_libraries (your-lib-name propertyvault)
```

3. Modify the source code files to `get` and `set` properties:

   `source.c`

```
#include "properties.h"

int main() {
    // set a property
    property_set("my-key", "my-value");

    // get a property
    char paramstr[PROP_VALUE_MAX];
    // value gets stored in paramstr
    property_get("my-key", paramstr, "default-val");


    // set a property which has to persist across reboots
    property_set("persist.my-key", "my-value");

    // get a persisted property
    property_get("my-key", paramstr, "default-val");

    return 0;
}
```

Note: The property key name can be up to 64 characters long, and the property value can be up to 92 characters long, as defined by `PROP_NAME_MAX` and `PROP_VALUE_MAX` in `properties.h`.

4. Build the recipe on the host computer and include it as part of the image to be flashed on the device.

Persist partition

The BSP software components use the persist partition defined for the UFS to store persistent data across reboots.

Note: The files under persist partition are supposed to remain intact, including reboots and OTA updates. Therefore, erasing or wiping the entire partition to delete files isn't recommended.

- **Persist mount point**

  The `/var/persist` directory is a mount point for a file system created on the persist partition. The `var-persist-mount_1.0.bb` recipe is responsible for installing the `var-persist.mount` systemd unit to `local-fs.target`.

  At boot up, the `var-persist.mount` systemd unit creates `/var/persist` path and mounts `/dev/disk/by-partlabel/persist` on `/var/persist`. To display the persist mount point, run the mount command as follows:

  ```
  sh-5.1# mount | grep persist
  /dev/sda4 on /var/persist type ext4 (rw,relatime,
  rootcontext=system_u:object_r:qcom_persist_t:s0,seclabel,
  stripe=128)
  ```

- **Resize persist partition**

  Persist partition is auto-resized by `resize-partition@persist.service` , which is installed by the `qcom-resize-partitions.bb` recipe. The `resize-partition` service runs at device boot to expand the filesystemd to the maximum available size in the partition.

Create secondary virtual machine

This section provides the steps for creating virtual machine (VM) images using Qualcomm Linux build system.

To use a virtual machine with Qualcomm Linux, ensure that the image recipes include the virtual machine manager (VMM) tools, and generate the Guest VM kernel and root file system images.

The following procedure provides a step-by-step guide for the following:

- Include essential VMM tools such as `crosvm` and `qemu` in the image recipes already defined in Qualcomm Linux.
- Use commands to build Guest VM kernel and root file system images.
- Use commands to launch the Guest VM.

1. **Set up the development machine.**

- Install clang on the host development machine.

  To install clang on the host development machine, run the following command:

  ```
  sudo apt install clang-11
  ```

- Enable the meta-rust layer in Qualcomm Linux environment.

  The meta-rust layer provides Rust compiler (rustc) and package manager (cargo) to compile crosvm, which is implemented in the Rust language.

  Add the meta-rust layer to EXTRALAYERS in the `meta-qcom-distro/conf/bblayers.conf` file.

  Open the `conf/bblayers.conf` file and add the layer path to `EXTRALAYERS` variable as follows:

  ```
  EXTRALAYERS ?= " \
  ${WORKSPACE}/layers/meta-rust \
  "
  ```

- Include Rust from meta-rust in build.

  Create the `meta-qcom-distro/conf/distro/include/rust_version.inc` file with the following content:

  ```
  # include this in your distribution to easily switch between versions
  # just by changing RUST_VERSION variable

  RUST_VERSION ?= "1.73.0"

  PREFERRED_VERSION_cargo ?= "${RUST_VERSION}"
  PREFERRED_VERSION_cargo-native ?= "${RUST_VERSION}"
  PREFERRED_VERSION_libstd-rs ?= "${RUST_VERSION}"
  PREFERRED_VERSION_rust ?= "${RUST_VERSION}"
  PREFERRED_VERSION_rust-cross-${TARGET_ARCH} ?= "${RUST_VERSION}"
  PREFERRED_VERSION_rust-llvm ?= "${RUST_VERSION}"
  PREFERRED_VERSION_rust-llvm-native ?= "${RUST_VERSION}"
  PREFERRED_VERSION_rust-native ?= "${RUST_VERSION}"
  ```

  Include the `rust_version.inc` file in the `meta-qcom-distro/conf/distro/qcom-wayland.conf` file as follows:

  ```
  require conf/distro/include/rust_version.inc
  ```

2. **Add crosvm tool to Qualcomm Linux images.**

To add crosvm tool to the images built by Qualcomm Linux, edit the `meta-qcom-distro/recipes-products/packagegroups/packagegroup-qcom-vm-host.bb` package group recipe file and add the following code block:

```
RDEPENDS:packagegroup-qcom-vm-host:append:qcom-custom-bsp = "\
    crosvm \
    "
```

3. **Build the image recipe and the Guest VM images.**

   - To build an existing Qualcomm Linux image recipe, run the `bitbake <image>` command. This includes the crosvm tool in the image generated.
   - For example, for `qcom-console-image` run the following command:

     ```
     bitbake qcom-console-image
     ```

   - To build a Guest VM image, run the following command:

     ```
     bitbake multiconfig:qcom-guestvm:qcom-guestvm-image
     ```

     Guest VM artifacts

     After the Guest VM image is successfully created, you can find the following artifacts:

     a. The Guest VM build command creates a folder named `tmp-qcom-guestvm-glibc` for the Guest VM image.

     b. The Guest VM kernel and rootfs images are generated in the `tmp-qcom-guestvm-glibc/deploy/images/<machine-name>` directory.

     c. The `tmp-qcom-guestvm-glibc/deploy/images/<machine-name>/Image` file is the Guest VM kernel image.

     d. The `tmp-qcom-guestvm-glibc/deploy/images/<machine-name>/qcom-guestvm-image-<machine-name>.ext4` file is the Guest VM root file system image.

     e. After flashing the images created by the `bitbake <image>` command and booting up the device, copy the Guest VM kernel and root file system images to the device, for example, under the `/var/gunyah` directory.

4. **Launch Guest VM.**

   To launch Guest VM, use the crosvm VMM tool.

   In the following command, the Guest VM kernel and the root file system images are from the `/var/gunyah` directory on the device.

```
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 \
--root /var/gunyah/qcom-guestvm-image-qcs9100-ride-sx.ext4 --no-
balloon --no-rng --params \
"earlyprintk=serial panic=0" /var/gunyah/Image
```

- The `--root /var/gunyah/qcom-guestvm-image-qcs9100-ride-sx.ext4` option specifies the path to the root file system for the Guest VM.
- The last parameter `/var/gunyah/Image` is passed to the `crosvm` tool that specifies the path to the kernel image.

For more information, see Virtualization.

OTA update for Qualcomm Linux

OTA updates are essential for keeping devices functioning, especially embedded systems and IoT devices. These updates allow devices to receive and install updates.

Qualcomm Linux uses capsule update mechanism for updating firmware images, and OSTree update mechanism for updating Linux OS.

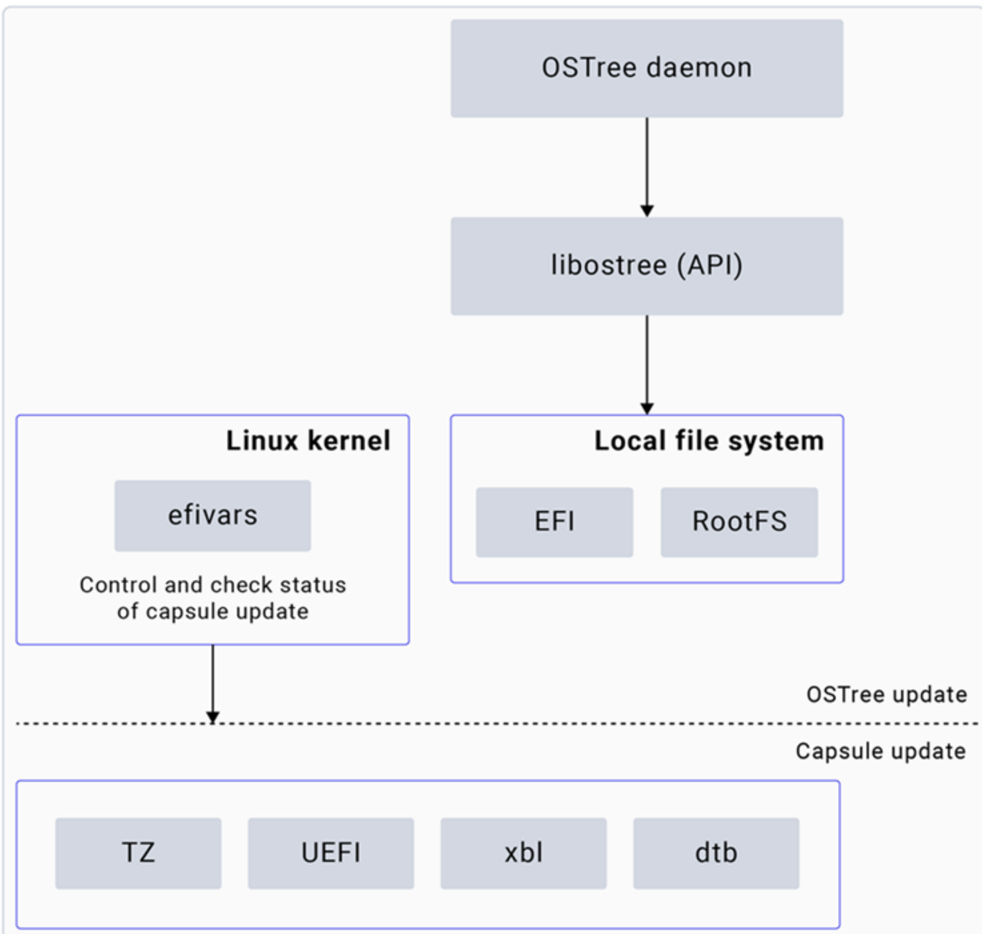


Figure : OTA update for Qualcomm Linux

- **Firmware update using capsule**

  Capsule update is a method for updating firmware on Qualcomm Linux-enabled devices. A UEFI capsule packages the firmware into a binary format. When the device boots up and functions in normal mission mode, it downloads and deploys the capsule to the EFI partition. On reboot, during the next bootup cycle, the UEFI processes this capsule and applies the update to the device firmware.

- **OSTree for Linux operating system updates**

OSTree is a tool for managing version-controlled, atomic updates of Linux-based operating systems. It works like a git repository for the entire Linux filesystem. OSTree stores snapshots of filesystem trees in repositories, which devices pull over the network. With OSTree, updates are atomic and support rollback, so an interrupted update won't break the system. This is useful for IoT or edge devices that need secure, consistent updates that can be rolled back if something goes wrong.

- **Use capsule and OSTree for complete OTA software updates**

  To manage firmware and OS updates in a single OTA system, use capsule and OSTree update mechanisms together. Capsule update mechanism handles the firmware updates first, updating the low-level firmware. After the capsule update, the system reboots and reaches Linux OS, where it checks and applies the OSTree updates.

  Qualcomm Linux uses capsule to update low-level firmware through UEFI.

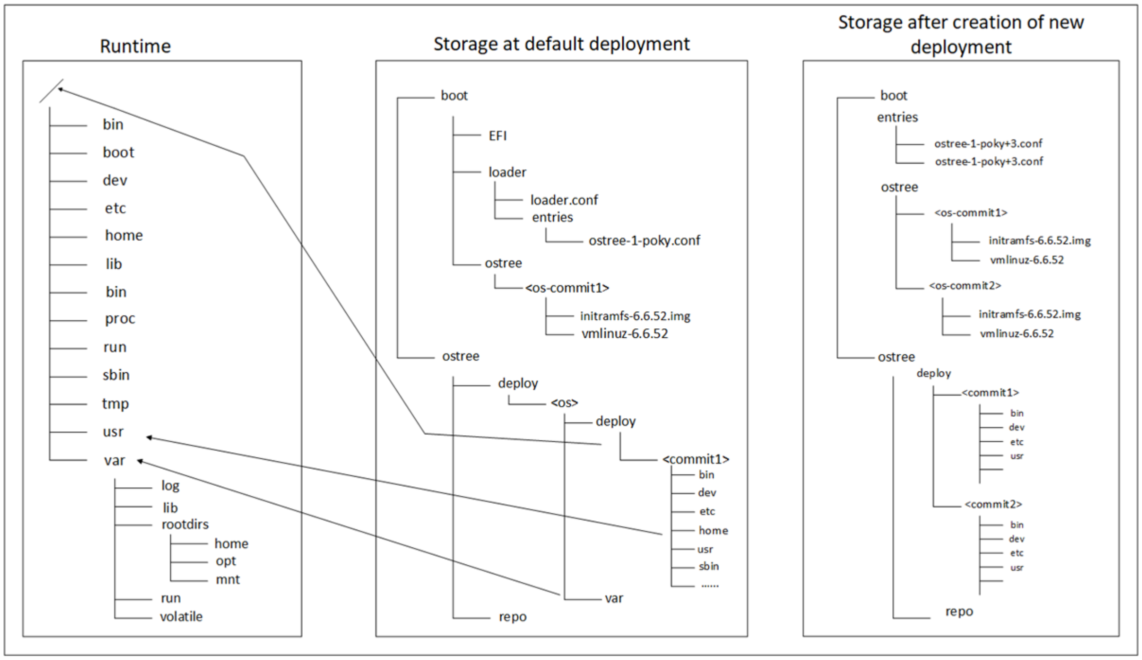
  Following are the steps to update firmware using capsule:

  1. Create capsule: A binary known as a UEFI capsule encapsulates the firmware update.
  2. Delivery: The system delivers the capsule binary to the UEFI by storing it in the mounted `/EFI` path.
  3. Processing: The UEFI firmware processes the capsule during the bootup cycle, and applies the update to the device firmware.

  Qualcomm Linux uses OSTree to manage the Linux OS. The system copies the Qualcomm Linux build-generated update package to the device and stages it for activation using commands listed in Linux OS update flow using OSTree. After the device reboots, it applies the update package.

  The following figure shows the storage view for the Linux OS with OSTree. The **Runtime** view indicates which directories from the **Storage at default deployment** are mapped to the mount-points at runtime.

  The **Storage after creation of new deployment** view shows how the flash storage updates when the device successfully creates a new OSTree deployment. This deployment is newly created and attempts as the new Linux OS on the next reboot.



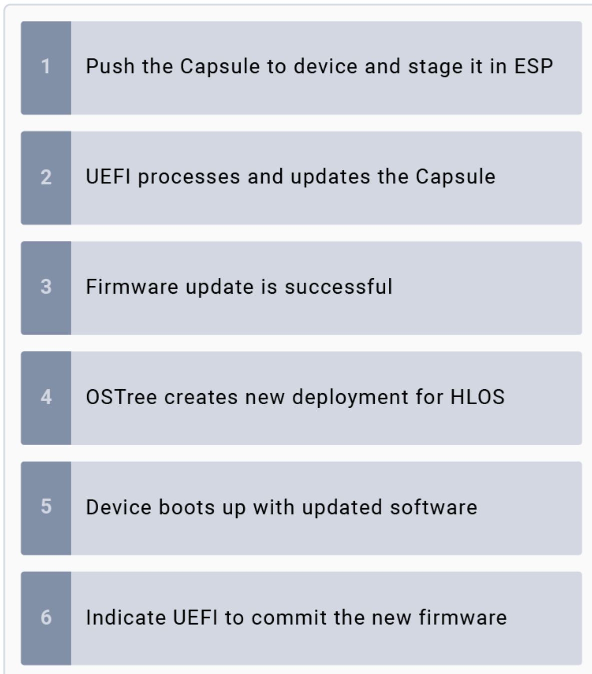
Overview of the storage during OTA update

Following is the list of Linux OS and firmware images updated as part of OTA:

- **Linux OS images**
  - The `efi.bin` file contains the UKI, initrd, and boot loader configuration files. OSTree creates a new configuration file during deployment. It lists the paths to the new kernel and initramfs images copied to the EFI partition.
  - The `system.img` file contains the rootfs, including key components like `/ostree`, `/ostree/repo`, and `/ostree/deploy`. When a new deployment is created, OSTree updates the filesystem tree to reflect the new version of the operating system.
- **Firmware images**

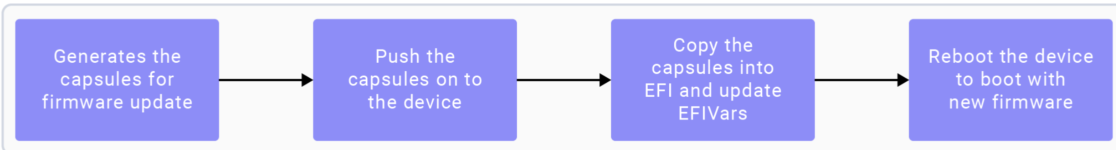
  For information related to the list of firmware images, see GitHub.
- **Capsule update flow**

  Following table shows the capsule and HLOS update flow with the description:

Workflow steps	Description
	1. Copy the `<capsule>.cap` capsule file to EFI partition, which is mounted at `/boot/EFI/UpdateCapsule` on the booted-up device. 2. Set the EFI variable (efivar) OsIndications flags with `EFI_OS_INDICATIONS_FILE_CAPSULE_DELIVERY_SUPPORTED` and reboot the device. UEFI identifies that there is a capsule available for update from OsIndications flag. UEFI authenticates the capsule and updates the firmware images from the capsule. The status of capsule update is updated in the ESRT table. If there is a failure during capsule update, UEFI rolls back the firmware to previous version. 3. UEFI successfully updates the firmware from the capsule, and the device boots up with the new firmware. 4. Copy the OSTree repo to the device. Create a new deployment for HLOS update using OSTree commands, a new configuration file with count tag gets created, reboot the device. 5. Systemd-boot picks the new config file and boots up the kernel and user space from this. The device boots up with the updated firmware and the HLOS software. `systemd-bless-boot.service` marks the new config as good. 6. Reset the TrialBootEnabled flag in OtaStatus efivar to indicate firmware is good. UEFI checks this efivar to commit the new firmware.

Workflow steps	Description

The following figure shows the firmware update flow:



Firmware update using capsule

To update the firmware using capsule, do the following:

1. Run the following command to create the UpdateCapsule folder on the device:

```
adb shell mkdir /boot/EFI/UpdateCapsule
```

2. Copy the capsule to the device:

```
scp -r <firmware_capsule.cap> <user>@<IP_address>:/boot/EFI/
UpdateCapsule/<firmware_capsule.cap>
```

   For more information about `firmware_capsule.cap` capsule generation, see Capsule generation in UEFI.

3. Create the `data.hex` file on the device containing the specified hexadecimal data:

```
echo -e -n "\x4\x0\x0\x0\x0\x0\x0\x0" > data.hex
```

4. Write the contents of `data.hex` on the device to the UEFI variable OsIndications using efivar tool:

```
efivar -n 8be4df61-93ca-11d2-aa0d-00e098032b8c-OsIndications -f
data.hex -w
```

   For more information about UEFI variables, see Update and recovery.

5. Print the value of the OsIndications UEFI variable using efivar tool:

```
efivar -n 8be4df61-93ca-11d2-aa0d-00e098032b8c-OsIndications -p
```

6. To save the efivars to RPMB, run the `uefi_sec` app on the device:

```
/usr/bin/uefi_sec 1
```

7. Reboot the device:

```
reboot
```

8. Check the ESRT table entries:

```
cd /sys/firmware/efi/esrt/entries/entry0
```

Check the output of last_attempt_status command. If it's 0, then the update is successful:

```
cat last_attempt_status
```

Check the output of last_attempt_version command:

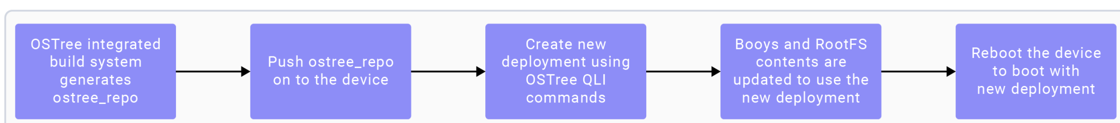
```
cat last_attempt_version
```

Check the output of fw_version command. If last_attempt_version and fw_version are the same, then the update is successful:

```
cat fw_version
```

- **Linux OS update flow using OSTree**

  The following figure shows the Linux OS update flow:



Linux OS update using OSTree

To update Linux OS using OSTree, do the following:

1. To check the current deployment in the Qualcomm device, run the following command:

```
ostree admin status
```

   Output:

```
* poky 643b332dd72b345b5040a1decbe7e04bd2d208a04ba59417aa667c901c485504.0
    Version: 1.0
    origin refspec: poky:qcs6490-rb3gen2-vision-kit
```

   The * shows the current deployment that the device has booted with.

2. The ostree_repo package is in the <workspace>/build-<DISTRO>/tmp-glibc/deploy/images/<MACHINE>/ path on the host development computer. For example, <workspace>/build-qcom-wayland/tmp-glibc/deploy/images/qcs6490-rb3gen2-vision-kit/. Copy the ostree_repo package from the host computer to the Qualcomm device using the following scp command:

```
scp -r <ostree_repo> <user>@<IP_address>:/tmp
```

3. Pull a local OSTree repository on the Qualcomm device.

   Following is the general syntax of the command:

   ```
   ostree pull-local /tmp/<ostree_repo> <branch_name>
   ```

   To find the `branch_name` for the command, run the following command:

   ```
   ostree refs
   ```

   Output:

   ```
   poky:qcs6490-rb3gen2-vision-kit
   ```

   Here, `qcs6490-rb3gen2-vision-kit` is an example branch name.

4. Create the deployment on the Qualcomm device:

   ```
   ostree admin deploy <branch_name>
   ```

   This creates the `ostree-2-poky.conf` configuration file in the `/boot/loader/entries/` directory. For more information, see Systemd boot counting - Successful boot.

5. Reboot the device:

   ```
   reboot
   ```

6. Check if the device is booted with the newly created deployment:

   ```
   ostree admin status
   ```

   Output:

   ```
   * poky 1e8a01bcc5cd9b3b34043db494ddcd01ec5c1c84312479a5525335
   8a92250fa3.0
      Version: 1.0
      origin refspec: qcs6490-rb3gen2-vision-kit
      poky 643b332dd72b345b5040a1decbe7e04bd2d208a04ba59417aa667
   c901c485504.0 (rollback)
      Version: 1.0
      origin refspec: poky:qcs6490-rb3gen2-vision-kit
   ```

   To verify the newly created deployment on the build host computer, check the deployment in the `<workspace>/build-<DISTRO>/tmp-glibc/work/`

<MACHINE>/<IMAGE>/ota-sysroot/ostree/deploy/poky/deploy path. For example, <workspace>/build-qcom-wayland/tmp-glibc/work/qcs6490-rb3gen2-vision-kit/qcom-multimedia-image/ota-sysroot/ostree/deploy/poky/deploy.

- **Systemd-boot counting - Successful boot**

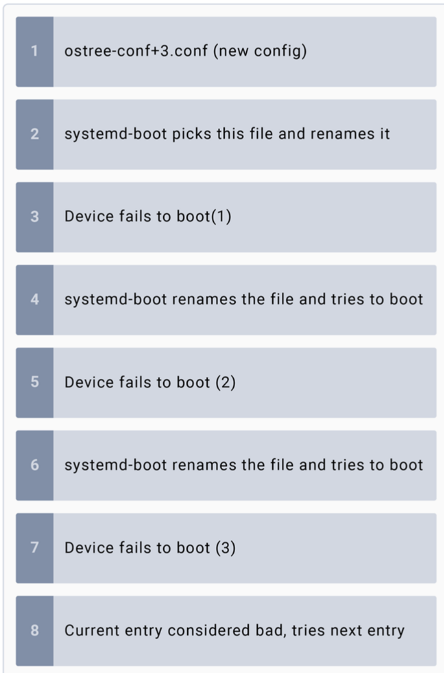
  Following table shows the systemd-boot counting on successful boot with description:

<table>
<tr><th>Workflow steps</th><th>Description</th></tr>
<tr><td>


</td><td>
1. When OSTree deploys a new configuration, it creates a configuration file with a +3 tag in the name, indicating the maximum retry count. This allows boot counting.
2. systemd-boot detects the +3 tag in the entry file name and renames it to ostree-conf+2-1.conf, indicating that one boot attempt has started. After renaming the file, the boot process continues.
3. The systemd-bless-boot-generator creates the systemd-bless-boot.service, which is set to start when boot-complete.target is reached.
4. The systemd-bless-boot.service marks the new configuration as successful by removing the counter tags +2-1 and renaming the file to ostree-conf.conf.
</td></tr>
</table>

- **Systemd boot counting - Unsuccessful boot and rollback**

  Following table shows the systemd-boot counting on unsuccessful boot and rollback with description:

<table>
<tr><th>Workflow steps</th><th>Description</th></tr>
<tr><td>


</td><td>

1. When OSTree deploys a new configuration, it creates a systemd configuration file with a `+3` tag in the name, indicating the maximum retry count, such as `ostree-boot+3.conf`. This enables boot counting.
2. systemd-boot detects the `+3` tag in the configuration file name and renames it to `ostree-boot+2-1.conf`, indicating that one boot attempt has started. After renaming the file, the boot process continues.
3. The `systemd-bless-boot-generator` creates the `systemd-bless-boot.service`, which starts when `boot-complete.target` is reached. If there are any failures during the Linux boot-up, the `systemd-bless-boot.service` does not remove the `+2-1` counter tags from the configuration file.
4. On the subsequent boot, systemd-boot detects the `+2-1` tag in the configuration file name, renames the file to `ostree-boot+1-2.conf`, and tries to boot with it.
5. If the Linux bootup fails on the second attempt, the `systemd-bless-boot.service` does not remove the counter tags `+1-2` from the configuration file.
6. On the next boot, systemd-boot detects the `+1-2` tag in the configuration file name, renames the file to `ostree-boot+0-3.conf`, and tries to boot with it. This is the last attempt to boot Linux deployment.
7. If the device fails to bootup Linux during the third attempt, the `systemd-bless-boot.service` does not remove the counter tags `+0-3` from the configuration file.
8. On the subsequent boot, systemd-boot finds the `+0-3` tag in the configuration file name. As the counter has reached zero, the entry (configuration file) is considered bad. The systemd-boot reverts to an earlier version by trying the valid configuration file entry.

</td></tr>
</table>

Workflow steps	Description

- **Usrmerge**

  The usrmerge feature in Linux simplifies the filesystem layout by merging certain directories under `/usr` path. It combines `/bin`, `/sbin`, and `/lib` with `/usr/bin`, `/usr/sbin`, and `/usr/lib`, respectively.

  With usrmerge, executables and libraries found in `/bin`, `/sbin`, and `/lib` are placed in `/usr/bin`, `/usr/sbin`, and `/usr/lib`. The original directories become symbolic links to their /usr counterparts. This unified structure makes maintenance easier and reduces redundancy, as there is only one location for binaries and libraries instead of separate location for root-level and `/usr` directories. Symbolic links ensure compatibility, allowing scripts and software that reference paths like `/bin` to continue functioning.

  Linux distributions are adopting usrmerge to align with the filesystem hierarchy standard (FHS) recommendations and simplify root filesystems, especially in containerized and embedded systems.

  Distributions like Debian, Ubuntu, and Fedora have adopted usrmerge as part of their system layout, making it a standard in recent versions. The transition involves creating symbolic links and moving any remaining files from the original directories to their `/usr` equivalents.

  In summary, the usrmerge feature consolidates the Linux filesystem, making it more manageable, modernized, and aligned with the evolving needs of system management and containerization.

- **Management of `/var`, `/home`, `/media`, `/mnt`, `/opt`, `/srv`, and `/usr` in Qualcomm Linux**

  OSTree considers `/var` as a persistent directory. This means user/runtime created contents under `/var` remian untouched by OSTree and persist across OTA updates. For more information, see OSTree Overview.

  Other directories that remain untouched by OSTree during an OTA update are `/home`, `/media`, `/mnt`, `/opt`, and `/srv`. OSTree maps these directories as symbolic links as follows:

  - `/home` is a symbolic link to `/var/rootdirs/home`
  - `/media` is a symbolic link to `/var/rootdirs/media`
  - `/mnt` is a symbolic link to `/var/rootdirs/mnt`
  - `/opt` is a symbolic link to `/var/rootdirs/opt`
  - `/srv` is a symbolic link to `/var/rootdirs/srv`

  Any runtime data stored under `/home`, `/media`, `/mnt`, `/opt`, `/srv`, and `/var` stays persistent across OTA updates.

  To maintain a clean and consistent filesystem, OSTree recommends not to install any artifacts under the above directories at build-time. Any artifacts installed in these directories during build-time are not packaged into the rootfs image generated by the Qualcomm Linux build command, that is, `bitbake <image recipe>`.

  To handle runtime creation of files and directories under persistent paths, do the following:

  1. The process creates the files or directories at runtime when it's required.

2. Paths under `/run/`, `/var/lib/`, `/var/cache/`, and `/var/log/` can be created from the respective systemd unit files. Here is a reference.
3. Use systemd-tmpfiles to create files, symbolic links, and directories at bootup.

OSTree creates a read only bind mount at `/usr`, ensuring that the core operating system files remain immutable by users. This approach helps to maintain system integrity and security. OSTree uses the `/usr` mount-point to deploy next update.

Note:

- OSTree allows the installation of files and directories under the `/var/local` path at build time.
- Although OSTree preserves the contents installed under `/usr` during build time, it doesn't package the contents installed under the `/usr/local` subdirectory into the rootfs image.

In Qualcomm Linux with OSTree enabled, the `/etc` directory is managed in a way that allows for both system updates and local customizations.

- The `/etc` directory is mutable, allowing modifications at runtime for maintaining system configurations that need to persist across updates.
- OSTree supports merging configuration files from the `/usr/etc` directory to `/etc`. This allows OSTree to update default configurations while preserving any local changes made.
- When an update is applied, OSTree performs a three-way merge for configuration files in `/etc` using the original version of the file, version of the file in new update and locally modified version of the file.
- If conflicts arise during the merge process, OSTree retains the runtime modifications. This helps maintain system stability and ensures that critical configurations aren't overwritten.

- **SOTA distribution feature**

  The software over-the-air (SOTA) distribution feature allows remote updates for embedded systems and IoT devices. It integrates tools like OSTree for system updates, allowing devices to receive and install updates without physical access. This feature is included in the Qualcomm Linux reference distribution configuration as follows:

  ```
  DISTRO_FEATURES:append = " pam overlayfs acl xattr selinux
  ptest security virtualization tpm usrmerge sota"
  ```

  The SOTA distribution feature is found in the `meta-qcom-distro/conf/distro/include/qcom-base.inc` file.

Handle out-of-tree kernel modules and device tree using recipes

To build an out-of-tree kernel module, see Add kernel module.

Note: Qualcomm Linux supports out-of-kernel device tree overlays only for the `custom` variant.

For device-tree/device-tree blob management, see Platform support.

Boot Linux operating system from SD card

You can boot up Qualcomm development kits from the SD card.

Prepare the SD card by flashing it with `EFI.bin` and `system.img`. Insert the prepared SD card into the SD card slot of the development kit. The development kit boots up from the SD card, which contains a backup or redundant operating system to ensure continuous operation.

To boot Qualcomm development kits from an SD card, follow these steps:

1. Insert the SD card into the Qualcomm development kit. Verify the device node for the inserted SD card, for example, `/dev/mmcblk1` or `/dev/mmcblk2`.

2. To format the SD card, run the `mkfs.ext4 <sdcard slot>` command on the UART shell.

   In the following example, the dev node for the SD card is `/dev/mmcblk1`.

   ```
   mkfs.ext4 /dev/mmcblk1
   ```

   Output:

   ```
   mke2fs 1.46.5 (30-Dec-2021)
   Found a dos partition table in /dev/mmcblk1
   Proceed anyway? (y,N) y
   Discarding device blocks: done
   Creating filesystem with 3889536 4k blocks and 972944 inodes
   Filesystem UUID: 069e04f8-0b72-4a94-aa93-f5ec03bea38d
   Superblock backups stored on blocks:
      32768, 98304, 163840, 229376, 294912, 819200, 884736,
   1605632, 2654208

   Allocating group tables: done
   Writing inode tables: done
   Creating journal (16384 blocks): done
   Writing superblocks and filesystem accounting information: done
   ```

3. Use the `fdisk` command to create two partitions on the SD card to copy `efi.bin` and

system.img.

a. Create the first partition as `vfat` type to copy `efi.bin`.

   To create the partition, run the following command (for example, the dev node is `/dev/mmcblk1`):

   ```
   fdisk /dev/mmcblk1
   ```

   Output:

   ```
   Welcome to fdisk (util-linux 2.37.4).
   Changes will remain in memory only, until you decide to write
   them.
   Be careful before using the write command.

   The device contains 'ext4' signature and it will be removed
   by a write command.
   See fdisk(8) man page and --wipe option for more details.

   Device doesn't contain a recognized partition table.
   Created a new DOS disklabel with disk identifier 0xa5fa6b03.

   Command (m for help): n
   Partition type
      p   primary (0 primary, 0 extended, 4 free)
      e   extended (container for logical partitions)
   Select (default p): p
   Partition number (1-4, default 1): 1
   First sector (2048-31116287, default 2048):
   Last sector, +/-sectors or +/-size{K,M,G,T,P} (2048-31116287,
   default 31116287): +512M

   Created a new partition 1 of type 'Linux' and of size 512
   MiB.

   Command (m for help): w
   The partition table has been altered.
   Calling ioctl() to re-read partition table.
   Syncing disks.
   ```

   To convert the partiton to `vfat` type, run the following command:

   ```
   mkfs.vfat /dev/mmcblk1p1
   ```

   Output:

```
mkfs.fat 4.2 (2021-01-31)
```

Note: In this example, leave the `First sector` field empty and ensure that the `Last sector` is large enough to copy `efi.bin`.

b. Create the second partition as `ext4` type to copy `system.img` as follows:

To create the partition, run the following command (for example, the dev node is `/dev/mmcblk1`):

```
fdisk /dev/mmcblk1
```

Output:

```
Welcome to fdisk (util-linux 2.37.4).
Changes will remain in memory only, until you decide to write
them.
Be careful before using the write command.

Command (m for help): n
Partition type
   p   primary (1 primary, 0 extended, 3 free)
   e   extended (container for logical partitions)
Select (default p): p
Partition number (2-4, default 2): 2
First sector (1050624-31116287, default 1050624):
Last sector, +/-sectors or +/-size{K,M,G,T,P} (1050624-
31116287, default 31116287): +10G

Created a new partition 2 of type 'Linux' and of size 10 GiB.

Command (m for help): w
The partition table has been altered.
Calling ioctl() to re-read partition table.
Syncing disks.
```

To convert the partiton to `ext4` type, run the following command:

```
mkfs.ext4 /dev/mmcblk1p2
```

Output:

```
mke2fs 1.46.5 (30-Dec-2021)
Discarding device blocks: done
```

```
Creating filesystem with 2621440 4k blocks and 655360 inodes
Filesystem UUID: 9e470d01-77fe-4382-a273-ab8b022bdd8b
Superblock backups stored on blocks:
         32768, 98304, 163840, 229376, 294912, 819200,
884736, 1605632

Allocating group tables: done
Writing inode tables: done
Creating journal (16384 blocks): done
Writing superblocks and filesystem accounting information:
done
```

Note: In this example, leave the `First sector` field empty and ensure that the `Last sector` is large enough to copy `system.img`.

4. Verify the partitions created as follows:

```
lsblk -f /dev/mmcblk1
```

Output:

```
NAME FSTYPE FSVER LABEL UUID
FSAVAIL FSUSE% MOUNTPOINTS
mmcblk1
|
|-mmcblk1p1
|    vfat                1EAC-8FF8
`-mmcblk1p2
     ext4                9e470d01-77fe-4382-a273-ab8b022bdd8b
```

5. Copy `efi.bin` and `system.img` from the successfully built Qualcomm Linux image on the host computer to the device using scp command.

```
scp -r <efi.bin> <IP_address>:<path>
scp -r <system.img> <IP_address>:<path>
```

6. To copy `efi.bin` and `system.img` from the device to the partitions on the SD card, run the following commands on the device:

   - `dd if=<path>/efi.bin of=<sdcard slot> bs=4M status=progress`
   - `dd if=<path>/system.img of=<sdcard slot> bs=4M status=progress`

   To copy both images from the `/tmp` path on the device, following is an example of how to copy them to the SD card:

```
dd if=/tmp/efi.bin of=/dev/mmcblk1p1 bs=4M status=progress

dd if=/tmp/system.img of=/dev/mmcblk1p2 bs=4M status=progress
```

Note: In this command, the dev node for the SD card is `/dev/mmcblk1`.

7. The SD card is ready to boot the images. If the EFI partition on UFS is corrupted or systemd-boot fails, the device boots from the SD card to ensure continuous operation.

Use of efivar tool

The efivar tool manages UEFI environment variables stored in the non-volatile firmware storage. Use these variables to configure the UEFI firmware and its environment.

The efivars are key-value pairs stored in the UEFI firmware. They hold settings for boot configuration, system state, and other essential parameters. The efivars manage boot entries and firmware settings, ensuring the system boots correctly and maintains its configuration.

To interact with the UEFI environment variables, use the following command:

```
efivar [-n name] [-f file] [-p] [-w]
```

Command options:

```
-n name: Specify the name of the UEFI variable.
-f file: Load or save variable contents from a file.
-p: Print the value of the specified UEFI variable.
-w: Write to the specified UEFI variable.
```

- **Update the OsIndications variable**

  Following are the steps to update the OsIndications variable and verify the value of the EFI variable:

  1. Create a `data.hex` file containing a specific byte sequence `04 00 00 00 00 00 00 00`.

     ```
     echo -e -n "\x4\x0\x0\x0\x0\x0\x0\x0" > data.hex
     ```

  2. Write the contents of `data.hex` to the EFI variable `8be4df61-93ca-11d2-aa0d-00e098032b8c-OsIndications`.

     ```
     efivar -n 8be4df61-93ca-11d2-aa0d-00e098032b8c-OsIndications
     -f data.hex -w
     ```

3. To verify the value of the EFI variable `8be4df61-93ca-11d2-aa0d-00e098032b8c-OsIndications`, run the following command:

```
efivar -n 8be4df61-93ca-11d2-aa0d-00e098032b8c-OsIndications
-p
```

Output of the previous command:

```
GUID: 8be4df61-93ca-11d2-aa0d-00e098032b8c
Name: "OsIndications"
Attributes:
    Non-Volatile
    Boot Service Access
    Runtime Service Access
Value:
00000000  04 00 00 00 00 00 00 00
```

For more information, see efivars Manual.

3 User customizations

This section provides instructions to customize Qualcomm Linux and includes the following topics:

- Add custom machine configurations
- Modify distribution configurations
- Create image recipes
- Customize the kernel

3.1 Add custom machine configurations

To add a custom machine configuration and rebuild the workspace, do the following:

1. To add a machine, introduce a new machine configuration file at `layers/meta-qcom-hwe/conf/machine/`; for example, `layers/meta-qcom-hwe/conf/machine/test-board.conf`.

2. If the new machine is using the QCS6490 hardware SoC, in the newly created file, add the following content:

```
#@TYPE: Machine
#@NAME: TestBoard
#@DESCRIPTION: Machine configuration for a development board,
based on Qualcomm QCS6490

require conf/machine/include/qcom-qcs6490.inc
```

3. Source the setup-environment script:

```
MACHINE=test-board DISTRO=qcom-wayland QCOM_SELECTED_BSP=custom
source setup-environment
```

Note: If you have access to `meta-qcom-extras`, add it to `EXTRALAYERS ?= ${WORKSPACE}/layers/meta-qcom-extras` in the `conf/bblayers.conf` file.

4. To build an image for the machine added, run the following command:

```
bitbake qcom-multimedia-image
```

3.2 Add custom distribution configurations

The following subsections explain how to add a custom distribution configuration and rebuild the workspace.

Qualcomm defined distribution configuration overview

The reference distribution defined for Qualcomm Linux is at `<workspace>/layers/meta-qcom-distro/conf/distro/qcom-wayland.conf`. The DISTRO_FEATURES variable can be inspected using the following command:

```
bitbake -e | grep ^DISTRO_FEATURES=
```

Output:

```
DISTRO_FEATURES="acl alsa argp bluetooth debuginfod ext2 ipv4 ipv6
largefile pcmcia usbgadget usbhost wifi xattr nfs zeroconf pci 3g nfc
 vfat seccomp pam overlayfs acl xattr selinux ptest security
virtualization tpm wayland vulkan opengl systemd pulseaudio gobject-
introspection-data ldconfig"
```

Note: Distribution features describe the DISTRO_FEATURES shown in the output.

Add distribution configuration

To add a distribution configuration file, do the following:

1. Add a `test-distro.conf` file in `<workspace>/layers/meta-qcom-distro/conf/distro`.
2. Use the same content as `qcom-wayland.conf`, that is, `cp qcom-wayland.conf test-distro.conf`.
3. Set `DISTRO_NAME = "Test Reference Distro with Wayland"` in the `test-distro.conf` file.
4. Set `DISTROOVERRIDES = "test-dist"` in the `test-distro.conf` file.
5. Source the environment and export variables as follows:

```
MACHINE="qcs6490-rb3gen2-core-kit" DISTRO="test-distro" source
setup-environment
```

After sourcing the environment, the current workspace directory changes to `<workspace>/build-test-distro`. To verify if the `test-distro` defined has taken effect, open the `conf/auto.conf` file to confirm the following:

```
# This configuration file is dynamically generated every time
# set_bb_env.sh is sourced to set up a workspace.  DO NOT EDIT.
#--------------------------------------------------------------
DISTRO ?= "test-distro"
```

To verify the output, run the following command:

```
bitbake -e | grep ^DISTROOVERRIDES=
DISTROOVERRIDES="test-dist"
```

Note: If you have access to `meta-qcom-extras`, add it to `EXTRALAYERS ?= ${WORKSPACE}/layers/meta-qcom-extras` in the `conf/bblayers.conf` file.

To rebuild the image, run the following command:

```
bitbake qcom-multimedia-image
```

Enable or disable DISTRO_FEATURES

DISTRO_FEATURES provide a mechanism to verify, which packages must be included in the generated images. You can select the features you want to enable through the DISTRO_FEATURES variable, which is set or appended to in a `test-distro.conf` configuration file of a distribution. For more information, see Distribution features.

Note: Select a feature defined by the community.

1. Open the `test-distro.conf` file and add the following line:

   ```
   DISTRO_FEATURES:append = " cramfs"
   ```

2. Test if the change has taken effect, and then rebuild:

   ```
   bitbake -e | grep ^DISTRO_FEATURES=
   ```

   The output is as follows:

```
DISTRO_FEATURES="acl alsa argp bluetooth debuginfod ext2 ipv4
ipv6 largefile pcmcia usbgadget usbhost wifi xattr nfs zeroconf
pci 3g nfc  vfat seccomp pam overlayfs acl xattr selinux ptest
security virtualization tpm usrmerge sota wayland vulkan opengl
cramfs systemd pulseaudio gobject-introspection-data ldconfig"
```

3. To rebuild, run the following command:

```
bitbake qcom-multimedia-image
```

3.3 Add image recipes

Add image recipes

1. To add an image recipe, change the directory to `<workspace>/layers/meta-qcom-distro/recipes-products/images`.
2. Create a file, for example `test-image.bb`, and add the following content:

```
SUMMARY = "Test image"

LICENSE = "BSD-3-Clause-Clear"

IMAGE_FEATURES += "splash \
    tools-debug \
    debug-tweaks \
    enable-adbd \
"

inherit core-image features_check extrausers image-adbd image-
qcom-deploy image-efi

REQUIRED_DISTRO_FEATURES = "pam systemd"

CORE_IMAGE_BASE_INSTALL += " \
    kernel-modules \
    packagegroup-filesystem-utils \
"

CORE_IMAGE_EXTRA_INSTALL += "overlayfs-qcom-paths"
```

3. Source the environment:

```
MACHINE=qcs6490-rb3gen2-core-kit DISTRO=qcom-wayland source
setup-environment
```

4. The following step is optional and applies only if you have cloned `meta-qcom-extras` in the workspace.

   Open the `conf/bblayer.conf` file and verify the change in the content:

Before change	After change

`# Add your overlay location to EXTRALAYERS`
`# Make sure to have a conf/layers.conf in there`
`EXTRALAYERS ?= ""`
	`# Add your overlay location to EXTRALAYERS`
	`# Make sure to have a conf/layers.conf in there`
	`EXTRALAYERS ?= "${WORKSPACE}/layers/meta-qcom-extras"`

5. Build using the BitBake command:

```
bitbake test-image
```

The generated images are as follows:

```
build-qcom-wayland> $ ls tmp-glibc/deploy/images/qcs6490-rb3gen2-
core-kit/test-image
aop.mbn     gpt_backup0.bin  gpt_backup5.bin  gpt_main4.bin  kernel-
modules.tgz  patch2.xml            prog_firehose_lite.elf
rawprogram2.xml  system.img    xbl_config.elf
cpucp.elf   gpt_backup1.bin  gpt_main0.bin    gpt_main5.bin  logfs_
ufs_8mb.bin   patch3.xml            qdl
rawprogram3.xml  tz.mbn        xbl.elf
devcfg.mbn  gpt_backup2.bin  gpt_main1.bin    hypvm.mbn      multi_
image.mbn     patch4.xml            qupv3fw.elf
rawprogram4.xml  uefi.elf      XblRamdump.elf
dtb.bin     gpt_backup3.bin  gpt_main2.bin    Image          patch0.
xml          patch5.xml            rawprogram0.xml
rawprogram5.xml  uefi_sec.mbn  zeros_5sectors.bin
efi.bin     gpt_backup4.bin  gpt_main3.bin    imagefv.elf    patch1.
xml          prog_firehose_ddr.elf  rawprogram1.xml         shrm.elf
       vmlinux
```

3.4 Customize image features

This section provides examples of how to customize images:

Remove display packages from image

`layers/meta-qcom-distro/recipes-products/packagegroups/packagegroup-qcom-multimedia.bb`

<table>
<tr><th>Before change</th><th>After change</th></tr>
<tr><td>

```
RDEPENDS:${PN}:append:qcom-custom-bsp = "\
    camera-server \
    packagegroup-qcom-audio \
    packagegroup-qcom-bluetooth \
    packagegroup-qcom-camera \
    packagegroup-qcom-display \
    packagegroup-qcom-fastcv \
    packagegroup-qcom-graphics \
    packagegroup-qcom-iot-base-utils \
    packagegroup-qcom-location \
    packagegroup-qcom-video \
    packagegroup-qcom-voiceai \
    "
```

</td><td>

```
RDEPENDS:${PN}:append:qcom-custom-bsp = "\
    camera-server \
    packagegroup-qcom-audio \
    packagegroup-qcom-bluetooth \
    packagegroup-qcom-camera \
    packagegroup-qcom-fastcv \
    packagegroup-qcom-graphics \
    packagegroup-qcom-iot-base-utils \
    packagegroup-qcom-location \
    packagegroup-qcom-video \
    packagegroup-qcom-voiceai \
    "
```

</td></tr>
</table>

After running the `bitbake qcom-multimedia-image` command, the display-related packages are removed from the build. The content from the `packagegroup-qcom-display.bb` package group is removed from the image.

Remove Bluetooth® from image

`layers/meta-qcom-distro/recipes-products/packagegroups/packagegroup-qcom-multimedia.bb`

<table>
<tr><th>Before change</th><th>After change</th></tr>
<tr><td><pre>RDEPENDS:${PN}:append:qcom-
custom-bsp = "\
   camera-server \
   packagegroup-qcom-audio \
   packagegroup-qcom-bluetooth \
   packagegroup-qcom-camera \
   packagegroup-qcom-display \
   packagegroup-qcom-fastcv \
   packagegroup-qcom-graphics \
   packagegroup-qcom-iot-base-
utils \
   packagegroup-qcom-location \
   packagegroup-qcom-video \
   packagegroup-qcom-voiceai \
   "</pre></td><td><pre>RDEPENDS:${PN}:append:qcom-
custom-bsp = "\
   camera-server \
   packagegroup-qcom-audio \
   packagegroup-qcom-camera \
   packagegroup-qcom-fastcv \
   packagegroup-qcom-graphics \
   packagegroup-qcom-iot-base-
utils \
   packagegroup-qcom-location \
   packagegroup-qcom-video \
   packagegroup-qcom-voiceai \
   "</pre></td></tr>
</table>

After running the `bitbake qcom-multimedia-image` command, the bluetooth-related packages are removed from the build. The content from the `packagegroup-qcom-bluetooth.bb` package group is removed from the image recipe.

3.5 Modify partition layout

This section explains how to add, delete, modify, or rename partitions. A configuration file defining partitions for the UFS device are present at `layers/meta-qcom-hwe/recipes-devtools/partition-utils/qcom-partition-confs/qcm6490-partitions.conf`. To add a partition, add a row entry to this file. To remove a partition, delete the corresponding entry to remove the target partition from the set of images to be flashed.

Many partitions are crucial for functionality. To understand the details of how this file plays a role in generating the partition table, see Managing partitions in Qualcomm Linux.

Example 1: Adding a partition

To add a partition with name `test` in LUN0, add the following line to the `qcm6490-partitions.conf` under LUN0 section:

```
--partition --lun=0 --name=test --size=4096KB --type-guid=1B81F7E6-
F50D-419B-A739-2AEFF8DA3335
```

This adds a 4 MB partition test to LUN0 and a GUID determined by you. This partition isn't flashed with any image, but it's available as a raw partition after the device boots up. As this partition is added to LUN0, it shows up at either of the following options:

- `/dev/sda<N>`
- `/dev/disk/by-partlabel/test`

Note: To reflect the changes in `qcm6490-partitions.conf`, update the preferred provider. For more information, see Modify partition.

Example 2: Add an example partition with a binary to be flashed to the newly created partition

To add a partition with name `test1` in LUN0, add the following line to `qcm6490-partitions.conf` under LUN0 section:

```
--partition --lun=0 --name=test --size=4096KB --type-guid=1B81F7E6-
F50D-419B-A739-2AEFF8DA3335 --filename=test1.bin
```

Deploy the new `test1.bin` binary in `build-qcom-wayland/tmp-glibc/deploy/images/qcs6490-rb3gen2-core-kit/$(image_name).`

3.6 Use of devtool

The following examples show the devtool usage for Qualcomm software components:

- **Kernel**

  See Yocto support.

- **QDL tool**

  Use the devtool to modify the QDL source in the workspace created.

  ```
  devtool modify qdl
  INFO: Source tree extracted to <workspace>/build-qcom-wayland/
  workspace/sources/qdl
  INFO: Using source tree as build directory since that would be
  the default for this recipe
  INFO: Recipe qdl now set up to build from <workspace>/build-
  qcom-wayland/workspace/sources/qdl
  ```

  1. The following tree is checked-out locally:

     ```
     tree -L 2 build-qcom-wayland/workspace/
     build-qcom-wayland/workspace/
     ├── appends
     │   └── qdl_git.bbappend
     ├── conf
     ```

```
│   └── layer.conf
├── README
└── sources
    └── qdl
```

2. Inspect the checked-out QDL source-tree:

```
ls workspace/sources/qdl/
firehose.c  LICENSE  Makefile  patch.c  patch.h  program.c
program.h  qdl.c  qdl.h  README  sahara.c  ufs.c  ufs.h
util.c
```

3. Change the source tree under `workspace/sources/qdl/` and build your changes:

```
devtool build qdl
devtool build-image qcom-multimedia-image
```

4. The generated images are in the path `build-qcom-wayland/tmp-glibc/deploy/images/qcs6490-rb3gen2-core-kit/qcom-multimedia-image`.

- **Weston**

```
devtool modify weston
INFO: Adding local source files to srctree...
INFO: Source tree extracted to <workspace>/build-qcom-wayland/
workspace/sources/pulseaudio
INFO: Recipe weston now set up to build from <workspace>/build-
qcom-wayland/workspace/sources/weston
```

1. The following tree is checked-out locally:

```
tree -L 2 build-qcom-wayland/workspace/
build-qcom-wayland/workspace/
├── appends
│   └── weston_10.0.2.bbappend
├── conf
│   └── layer.conf
├── README
└── sources
    └── weston
```

2. Inspect the checked-out Weston source-tree:

```
ls workspace/sources/weston/
clients           COPYING      desktop-shell     include
libweston          meson.build        oe-local-files
```

```
pipewire    releasing.rst  tests  weston.ini.in
compositor        data          doc                ivi-shell
libweston-desktop  meson_options.txt  oe-logs
protocol    remoting        tools  xwayland
CONTRIBUTING.rst  DCO-1.1.txt  fullscreen-shell  kiosk-shell
man                notes.txt         oe-workdir      README.
rst  shared         wcap
```

3. Change the source tree under `workspace/sources/weston/` and build your changes:

```
devtool build weston
devtool build-image qcom-multimedia-image
```

4. The generated images are in the path `build-qcom-wayland/tmp-glibc/deploy/images/qcs6490-rb3gen2-core-kit/qcom-multimedia-image`.

3.7 Customize kernel

To customize the kernel, see Qualcomm Linux Kernel Guide.

3.8 Add third-party layers to workspace

To add a third-party layer to the workspace, perform the following steps:

1. Clone the layer under `<WORKSPACE>/layers/`.
2. Add the layer in the `layers/meta-qcom-distro/conf/bblayers.conf` file as part of the `BBLAYERS` variable, as follows:

```
# These layers hold machine specific content, aka Board Support
Packages
BSPLAYERS ?= " \
  ${WORKSPACE}/layers/meta-testlayer \
  ${WORKSPACE}/layers/meta-qcom-hwe \
  ${WORKSPACE}/layers/meta-qcom \
"
```

Cloning and adding the layer allows the BitBake to parse the layer.

3.9 Create a build for debugging

To generate a debug build to export `DEBUG_BUILD=1` before issuing Yocto BitBake build instructions, run the following command:

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

To understand which kernel defconfig and config fragments are used when `DEBUG_BUILD` is set to **1**, see Kernel configuration, under Kernel recipes.

3.10 Create a build for most optimized boot KPI

To achieve the best bootup time for the device, use the `PERFORMANCE_BUILD` variable. When the `PERFORMANCE_BUILD` variable is set to **1**, it modifies `KERNEL_CMDLINE_EXTRA` to drop the **console** option from the command line. The `PERFORMANCE_BUILD` variable takes effect when `DEBUG_BUILD` is set to **0**.

To understand how the `PERFORMANCE_BUILD` variable affects the boot up time, see the following table:

	PERFORMANCE_ BUILD="0"	**PERFORMANCE_ BUILD="1"**
`DEBUG_BUILD=0`	Console on UART is enabled	• The console on UART is disabled • Best boot up time
`DEBUG_BUILD=1`	Console on UART is enabled	• The `PERFORMANCE_ BUILD="1"` when `DEBUG_BUILD` is set to `"1"`.

4 Debug

This information explains how to customize the Yocto-based workspace, and resolve general problems and issues.

4.1 Verify QDL and ModemManager

If you are using a Linux distribution with `systemd`, use `systemctl` command to stop `ModemManager`. The following is an example from Ubuntu 22.04:

1. To verify the `ModemManager` status, run the following command:

```
systemctl status ModemManager
```

```
ModemManager.service - Modem Manager
Loaded: loaded (/lib/systemd/system/ModemManager.service;
enabled; vendor preset: enabled)
Active: active (running) since Tue 2023-11-28 16:28:15 IST; 3
months 4 days ago
Main PID: 1338 (ModemManager)
Tasks: 3 (limit: 4915)
CGroup: /system.slice/ModemManager.service
        └─1338 /usr/sbin/ModemManager --filter-policy=strict
```

```
ps aux | grep –i modemmanager
```

```
root      1338  0.0  0.0 434332  9544 ?        Ssl   2023  10:39
/usr/sbin/ModemManager --filter-policy=strict
```

2. To stop `ModemManager`, run the following command:

```
systemctl stop ModemManager
systemctl status ModemManager
```

```
ModemManager.service - Modem Manager
Loaded: loaded (/lib/systemd/system/ModemManager.service;
```

```
enabled; vendor preset: enabled)
Active: inactive (dead) since Sun 2024-03-03 20:08:32 IST; 4s
ago
Process: 1338 ExecStart=/usr/sbin/ModemManager --filter-
policy=strict (code=exited, status=0/SUCCESS)
Main PID: 1338 (code=exited, status=0/SUCCESS)
```

3. The `ps aux` command doesn't show any entry for `/usr/sbin/ModemManager`. If you need `ModemManager`, you must restart it after the flashing is complete and verify if it has started:

```
systemctl start ModemManager
systemctl status ModemManager
```

```
ModemManager.service - Modem Manager
Loaded: loaded (/lib/systemd/system/ModemManager.service;
enabled; vendor preset: enabled)
Active: active (running) since Sun 2024-03-03 20:11:46 IST; 43s
ago
Main PID: 14785 (ModemManager)
Tasks: 3 (limit: 4915)
CGroup: /system.slice/ModemManager.service
        └─14785 /usr/sbin/ModemManager --filter-policy=strict
```

```
ps aux | grep -i modemmanager
```

```
root      14785  4.6  0.0 434332  9160 ?        Ssl  20:11   0:00
/usr/sbin/ModemManager --filter-policy=strict
```

4.2 Verify newly added layer excluded from build

If BitBake didn't parse a newly added layer, the recipes from that layer aren't included in the image. Run the following command, and verify that you see the layer in the output:

```
bitbake -e | grep ^BBLAYERS=
```

If you can't find the layer, confirm the contents of the `conf/bblayers.conf` file, to ensure that the layer is included here:

```
# These layers hold recipe metadata not found in OE-core, but lack
any machine or distro content
BASELAYERS ?= " \
```

```
  ${WORKSPACE}/layers/meta-openembedded/meta-oe \
  ${WORKSPACE}/layers/meta-openembedded/meta-filesystems \
  ${WORKSPACE}/layers/meta-openembedded/meta-networking \
  ${WORKSPACE}/layers/meta-openembedded/meta-perl \
  ${WORKSPACE}/layers/meta-openembedded/meta-python \
  ${WORKSPACE}/layers/meta-openembedded/meta-gnome \
  ${WORKSPACE}/layers/poky/meta \
  ${WORKSPACE}/layers/poky/meta-poky \
  ${WORKSPACE}/layers/meta-security \
  ${WORKSPACE}/layers/meta-selinux \
  ${WORKSPACE}/layers/meta-virtualization \
"

# These layers hold machine specific content, aka Board Support
Packages
BSPLAYERS ?= " \
  ${WORKSPACE}/layers/meta-qcom-hwe \
  ${WORKSPACE}/layers/meta-qcom \
"

# Add your overlay location to EXTRALAYERS
# Make sure to have a conf/layers.conf in there
EXTRALAYERS ?= "${WORKSPACE}/layers/meta-qcom-extras"

BBLAYERS = " \
  ${WORKSPACE}/layers/meta-qcom-distro \
  ${EXTRALAYERS} \
  ${BASELAYERS} \
  ${BSPLAYERS} \
"
```

4.3 Verify QA issue: Version going backwards

When using the same workspace for `base` and `custom` builds, you can observe the following signatures:

```
ERROR: <package>-<version> do_packagedata_setscene: QA Issue: Package
version for package wpa-supplicant-src went backwards which would
break
package feeds (from 0:2.10.qcom-r0 to 0:2.10-r0) [version-going-
backwards]
```

For example:

```
ERROR: wpa-supplicant-2.10-r0 do_packagedata_setscene: QA Issue:
Package
version for package wpa-supplicant-src went backwards which would
break
package feeds (from 0:2.10.qcom-r0 to 0:2.10-r0) [version-going-
backwards]
```

This quality assurance (QA) issue occurs with the `custom` variant of recipes in the `meta-qcom-hwe` layer. For example, the `wpa_supplicant` recipe has its version set as `2.10.qcom`. The `.qcom` in the version indicates that the recipes build a different source tree compared to the recipe in `poky/meta/recipes-connectivity`.

When you build the `base` variant after building the `custom` variant, the BitBake build system detects that the version is regressing from `wpa-supplicant_2.10` to `wpa-supplicant-2.10.qcom`. BitBake warns that this regression could cause problems if you use it for creating package feeds.

To avoid this QA issue or to set up package feeds without problems, do any of the following:

- Create different workspaces for `base` and `custom` build variants.
- In the same workspace, create separate build directories as follows:

For build variant	Command to create a build directory	Created build directory
`base`	`MACHINE=qcs9100-ride-sx DISTRO=qcom-wayland QCOM_SELECTED_BSP=base \ source setup-environment build-qcom-wayland-base`	`/<workspace>/Qualcomm_Linux/build-qcom-wayland-base/`
`custom`	`MACHINE=qcs9100-ride-sx DISTRO=qcom-wayland QCOM_SELECTED_BSP=custom \ source setup-environment build-qcom-wayland-custom`	`/<workspace>/Qualcomm_Linux/build-qcom-wayland-custom/`

5 References

5.1 Related documents

Document title	Document number
Qualcomm Technologies, Inc.	
Qualcomm Linux Build Guide	80-70018-254
Qualcomm Linux Release Notes	RNO-250403001134
Qualcomm Linux Kernel Guide	80-70018-3
Qualcomm Linux Security Guide	80-70018-11
Qualcomm Linux Debug Guide	80-70018-12
Qualcomm Intelligent Robotics Product (QIRP) SDK 2.0 User Guide	80-70018-265
Resources	
The Yocto Project	http://git.yoctoproject.org/
Yocto Project documentation	https://docs.yoctoproject.org/5.0.6/singleindex.html
Qualcomm manifest	https://github.com/qualcomm-linux/qcom-manifest
poky/meta	https://git.yoctoproject.org/poky/
meta-openembedded	https://git.openembedded.org/meta-openembedded/
meta-selinux	https://git.yoctoproject.org/meta-selinux/
meta-virtualization	https://git.yoctoproject.org/meta-virtualization/
Classes	https://github.com/qualcomm-linux/meta-qcom-hwe/tree/scarthgap/classes
GitHub	https://github.com/qualcomm-linux/meta-qcom-hwe
GitHub	https://github.com/qualcomm-linux/meta-qcom-hwe/tree/scarthgap/conf/machine
GitHub	https://github.com/qualcomm-linux/meta-qcom-hwe/tree/scarthgap/recipes-firmware
GitHub	https://github.com/qualcomm-linux/meta-qcom-realtime

Document title	Document number
GitHub	https://github.com/qualcomm-linux/meta-qcom-distro/tree/scarthgap/conf/distro
GitHub	https://github.com/qualcomm-linux/meta-qcom-distro/tree/scarthgap/recipes-products/images
GitHub	https://github.com/quic/host-signing-tool
Kernel	https://git.codelinaro.org/clo/la/kernel/qcom.git
realtime	https://wiki.linuxfoundation.org/realtime/start
INIT_MANAGER	https://docs.yoctoproject.org/5.0.6/singleindex.html#term-INIT_MANAGER
Distribution features	https://docs.yoctoproject.org/5.0.6/singleindex.html#distro-features
Image features	https://docs.yoctoproject.org/5.0.6/singleindex.html#image-features
Conditional syntax (Overrides)	https://docs.yoctoproject.org/bitbake/2.8/bitbake-user-manual/bitbake-user-manual-metadata.html#conditional-syntax-overrides
systemd-boot	https://wiki.archlinux.org/title/systemd-boot
Docker Compose	https://docs.docker.com/compose/gettingstarted/
OSTree Overview	https://ostreedev.github.io/ostree/introduction/

5.2 Acronyms and terms

Acronym or term	Definition
aDSP	Advanced digital signal processor
cDSP	Compute digital signal processor
DB	Database
DBX	Forbidden signatures database
EFI	Extensible firmware interface
efivar	EFI variable
eMMC	Embedded multimedia card
ESP	EFI system partition
KEK	Key exchange key
KVM	Kernel-based virtual machine
LBA	Logical block addressing
OE	OpenEmbedded
OS	Operating system
OSS	Open-source software

Acronym or term	Definition
OTP	One time provisioning
PE	Portable executable
Ptool	Partition tool
QDL	Qualcomm download tool
SoC	System on chip
SOTA	Software over-the-air
UEFI	Unified extensible firmware interface
UFS	Universal flash storage
UKI	Unified kernel image
UUID	Universal unique identifier
VFAT	Virual file allocation table
VMM	Virual machine manager
WLAN	Wireless local area network

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.