See all versions of this document

Xilinx Quick Emulator User Guide

QEMU

UG1169 (v2020.1) June 3, 2020

XILINX®

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
06/03/2020 Version 2020.1	
QEMU Supported Features	Updated table.
Using CAN with Xilinx QEMU	New section.

Table of Contents

XILINX

Chapter 1

Using Xilinx QEMU

What is QEMU?

Xilinx provides a Quick Emulator (QEMU) for software developers targeting Zynq®-7000 SoC, Zynq® UltraScale+™ MPSoC, and MicroBlaze™ development platforms. This system-emulation-model runs on an Intel-compatible Linux or Windows host systems. To use this system emulation model you must be familiar with:

- Device architecture
- GNU debugger (GDB) for debugging QEMU remotely
- Generation of guest software application using Xilinx® PetaLinux and Software Development Kit (SDK) tools
- Device trees

This document provides the basic information to familiarize, use, and debug software with QEMU.

Getting Started with QEMU

QEMU Supported Features

The following table summarizes the status each element of the QEMU model according to the delivery.

Table 1: **QEMU Supported Features**

Description	QEMU Status
Application Processing Units	
Arm® interrupt controller (GIC v2 and v3)	Yes
Arm v8 (A53 and A72) implementation. Quad Core.	Little endian only
Arm v8 EL0 support	AArch64 and AArch32
Arm v8 EL1 support	AArch64 and AArch32
Arm v8 EL2 support	AArch64
Arm v8 EL3 support	AArch64
Arm v8 Crypto instruction support	Supported
Vector Floating Point (VFP) support	As maintained by mainline. No formal acceptance criteria to feature.
SIMD support	As maintained by mainline. No formal acceptance criteria to feature.
Arm v7 Support	A9, R5, R4 supported.
Realtime Processing Units	
Dual core Cortex™-R5F	Incomplete coverage of system register set, little endian only.
Dual core R5 CPU run-time configuration	Static dual core, no parallel/lock transitioning.
Fault Handling	Faults can be externally injected.
Tightly coupled Memories	No R5 local view. Only globally accessible TCM memory region is accessible. Flat memory only, no control register implementation.
Interrupt controller	Yes.
SLCRs	Very limited functionality. Only dummy registers, except SD is_MMC control.
PMU Zynq UltraScale+ MPSoC	
IP Integrator	Limited Connectivity specific to PMU functionality.
Global Registers	Yes.
PMU MicroBlaze™	Yes.
PMU Interrupt Control	Yes.

Table 1: **QEMU Supported Features** *(cont'd)*

Description	QEMU Status
I/O Peripherals and Devices	
I/O Peripherals	Not all peripherals are implemented. Some standard peripherals are slight variations on the actual cores configuration-wise.
Cadence Gigabit Ethernet Controller	1588 not supported.
SD Host Controller Interface (v3.0)	Yes.
SD Card model	No SDxC.
QSPI controller (excludes Linear and Generic)	Yes.
QSPI linear region	No XIP. Slow emulation performance.
QSPI NOR flash devices	Incomplete but reasonable selection of parts including many modern QSPI capable devices.
UART Controller	Yes.
SPI controller	Master mode only.
I2C controller	Master mode only.
DDR	Simple flat RAM model, no ECC.
CAN	Yes.
XADC	No.
GPIO	Limited functionality: connects to remote port.
MDIO and Ethernet PHY	Dummy models, show link up on requested PHY using MDIO.
USB	No.
SATA	Yes.
PCI™	Yes.
OSPI	Yes.
RTC	Yes.
Display Ports	
DP model	AUX Communication. DPCP: DisplayPort Configuration Information. EDID.
DPDMA	Yes
2 Layers	Yes.
Alpha Blending	Yes.
Audio	With some unexpected behavior.
Dynamic resolution changes	Yes.
Multiple Pixel formats	Not all.
Mali™ GPU	No.
AMBA® AXI Bus	
AMBA/AXI bus interconnect system	Simple bus model, no AXI/AMBA-specific features (such as MIDs). Master IDs and Trustzone (secure versus non-secure) transactions supported.
Bus quality of service monitoring and control	N/A
On Chip Memory	Yes

Table 1: **QEMU Supported Features** *(cont'd)*

Description	QEMU Status
AXI Performance Monitor (APM) ATM AXI Trace Monitor (ATM)	N/A
Additional Zynq UltraScale+ MPSoC Capabilities	
XMPU	Does not return Slave error; CPU does not recognize asynchronous aborts on failed accesses.
XPPU	Does not return Slave error; CPU does not recognize asynchronous aborts on failed accesses.
SMMU	Only supports 64-bit page tables
Clock/reset controllers for low-power and high power domains.	Limited feature set specific to CPU functionality.
Interprocessor Interrupt Control	Yes.
PL-based AMS block	N/A.
Miscellaneous QEMU Non-IP Related Feature	
Ability to boot multiple software in different CPUs.	Yes.
Create QEMU Machine models from Linux device tree binaries (DTB)s.	Limited to QEMU maintained DTBs only. IPI/HSI generated DTBs unsupported.
FPDDMA	No FCI and no rate-control.
LPDDMA	No FCI and no rate-control.
MTTCG	Yes.
Timers and Clock Generators	
Triple Timer Counter	Yes.
SWDT, WDT	No.
Si570/71	I2C device. Dummy emulation of clock generator.

Installing QEMU

QEMU comes with the Xilinx® PetaLinux Tools and Xilinx SDK installer. See the PetaLinux Tools documentation for installation instructions.

TIP: *Use* `which-qemu-system-aarch64` *command to know where the QEMU binary is installed after PetaLinux or SDK paths are set.*

TIP: *Xilinx QEMU Source is available at https://github.com/Xilinx/qemu, for building QEMU manually refer to Chapter 3, QEMU Quick Reference Card.*

Launch QEMU Using PetaLinux Commands

QEMU is integrated with the PetaLinux workflow.

It is assumed that at this step you have already downloaded and installed Xilinx PetaLinux on your Linux machine. If you have not done so, please see the PetaLinux Tools documentation for installation instructions, and install it now.

For this example, to get you started quickly without having to have a Vivado® project ready, Xilinx® recommends you to download a pre-built PetaLinux BSP. Also, download the `ZCU102 BSP (prod-silicon)` BSP from the Petalinux Download Page.

To use QEMU with a PetaLinux project, you need to create and build a PetaLinux project for the Zynq® UltraScale+™ MPSoC platform (use the pre-built ZCU102 BSP).

Open your favorite terminal and type the following:

1. `source <petalinux-install-path>/settings.csh`
2. `petalinux-create -t project -s <path to bsp>/xilinx-zcu102-v201X.X-final.bsp -n xilinx-qemu-first-run`
3. `cd xilinx-qemu-first-run`
4. `petalinux-boot --qemu --prebuilt 3`

On completing step 4 you should see the QEMU boot sequence loading the prebuilt linux image. At the prompt enter the username and passowrd as `root`.

Note: On step 4 you can pass additional arguments to QEMU using the following option: `--qemu-args "...."`. You can pass any additional argument within double quotes.

QEMU DTBs

The QEMU tool uses device tree blobs (dtb) for machine creation. For MicroBlaze™ and Zynq®-7000, dtbs built for Linux kernels are used for emulation. Xilinx maintains device-tree sources at https://github.com/Xilinx/qemu-devicetrees for Zynq UltraScale+ MPSoC. For more information on how to use device trees see Chapter 3, QEMU Quick Reference Card.

QEMU Quick Reference Card

Zynq UltraScale+ MPSoC Command Base Template

This is a basic template for Zynq® UltraScale+™ MPSoC QEMU command lines:

```
qemu-system-aarch64 -M arm-generic-fdt -nographic -dtb <hw-dtb> \
-device loader,file=<progam.elf>,cpu-num=<cpu-id> \
-global xlnx,zynqmp-boot.cpu-num=<cpu index> \
[-device loader,addr=<cpu-reset-register-addr>,data=<value>,data-len=4] \
[-global xlnx,zynqmp-boot.use-pmufw=true] \
[-global xlnx,zynqmp-boot.load-pmufw-cfg=<false/true>] \
[-boot mode=<boot-mode-id>] \
[-m <ddr ram size>]
[-drive file=<image-path>, if=<( sd | mtd | pflash )>, format=raw,\
index=<index_num> ]
```

QEMU Command Options and Descriptions

The following table lists the options and descriptions for QEMU commands.

Table 2: **QEMU Command Options and Descriptions**

Option	Description
`-device loader,file=<progam.elf>,cpu=<cpu-id>`	Specify the software to run (in ELF format).
`-device loader,addr=<cpu-reset-register-addr>,data=<value>,data-len=4`	Release CPU from reset. This command is optional because the next command can also be used to release resets of A53s. ***Note***: R5 resets can be released only with this command.
`-global xlnx,zynqmp-boot.cpu-num=<cpu index>`	Release the specified A53 core. For details regarding the CPU index, refer to CPU Enumeration.
`-global xlnx,zynqmp-boot.use-pmufw=<true/false>`	Specify true if running with pmu firmware. ***Note***: This option will be deprecated in future, when pmufw uses a default configuration.

Table 2: **QEMU Command Options and Descriptions** *(cont'd)*

Option	Description
`-global xlnx,zynqmp-boot.load-pmufw-cfg=<false/true>`	Mention QEMU to use or not to use static pmufw config data. • true: loads static pmufw config data • false: Do not use static data, FSBL populates it
`-boot mode=<boot-mode-id>`	Specify the boot mode pins.
`-drive file=<image-path>, if=<[ sd | mtd | pflash ]>, format=raw, index=<index_num>`	Specify files for persistent storage media (SD, QSPI, or NAND respectively). <index_num> specifies the respective controller for each media type.
`-M arm-generic-fdt -nographic -dtb <hw-dtb>`	Standard options. <hw-dtb> is the QEMU machine description.
`-m <ddr ram size>`	Mention size of ram to be emulated, default is 2G. ex: -m 4G

Standalone Hello World Example

This example runs a “Hello-world” ELF file on an A53 processor. Substitute the `hello_world.elf` with the location of your target software, as follows:

```
qemu-system-aarch64 -nographic -M arm-generic-fdt \
-dtb zcu102-arm.dtb \
-device loader,file=./hello_world.elf,cpu-num=0 \
   -device loader,addr=0xfd1a0104,data=0x8000000e,data-len=4
```

Zynq-7000 SoC Command Base Template

Following is the command base template for Zynq-7000 SoC devices.

```
qemu-system-aarch64 -M arm-generic-fdt-7series \
 [-machine linux=on]  \
-serial /dev/null -serial mon:stdio \
 -display none \
-kernel <guest image path> \
-dtb <dtb path> \
[-device loader,addr=0xf8000008,data=0xDF0D,data-len=4 \
-device loader,addr=0xf8000140,data=0x00500801,data-len=4 \
-device loader,addr=0xf800012c,data=0x1ed044d,data-len=4 \
-device loader,addr=0xf8000108,data=0x0001e008,data-len=4 ]
```

Table 3: **Command Options for Zynq-7000 SoC Devices**

Option	Description
`-machine linux=on`	Specifies if -kernel is provided an linux kernel image.

Table 3: **Command Options for Zynq-7000 SoC Devices** *(cont'd)*

Option	Description
`-kernel <guest image path>`	Specify path to guest image, such as kernel
`-dtb <dtb path>`	Specify the linux dtb path which is used by QEMU emulation

MicroBlaze Processor Command Base Template

Following is the command base template for MicroBlaze™ processors.

```
qemu-system-microblazeel -M microblaze-fdt-plnx \
-m <ram_size> \
-serial mon:stdio \
-display none \
-kernel <guest image path> \
-dtb <path to the dtb>
```

Table 4: **Command Options for Zynq-7000 SoC Devices**

Option	Description
`-m <ram_size>`	Specifies the RAM needed to be created.
`-kernel <guest image path>`	Specifies the RAM needed to be created.
`-dtb <dtb path>`	Specifies the RAM needed to be created.

QEMU Command Line Options

This section provides details of QEMU command line options. The following sections detail boot, network, serial, storage, and miscellaneous command line options.

Using Extra QEMU Command Line Arguments with PetaLinux

The `petalinux-boot --qemu` command has an argument `--qemu-args` that lets you specify extra QEMU command line arguments.

Some of the optional arguments specified in the section can be passed to the PetaLinux QEMU using this switch.

IMPORTANT! *Do not specify the standard, boot, or network options to petalinux-boot; those options are handled transparently by the application.*

QEMU Boot Options

```
[ -device loader,(file=<file_name>|data=<value>,data-len=4),\
[addr=<value>],[cpu-num=<value>],[force-raw=true] ] ...
```

This (repeatable) argument configures the QEMU machine for boot. The boot options perform the following tasks:

- Loads software or data into RAM sections
- Sets the CPU entry points
- Releases CPUs from reset

By default, theZynq UltraScale+ MPSoC has six Arm® CPUs (four cores of Cortex™-A53 and two cores of Cortex-R5F) are in reset by their respective reset controllers when no software is loaded. You can use a combination of `-device` loader arguments to load software and setup the CPUs.

There are two basic modes for the loader argument: *file mode* and *single transaction* mode. Specify one mode only. The following subsections describe these modes.

File Mode

In file mode, the loader accepts a file as data to load. The file can be in any format and is passed using the `file=<file_name>` sub-option. If the file is an ELF or a U-Boot image, the file is parsed and the sections loaded into memory as specified by the image; otherwise, the file is assumed as a raw image and loaded accordingly as an image into memory. Specify the address for loading raw images with the `addr=<value>` argument. The address default is 0. The address is ignored if the file is an ELF or U-Boot image. Optionally, you can specify a CPU using the `cpu-num=<value>` sub-option.

***Note*:** Specifying `cpu-num` also sets the Program Counter (PC).

If specified, the CPU has a set entry point.

- For ELFs and U-Boot images, the address is set as specified by the image.
- For raw images, the entry point is set to the start address.
- If you do not specify a CPU, the bus for CPU0 loads images, but no program entry point is set. See CPU Enumeration, for more details on the meaning of `<value>`.

There are cases where you might want to treat an ELF or a U-Boot image as a raw data image (particularly useful for testing bootloaders with ELF or U-Boot capability). You can pass the `force-raw=true` sub-option to instruct the loader to treat the image as raw in this case. You must specify the `addr` ; in this case and the section information in the ELF or U-Boot image are ignored.

Single Transaction Mode

In single transaction mode, a single bus transaction occurs.

- data `addr` and `data-len` must be specified.
  - `data-len` must equal 4 (corresponding to a single 4-bit transaction).
  - `addr` must be aligned to 4-bits.

Before the initial system reset for Zynq UltraScale+ MPSoC, the QEMU performs the specified bus transaction. The initial system reset might clear the value set by the single transaction when the transaction accesses the I/O peripherals. As a work-around, key registers interpret bit 31 (usually reserved) as an indicator to not reset that particular register when resetting this system. This is useful for releasing CPU resets from the command line. The registers that support bit-31 `reset-ignore` are:

- `CRF.RST_FPD_APU`
- `CRL.RST_LPD_TOP`
- `PMU_LOCAL.LOCAL_RESET`
- `RPU.RPU_GLBL_CNTL`

Optionally, you can specify a CPU using the `cpu-num=<value>` sub-option. The single transaction occurs from the perspective of the specified CPU. If you do not specify a CPU, then the system assumes `CPU0`. See the *Zynq UltraScale+ MPSoC Register Reference* (UG1087) for more information about registers.

CPU Enumeration

The `cpu-num=<value>` argument interprets value are shown in the following table:

Table 5: **CPU Enumeration Values**

Value	CPU
0	A53-0
1	A53-1
2	A53-2
3	A53-3

Table 5: **CPU Enumeration Values** *(cont'd)*

Value	CPU
4	R5-0
5	R5-1

Advanced

If you edit the `/cpus dtb` node, these enumerations change. The enumerations of the CPUs matches the `DTS /cpus` node ordering.

Hot Loading

You can use the loader at runtime to load new software into an already running system. This is accessible from the QEMU monitor. See the Non-Graphical I/O Option for information on accessing the monitor. From the monitor, you can stop the emulation using the `stop` command:

```
(qemu) stop
```

You can then use the loader to add new software or release CPUs from reset. The syntax is:

```
(qemu) device_add loader,(file=<file>|data=<value>,data len=4),\
[addr=<value>],[cpu-num=<value>],[force-raw=true]
```

All sub-options are the same as described in the previous section. The emulation can then be resumed (with the new memory and CPU state from the loading operations) using the following `c` command:

```
(qemu) c
```

QEMU Command Examples

The following table provides command examples:

Table 6: **Command Examples and Descriptions**

Description	Command
Load an ELF onto Cortex™-A53-0.	`-device loader, file=./hello_world.elf,cpu-num=0`
Release Cortex-A53-0 from reset.	`-device loader,addr=0xfd1a0104, data=0x8000000e,data-len=4`
Load a binary image into RAM at a specific address (no CPU entry point will be set).	`-device loader,file=./Image,addr=0x0008000`
Load an ELF to RAM (no CPU entry point will be set).	`-device loader,file=./foo.elf`

Table 6: **Command Examples and Descriptions** *(cont'd)*

Description	Command

Load an ELF to Cortex-A53-0 from the monitor in an already running system.	`(qemu) stop`
	`(qemu) device_add loader,file=./foo.elf,cpu-num=0`
	`(qemu) device_add loader,addr=0xfd1a0104,data=0x8000000e,data-len=4`
	`(qemu) c`

Using Single Transactions to Unlock CPUs

You can unlock CPUs by writing into system configuration registers using single transactions. The A53-0, 1, 2, 3 reset register is `CRF_APB.RST_FPD_APU`. The following table lists the arguments that can unlock certain CPU combinations:

Table 7: **Single Transaction Unlock Arguments**

Argument	Command
A53-0	`-device loader addr=0xfd1a0104,data=0x8000000e,data-len=4`
A53-1	`-device loader addr=0xfd1a0104,data=0x8000000d,data-len=4`
A53-2	`-device loader addr=0xfd1a0104,data=0x8000000b,data-len=4`
A53-3	`-device loader addr=0xfd1a0104,data=0x80000007,data-len=4`
All A53	`-device loader addr=0xfd1a0104,data=0x80000000,data-len=4`

Similarly the `R5-0,1` reset register is `CRL_APB.RST_LPD_TOP`. As `R5-0` and R5-1 can work in split-mode or lockstep mode, split-mode/lock-step requires extra configuration for `R5`, which is done using register `RPU.RPU_GLBL_CNTL`.

Table 8: **Cortex™-R5 Registers and Commands**

Register	Command
R5-0 (split mode)	`-device loader,addr=0xff5e023c,data=0x80008fde,data-len=4` `-device loader,addr=0xff9a0000,data=0x80000218,data-len=4`
R5-1 (split mode)	`-device loader,addr=0xff5e023c,data=0x80008fdd,data-len=4` `-device loader,addr=0xff9a0000,data=0x80000218,data-len=4`
Both R5 (split mode)	`-device loader,addr=0xff5e023c,data=0x80008fdc,data-len=4` `-device loader,addr=0xff9a0000,data=0x80000218,data-len=4`
Lockstep Mode	`-device loader,addr=0xff5e023c,data=0x80008fde,data-len=4`

See the *Zynq UltraScale+ MPSoC Register Reference* (UG1087) for more information.

Hardware Specifications

`-M arm-generic-fdt ( -hw-dtb | -dtb ) <file> ...`

These arguments are required for all supported boot flows. The `-M` argument to QEMU specifies the QEMU machine to create. In this case, you are selecting the `arm-generic-fdt` machine option, which tells QEMU to parse a device tree binary (or DTB) for machine generation.

QEMU automatically creates CPUs and peripherals for each node in the device tree it has a corresponding device model for (QEMU has a library of known DTS compatible strings).

The `-dtb` or `-hw-dtb` argument specifies the DTB describing the system.

***Note*:** For Zynq UltraScale+ MPSoC device, dts are available on github: https://github.com/Xilinx/qemu-devicetrees.git. For Zynq-7000 SoC device and MicroBlaze™ processor kernel, dtb can be used.

The Zynq UltraScale+ MPSoC DTB is also available in a PetaLinux project at:

```
<proj_path>/images/linux/zynqmp-qemu-arm.dtb
```

or

```
<proj_path>/pre-built/linux/images/zynqmp-qemu-arm.dtb
```

Advanced

DTSs for QEMU are available in the following directory:

```
<project_directory>/project-spec/meta-user/recipes-bsp/device-tree/
files/.
```

Example

```
qemu-system-aarch64 -nographic -M arm-generic-fdt \-dtb ./images/
linux/zynqmp-qemu-arm.dtb
```

-dtb vs -hw-dtb

For Linux Kernel boots, QEMU supports a flow where different DTBs are used for machine generation and Linux Kernel boot. In this flow, both `-dtb` and `-hw-dtb` are specified on the command line.

The `-hw-dtb` is used for machine generation and `-dtb` is passed to the Linux Kernel (using a memory buffer). When `-dtb` is passed, QEMU removes the nodes that it cannot emulate and later copies it to RAM for kernel boot.

***Note*:** This procedure is applicable only when kernel is passed on QEMU command line.

For standalone flows, these two arguments are fully interchangeable; specify only one or the other.

CAUTION! *For Zynq UltraScale+ MPSoC, the QEMU DTB is different from the kernel* `system.dtb`.

QEMU DTS are different for Zynq UltraScale+ MPSoC single and multi-architecture models. The following DTBs are available in PetaLinux project.

- Single-arch : `zynqmp-qemu-arm.dtb`
- Multi-arch : `zynqmp-qemu-multiarch-arm.dtb`, `zynqmp-qemu-multiarch-pmu.dtb`

Non-Graphical I/O Option

```
-nographic
```

By default, QEMU attempts to create a display for user I/O. This option instructs the QEMU that there is no need for a display and I/O is serial.

QEMU attaches the invoking terminal to the serial port in this case (in the default use cases, this is `UART0`). See Serial Options for more information and choices.

In this mode, the QEMU monitor (a command line interface for sending control commands to QEMU) is multiplexed on `stdio`. To switch between the serial port and the monitor, use the following command:

```
CTRL-a c
```

Boot Mode

```
-boot mode=<value>
```

This command line argument selects the value of boot mode pins.

Multiple-Architecture QEMU (Zynq UltraScale+ MPSoC)

A multiple architecture (Multi-Arch) QEMU is a special concept in which more than one instance of QEMU of completely different architecture can communicate with each other and can run together using socket communication.

This version of QEMU supports running the Arm® Cortex-A53s and Cortex-R5Fs, and the MicroBlaze™ device power management unit (PMU).

The multi-architecture version of QEMU needs different device tree binaries (DTBs) than what is necessary for Single-Architecture.

Multi-Architecture DTB

The following are PetaLinux DTBs:

- `zynqmp-qemu-multiarch-arm.dtb`: DTB for the qemu-system-aarch64
- `zynqmp-qemu-multiarch-pmu.dtb`: DTB for qemu-system-microblazeel

In `qemu-devicetrees` repositories, you can find them in the following path:

- `qemu-devicetrees/LATEST/MULTI_ARCH/zcu102-arm.dtb`: DTB for the qemu-system-aarch64
- `qemu-devicetrees/LATEST/MULTI_ARCH/zynqmp-pmu.dtb`: DTB for qemu-system-microblazeel

Single-Architecture DTB

- `zynqmp-qemu-arm.dtb`: DTB for qemu-system-aarch64
- `qemu-devicetrees/LATEST/SINGLE_ARCH/zcu102-arm.dtb`

Machine-path

- `-machine-path [./qemu-tmp]`
- `-machine-path` takes any folder path as argument, and QEMU uses that area for creating Unix sockets for remote-port links and shared memories.

***Note*:** These are mandatory arguments for using Multiple-Architecture.

As mentioned above machine-path works with Unix sockets. To use tcp sockets use chardev devices as in the following example.

- To create RP links for APU and PMU:
  - `-chardev socket,id=pmu-apu-rp,host=<hostname>,port=<port-num>[,server]`
- To create RP links for APU and Cosim:
  - `-chardev socket,id=pl-rp,host=<hostname>,port=<port-num>[,server]`

See the *Zynq UltraScale+ MPSoC: Software Developers Guide* (UG1137) for steps on running QEMU with Arm and PMU.

Storage Media

Several disk and storage media interfaces are modeled. You can pass each to a regular file(s) to use for stored data. QEMU updates the files so the data can be persistent across multiple sessions.

Argument Format

The format is: `-drive file=<image-path>,if=(mtd|sd|pflash),format=raw,index=<value>[,readonly]`

The argument allows specification of extra options such as marking the file as read-only. The argument can also be used to specify the index of the device, allowing specifying files for devices in an order-independent way.

QSPI

QSPI is modeled with the flash specified in DTS. The SPI flashes can connect in a dual-parallel arrangement. The flash file size should match the flash model size.

If you are using only a single mode QSPI, then only one QSPI argument is needed. For each QSPI flash, if an image is not provided, QEMU still models the flash, but initializes with `NULL` data and discards the data after QEMU exits. The data can be written and read back within a single session in this case.

Flash Striper Utility

In parallel mode, the QSPI data passes in for each flash is unique to that flash chip. Because the QSPI controller implements bit-striping in dual parallel mode, a special utility is needed to take a single QSPI data image and format it into two images. The syntax is as follows:

```
flash_strip_bw <input> <out1> <out0>
```

where:

- `<input>` is a 128MB image.
- `<out0>` and `<out1>` are the two 64MB images passable to the `-mtdblock` arguments for QSPI.

This can also be reversed by taking two striped images and converting them back to a single 64MB image as shown in the following command:

```
flash_unstrip_bw <output> <in1> <in0>
```

Building the Flash Strip

Compile for your host with the following commands:

```
SOURCE=flash_stripe.c
gcc $SOURCE -o flash_strip_bw -DFLASH_STRIPE_BW
gcc $SOURCE -o flash_unstrip_bw -DUNSTRIP -DFLASH_STRIPE_BW
```

Download the flash strip utilities (flashstrip.c) from the QEMU wiki page. `flash_strip_be_bw` is also available as part of the PetaLinux tools. The following table lists the supported SPI Flash models.

Table 9: **Supported QSPI Flash Models**

Vendor	Flash Models
Atmel	at25fs010, 25fs040, at25df041a, at25df321a, at25df641, at26f004, at26df081a, at26df161a, at26df321, at45db081d
EON -- en25xxx	en25f32, en25p32, en25q32b, en25p64, en25q64
GigaDevice	gd25q32, gd25q64
Intel/Numonyx -- xxxs33b	160s33b, 320s33b, 640s33b n25q064
Macronix	mx25l2005a, mx25l4005a,mx25l8005, mx25l1606e, mx25l3205d, mx25l6405d mx25l12805d, mx25l12855e, mx25l25635e, mx25l25655e
Micron	n25q032a11. n25q032a13, n25q064a11, n25q064a13, n25q128a11, n25q128a13 n25q256a11, n25q256a13, n25q512a11, n25q512a1
Spansion -- single (large) sector size only, at least for the chips listed here (without boot sectors)	s25sl032p, s25sl064p, s25fl256s0, s25fl256s1, s25fl512s, s70fl01gs, s25sl12800, s25sl12801, s25fl129p0, s25fl129p1, s25sl004a, s25sl008a, s25sl016as, 25sl032a s25sl064a, s25fl016k, s25fl064k
Winbond -- w25x “blocks” are 64k, “sectors” are 4KiB	w25x10, w25x20, w25x40, w25x80, w25x16, w25x32, w25q32, w25q32dw, w25x64. w25q64, w25q80, w25q80b, w25q256
Numonyx	25q128
SST	sst25vf040b, sst25vf080b, sst25vf016b, sst25vf032b, sst25wf512, sst25wf010, sst25wf020, sst25wf040, sst25wf080
ST Microelectronics	m25p05, m25p10, m25p20, m25p40, m25p80, m25p16, m25p32, m25p64, m25p128, n25q032, m45pe10, m45pe80, m45pe16, m25pe20, m25pe80, m25pe16, m25px32, m25px32-s0, m25px32-s1, m25px64

SPI

For each SPI Flash, if an image is not provided QEMU still models the flash, but initializes with NULL data and discards the data after QEMU exits. Data can be written and read back within a single session in this case.

SD

QEMU models an SD card for `-drive file=<file_path>,if=sd`.

The SD card model in QEMU is generic and does not attempt to model a specific physical part. The size of the input file initializes the size of the emulated SD card. Only 512MB SD images are officially supported, although powers of two around that order of magnitude will work.

IMPORTANT! *SDXC (>32GB) sizes do not work.*

If an SD argument is not specified, no SD card is modeled, the corresponding SD slot is ejected.

***Note*:** This is different form the SPI, where if there is no argument those modes still model a physical device.

EMMC

QEMU models an EMMC card for `-drive file=<file_path>,if=sd`. EMMC connects to its respective host interface controllers, based on the `slcr` settings. For `zynqmp index=2` works as EMMC card connected to `sdhci0`.

The size of the input file initializes the size of the emulated MMC card. Only 512MB images are supported, although powers of two, around that order of magnitude will work.

EEPROM

QEMU models EEPROMS connected to I2C. A back-end file can be passed as follows:

`-drive file=<file_path>,if=mtd, index=<id>.`

Users can find the information on which I2C controller eeproms are connected in respective board DTS.

Passing Bootable Images

Boot images are passed to the QEMU using the command line with the storage media options. This is useful for testing, or testing with FSBL or U-Boot bootloaders. See the Bootgen documentation in the *Zynq UltraScale+ MPSoC: Software Developers Guide* (UG1137) for how to create bootable images.

See the Example command for a comprehensive suite of examples for bootable image commands.

Chardev Options

`-chardev backend, id=id [,options]`

The -chardev arguments lets you create a character device. This can be thought of as a file descriptor that routes text from inside QEMU to outside QEMU. The -chardev arguments consists of three main parts:

1. The output
2. If the `chardev` is muxable
3. The ID of this chardev

Character Device Option Examples

```
-chardev null,id=id[,mux=<on|off>]
-chardev socket,id=id[,host=host],port=port,[to=to],[ipv4],[ipv6]\ ,
[nodelay],[reconnect=seconds],[server],[nowait][,telnet]\
,[reconnect=seconds] [mux=<on|off>] (tcp)
-chardev socket,id=id,path=path[,server][,nowait][,telnet]\ ,
[reconnect=seconds],[mux=<on|off>] (unix)
-chardev udp,id=id[,host=host],port=port[,localaddr=localaddr]\ ,
[localport=localport],[ipv4],[ipv6],[mux=on|off]
-chardev msmouse,id=id, [mux=on|off]
-chardev vc,id=id[[,width=width][,height=height]][[,cols=cols]\
,[rows=rows]] ,mux=on|off]
-chardev ringbuf,id=id,[size=size]
-chardev file,id=id,path=path,[mux=on|off]
-chardev pipe,id=id,path=path[,mux=on|off]
-chardev pty,id=id[,mux=on|off]
-chardev stdio,id=id[,mux=on|off][,signal=on|off]
-chardev serial,id=id,path=path[,mux=on|off]
-chardev tty,id=id,path=path[,mux=on|off]
-chardev parallel,id=id,path=path[,mux=on|off]
-chardev parport,id=id,path=path[,mux=on|off]
```

STDIO

The following is an example `chardev`:

```
-chardev stdio,mux=on,id=terminal
```

In this case anything sent to the `chardev` is redirected to the standard I/O. This `chardev` supports muxing and is called `terminal`.

Server (TCP Socket)

The following is an example of using `chardev` to connect to a server:

```
-chardev socket,id=terminal,host=localhost,port=4444,mux=on
```

The server can be `nc`, in this case use:

```
nc -k -l localhost 4444
```

Socket (UNIX Socket)

The following is an example of connecting to a standard UNIX socket:

```
-chardev socket,id=output,path=/tmp/socket,mux=on
```

The socket can be created by nc by using:

```
nc -k -lU /tmp/socket
```

pty

The following is an example of connecting to a pseudo-terminal:

```
chardev pty,id=output,mux=on
```

There is no wait on this chardev to get connected to pty. You can open pty c with any serial terminal: Minicoy, putty, screen.

```
screen /dev/pty/188
```

QEMU puts out on which pty it is connected.

```
“char device redirected to /dev/pts/188 (label IO-base)”
```

Serial Options

```
-serial <arg>
```

By default, the QEMU connects the invoking terminal to `UART0` to provide the user I/O (see Non-Graphical I/O Option for more information.). You can override this behavior by providing at least one explicit `-serial` argument. The following table lists the supported values for `<arg>`:

Table 10: **Serial Arguments and Effects**

<arg>	Effects
`/dev/null`	Disconnect this particular serial.
`mon:stdio`	Connect this serial and monitor to the terminal.
`stdio`	Connect this serial to terminal.
`telnet::<port>,server, nowait`	Create a localhost telnet server on `<port>` for the serial connection. It can be accessed by: `telnet localhost <port>`.
`chardev: dev`	Connects serial to a backend; for example, to a socket, pipe, or terminal.

Serial Command Examples

The following are some of the common non-default serial setups:

- Disconnect all serials: `-serial /dev/null -serial /dev/null`
- Connect `UART1` to the terminal and ignore `UART0`: `-serial /dev/null -serial mon:stdio`
- Connect `UART0` to terminal and `UART1` to telnet: `-serial mon:stdio -serial telnet::1234,server,nowait`
- Connect `UART0` to chardev socket:`-serial chardev:terminal`

***Note*:** `terminal` is the `Chardev` device ID. For details see Chardev Options.

Using CAN with Xilinx QEMU

XlnxCAN is based on SocketCAN, QEMU CAN bus implementation. Zynq UltraScale+ MPSoC devices support two CANs: CAN0 and CAN1. Bus connection and socketCAN interface for each CAN module is set through command lines.

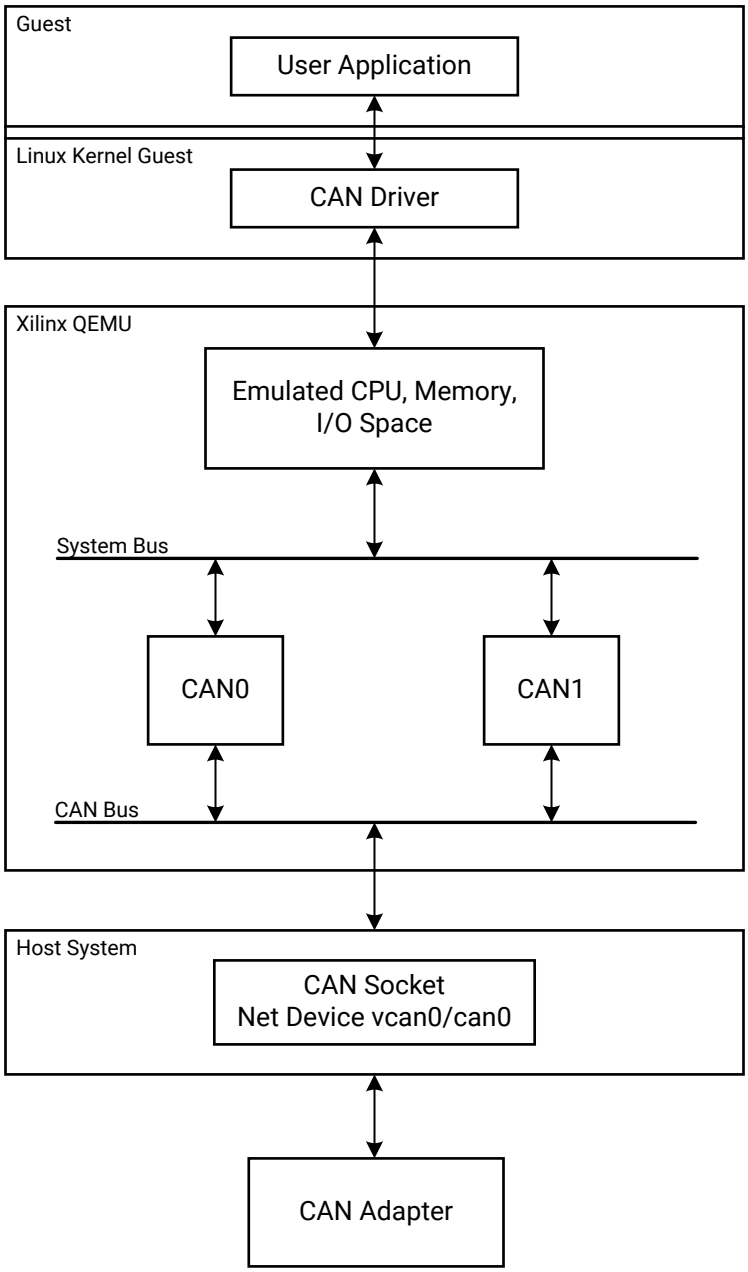
IMPORTANT! *SocketCAN is supported with Linux only. If this is not already installed on the host Linux machine, install it using* `sudo apt -get install can-utils`

You can use the following three commands to initialize a CAN device for Xilinx QEMU. These commands will be appended with the ARM instance.

1. `-object can-bus,id=canbus0`: Creates a new `canbus0`.
2. `-global driver=xlnx.zynqmp-can,property=canbus0,value=canbus0`: Connects the CAN0 controller with the above-created `canbus0`.
3. `-object can-host-socketcan,id=socketcan0,if=vcan0,canbus=canbus0`: Connects CAN0 (`canbus0`) to host system CAN bus (which is virtual CAN socket `vcan0` in this example). Any data transferred by CAN0 is sent to the `vcan0` socket and goes to all devices connected on this `vcan0` interface.

Before launching QEMU with CAN devices, you need to set up CAN interfaces on the host machine. Linux supports Virtual CAN and a physical CAN interface. A virtual CAN interface can be created easily. The physical CAN interface needs a physical CAN bus/adapter on the host machine to work.

Figure 1: **Overview of CAN with QEMU**



Creating a Virtual CAN Interface on the Host Machine

The following commands create a virtual CAN interface:

```
sudo modprobe vcan
sudo ip link add dev vcan0 type vcan
sudo ip link set up vcan0
```

Creating a Physical CAN Interface on the Host Machine

The host system's CAN interface must beconfigured for proper bitrate and set up. The configuration is not propagated from emulated devices through the bus to the physical host device. The following is an example configuration for 1 Mb/s:

```
ip set can0 type can bitrate 1000000
ip set can0 up
```

Creating a Single CAN with QEMU

The following example: creates a `canbus0`, connects CAN0 to `canbus0`, and connects the CAN0 bus (i.e., `canbus0`) to `vcan0` interface on the host device.

```
#ARM instance:
./qemu-system-aarch64 -M arm-generic-fdt \
-serial mon:stdio -serial /dev/null -display none\
-device loader,file=pre-built /linux/images/bl31.elf,cpu-num=0  \
-device loader,file=pre-built /linux/images/Image,addr=0x00080000\
-device loader,file=pre-built /linux/images/system.dtb,addr=0x1407f000\
-device loader,file=build /misc/linux-boot/linux-boot.elf\
-gdb tcp::9000 -net nic -net nic -net nic -net nic,vlan=1 -net
user,vlan=1,tftp=pre-built/linux/images/ \
-hw-dtb dts/LATEST/MULTI_ARCH/zcu102-arm.dtb -dtb pre-built/linux/images/
system.dtb \
-machine-path /tmp/tmp.ziqQHfo540 -global xlnx,zynqmp-boot.cpu-num=0 -
global xlnx,zynqmp-boot.use-pmu fw=true -m 4G \
-object can-bus,id=canbus0 \
-global driver=xlnx.zynqmp-can,property=canbus0,value=canbus0 \

#MicroBlaze instance is used same way.
```

Using Both CAN0 and CAN1 Devices with QEMU

The following example creates two separate can buses, `canbus0` and `canbus1`. It connects CAN0 to `canbus0` and CAN1 to `canbus1`, and then connects both CAN0 and CAN1 to the `vcan0` interface on the host device.

```
./qemu-system-aarch64 -M arm-generic-fdt \
-serial mon:stdio -serial /dev/null -display none\
-device loader,file=pre-built /linux/images /bl31.elf,cpu-num=0  \
-device loader,file=pre-built /linux/images/Image,addr=0x00080000 \
-device loader,file=pre-built /linux/images/system.dtb,addr=0x  1407f000\
-device loader,file=build /misc/linux-boot/linux-boot.elf\
-gdb tcp::9000 -net nic -net nic -net nic -net nic,vlan=1 -net
user,vlan=1,tftp=pre-built/linux/images/ \
-hw-dtb dts/LATEST/MULTI_ARCH/zcu102-arm.dtb -dtb pre-built/linux/images/
system.dtb \
-machine-path /tmp/tmp.ziqQHfo540 -global xlnx,zynqmp-boot.cpu-num=0 -
global xlnx,zynqmp-boot.use-pmu fw=true -m 4G \
-object can-bus,id=canbus0 \
```

```
-object can-bus,id=canbus 1 \
-global driver=xlnx.zynqmp-can,property=canbus0,value=canbus0 \
-global driver=xlnx.zynqmp-can,property=canbus1,value=canbus1 \
-object can-host-socketcan,id=socketcan0,if=vcan0,canbus=canbus0 \
-object can-host-socketcan,id=socketcan1,if=vcan0,canbus=canbus1
```

Dumping Random Data to CAN Through Virtual CAN Interface

The following command pumps random data to the `vcan0` interface, which is received by CAN0 and CAN1 if they are connected to the `vcan0` interface.

```
cangen - v vcan0
#use cangen -h to know all supported option with cangen utility
```

Analyzing Data on the Host CAN Interface

The host side CAN interface can be used to analyze CAN traffic with the `candump` command, included in `can-utils`. This shows any data sent from Xilinx CAN devices in QEMU.

```
candump vcan0
#use candump -h to know all use cases
```

Monitor Options

The monitor option specifies where to send the QEMU monitor. Generally, this is sent to the standard I/O which can be done with the following command:

```
-monitor chardev:terminal
```

Network Options

QEMU supports a range of detailed networking options. Some are covered here, but a more detailed list can be found at this link.

```
-net nic -net user,tftp=<directory>
```

This connects `GEM0` to a virtual network, with a TFTP server hosting the argument directory. The TFTP server IP is `10.0.2.2`. The guest software can configure the machine to an IP on the same subnet (for example `10.0.2.4`) and communicate. The following command connects GEM3 of the ZCU102 board to the user network:

```
-net nic -net nic -net nic -net user
```

See the QEMU public documentation for more comprehensive listing of QEMU networking options.

Port Redirection

Adding the following command makes every packet in/out of host-port to redirect to target-port of guest. For example, to get connected to guest using telnet, one can redirect the port 23 of guest to any free port of the host.

```
-redir tcp:<host-port>:<guest-ip>:<target-port>
```

Example

```
-redir tcp:1440:10.0.2.15:23
```

Tap Mode (Requires Sudo)

Tap network is similar to bridge network; it allows the guest to communicate with host DHCP and DNS. This makes it also a real device on a network.

Unlike user mode, in tap mode, networking guest is directly accessible without any port redirection/forwarding. The following is an example:

```
-net nic -net nic -net nic -net nic -net tap,downscript=no
```

Before running QEMU ensure that you have the `qemu-ifup` script available in `/etc/`. Follow the instructions in this link to set up tap network.

Debug Options

The best option is a multi-arch GDB. This is usually available through your distributor. Following are the QEMU arguments.

Enabling and using the GDB stub:

```
-gdb tcp:<host_name>:<port> -S
```

Where:

- `-gdb`: Creates a GDB stub on the local host at the specified port.
- `-S`: Causes the emulation to start in the pause state. This allows you to attach a debugger before software starts executing. You can attach your GDB to QEMU as follows: `(gdb) target remote :<port>`

On the GDB host, use the GDB that corresponds to your build toolchain; for example:

- `aarch64-none-elf-gdb or aarch64-linux-gnu-gdb` for debugging A53 code.

Or:

- `arm-none-eabi-gdb` for R5 software.

QEMU emulation can be resumed using a continue command in the GDB as shown below:

```
(gdb) c
```

Breakpoints can be inserted as normal. Either symbolic function names, file lines, or text memory addresses can be used. See the Arm Information Center for more information.

Debug-Related Monitor Commands

You can use a range of QEMU monitor commands to access helpful debug information and perform some basic operations. The following are a few of the more commonly used options. See Non-Graphical I/O Option for information on accessing the monitor.

Stopping and resuming the VM:

```
(qemu) stop
        (qemu) c
```

These commands stop and resume the emulation, respectively. If QEMU is started with the `-S` argument, you can use the `c` command to commence emulation.

Display Options

The QEMU display option emulates a virtual monitor for the display applications.

IMPORTANT! *To make use of the Display option, do not pass the `-nographic` argument in the command line; it restricts the ability to create a display console.*

CAUTION! *PetaLinux QEMU does Not include SDL support for Display Monitor Emulation, It is recommended that you build QEMU from source with SDL enabled. See Building QEMU from source.*

The following command option, when passed on the QEMU command line, creates a VNC session through which you can view the display console:

```
-vnc <hostname>:<display>
```

For example, the command `-vnc localhost:1` connects to a VNC session using VNC viewer on `localhost:1`, where `1` represents the display ID. In this way you can have n number of display monitors open.

Using TightVNC and RealVNC

In both TightVNC and RealVNC you can open the monitor using following command:

```
vncviewer localhost:<display id>
```

To open monitor for a different server, enable port forwarding using the following command:

```
ssh <target-host> -L <localhost-port>:localhost:<target-host-port>
```

For example, if the QEMU display id is 1, it should map to TCP port 5901. Then, run the port forwarding command as follows:

```
ssh qemu-host -L 5901:localhost:5901
```

Now use the same vncviewer command to open the display.

***Note*:** RealVNC does not directly work. Follow the workaround in the following link.

Listening on a VNC Session

You can connect to a display at any time using the `-vnc <hostname>:<display>` command; however, if you need the QEMU to wait until it is connected to a VNC session, configure the QEMU to work with a listening VNC session.

```
-vnc <hostname>:<tcp port number>,reverse
```

For example, `-vnc localhost:5501,reverse`, will be able to connect to a listening VNC session with `display-id 1`.

***Note*:** The TCP port number for display ID 1 maps as `5500+d`, where `d` is the display ID.

Running the VNC viewer with `-listen 1` results in the host listening for connections on the VNC session with display 1.

Listing and Selecting CPUs in the System

From the QEMU Monitor Console, use the following commands:

```
(qemu) info cpus
        (qemu) cpu <value>
```

Example for Zynq UltraScale+ MPSoC:

- The `info cpus` lists out the CPUs in the system and indicated the currently selected CPU. In a normal setup there are six CPUs: 4 x A53 and 2 x R5. QEMU has the concept of a currently selected CPU with respect to some monitor commands. This is very similar to GDBs concept of a currently selected thread.
- Change the selected CPU using the `cpu <value>` command; where `<value>` is a number from 0 to 5 that corresponds to the QEMU CPU indexes.
- CPUs with an asterisk (*) are currently selected.

See CPU Enumeration for information on how QEMU CPU indexes match the Zynq UltraScale+ MPSoC platform CPUs.

Inspecting CPU State

```
(qemu) info registers
```

The info registers command dumps out useful information about the current CPU (such as registers and current EL). For R5 CPUs, R15 is the program counter. See Arm Architecture Reference Manual for more information.

Inspecting Physical Memory

```
(qemu) ( xp | x ) <addr>

(qemu) memsave <addr> <length> <file>
```

Use these commands to dump memory data. The `x` or `xp` command can be used to read a single address:

- `xp: (qemu) xp 0x1234F00D`
  - `x` uses virtual addresses for the currently selected cpu
  - `xp` uses physical addresses
- The `memsave` command saves a buffer of specified length (`<length>`) and address (`<addr>`) to a file specified by `<file>`, where `<file>` is the data buffer as raw binary data.

For example: `(qemu) memsave 0xc0ae1a80 16384 dumpmem.logbuf`

Linux Kernel Logbuf Extraction

The following instructions are directly applicable to booting Linux Zynq® UltraScale+™ MPSoC QEMU. There is more information available at Linux Kernel Logbuf Extraction.

Multi-Threaded Tiny Code Generator (MTTCG)

The Multi-Threaded Tiny Code Generator runs each vcpu on an individual host cpu thread, this enhances performance when compared to the single threaded model. Xilinx® QEMU is now enabled with the MTTCG by default. It is available in Arm® SoC emulation. You need not pass any additional arguments. To control MTTCG, use the following arguments:

- To enable MTTCG:

  ```
  accel tcg,thread=multi
  ```

- To disable MTTCG:

  ```
  accel tcg,thread=single
  ```

Note: MTTCG is not compatible with icount. Enabling icount will force a single threaded run. See QEMU Wiki for more information.

Examples for Single Arch

This section provides few examples of elaborated QEMU command lines. Substitute particular arguments to suit your application as needed. See the QEMU Command Line Reference Manual for further details. For each command, it is assumed that the current working directory is a PetaLinux project root.

You can run the commands without a PetaLinux project but file paths of boot components will need to be adjusted. See Storage Media as required for details on how to generate storage media files for QSPI, NAND, and SD.

FSBL as an Application on A53-0

In each of the following examples, first stage boot loader (FSBL) is run as the application; however, these command line formats are applicable to other standalone software. Substituting the FSBL ELF file for another standalone application is valid.

A53-0 FSBL in JTAG Mode

```
qemu-system-aarch64 -M arm-generic-fdt -nographic \
-dtb ./images/linux/zynqmp-qemu-arm.dtb \
-device loader,file=./images/linux/zynqmp_a53_fsbl.elf,cpu-num=0 \
-device loader,addr=0xfd1a0104,data=0x8000000e,data-len=4
```

A53-0 FSBL in QSPI Boot Mode (Single)

```
qemu-system-aarch64 -M arm-generic-fdt -nographic \
-dtb ./images/linux/zynqmp-qemu-arm.dtb \
-device loader,file=./images/linux/zynqmp_a53_fsbl.elf,cpu-num=0 \
-device loader,addr=0xfd1a0104,data=0x8000000e,data-len=4 \
-drive file=qemu_qspi.bin,if=mtd,format=raw,index=0\
-boot mode=1
```

A53-0 FSBL in QSPI Boot Mode (Dual Parallel)

```
qemu-system-aarch64 -M arm-generic-fdt -nographic \
-dtb ./images/linux/zynqmp-qemu-arm.dtb \
-device loader,file=./images/linux/zynqmp_a53_fsbl.elf,cpu-num=0 \
-device loader,addr=0xfd1a0104,data=0x8000000e,data-len=4 \
-drive file=qemu_qspi_low.bin,if=mtd,format=raw,index=0\
-drive file=qemu_qspi_high.bin,if=mtd,format=raw,index=1\
-boot mode=1
```

Note: Default ZCU102 PetaLinux design works as Dual Parallel Configuration.

A53-0 FSBL in SD Boot Mode

```
qemu-system-aarch64 -M arm-generic-fdt -nographic \
-dtb ./images/linux/zynqmp-qemu-arm.dtb \
-device loader,file=./images/linux/zynqmp_a53_fsbl.elf,cpu-num=0 \
-device loader,addr=0xfd1a0104,data=0x8000000e,data-len=4 \
-drive file=qemu_sd.img,if=sd,format=raw,index=0\
-boot mode=3
```

***Note*:** Default ZCU102 board supports SD1. Index should be set to 1 for sd drive argument.

FSBL as an Application on R5-0

In each of the following examples, FSBL is run as the application. These command line format are however applicable to other standalone software. Substituting the FSBL ELF file for another standalone application is valid.

R5-0 FSBL in JTAG Mode

```
qemu-system-aarch64 -M arm-generic-fdt -nographic \
-dtb ./images/linux/zynqmp-qemu-arm.dtb \
-device loader,file=zynqmp_r5_fsbl.elf,cpu-num=4 \
-device loader,addr=0xff5e023c,data=0x80008fde,data-len=4 \
-device loader,addr=0xff9a0000,data=0x80000218,data-len=4
```

R5-0 FSBL in QSPI Boot Mode (Single)

```
qemu-system-aarch64 -M arm-generic-fdt -nographic \
-dtb ./images/linux/zynqmp-qemu-arm.dtb \
-device loader,file=zynqmp_r5_fsbl.elf,cpu-num=4 \
-device loader,addr=0xff5e023c,data=0x80008fde,data-len=4 \
-device loader,addr=0xff9a0000,data=0x80000218,data-len=4 \
-drive file=qemu_qspi.bin,if=mtd,format=raw,index=0\
-boot mode=1
```

R5-0 FSBL in QSPI Boot Mode (Dual Parallel)

```
qemu-system-aarch64 -M arm-generic-fdt -nographic \
-dtb ./images/linux/zynqmp-qemu-arm.dtb \
-device loader,file=zynqmp_r5_fsbl.elf,cpu-num=4 \
-device loader,addr=0xff5e023c,data=0x80008fde,data-len=4 \
-device loader,addr=0xff9a0000,data=0x80000218,data-len=4 \
-drive file=qemu_qspi_low.bin,if=mtd,format=raw,index=0 \
-drive file=qemu_qspi_high.bin,if=mtd,format=raw,index=1
-boot mode=1
```

R5-0 FSBL in SD Boot Mode

```
qemu-system-aarch64 -M arm-generic-fdt -nographic \
-dtb ./images/linux/zynqmp-qemu-arm.dtb \
-device loader,file=zynqmp_r5_fsbl.elf,cpu-num=4 \
-device loader,addr=0xff5e023c,data=0x80008fde,data-len=4 \
-device loader,addr=0xff9a0000,data=0x80000218,data-len=4 \
-drive file= qemu_sd.img,if=sd,format=raw,index=0 \
-boot mode=3
```

R5 Lockstep FSBL

Only one example is provided for lock step, although all boot modes are valid. See the previous example command line arguments for storage media and boot mode that could be applied to this command line. This specific example is JTAG boot mode:

```
qemu-system-aarch64 -M arm-generic-fdt -nographic \
-dtb ./images/linux/zynqmp-qemu-arm.dtb \
-device loader,file=zynqmp_r5_fsbl.elf,cpu=4 \
-device loader,addr=0xff5e023c,data=0x80008fde,data \
-len=4
```

Running PMU and Arm QEMU

The following examples run two instances of QEMU. One emulating the PS part of Cortex-A53s and Cortex-R5Fs and another emulating the PMU in a MicroBlaze™ device.

Note: DTBs used for single and present multiarch model are different.

1. Set up the Arm® instance by typing the following commands:

   ```
   qemu-system-aarch64 -M arm-generic-fdt -nographic \
   -dtb <proj>/pre-built/linux/images/zynqmp-qemu-multiarch-arm.dtb \
   -device loader,file=<proj>/pre-built/linux/images/bl31.elf,cpu-num=0 \
   -device loader,file=<proj>/pre-built/linux/images/u-boot.elf \
   -global xlnx,zynqmp-boot.cpu-num=0 \
   -global xlnx,zynqmp-boot.use-pmufw=true \
   -machine-path <any-folder-path> -gdb tcp::9000
   ```

   Note: When booting from fsbl, loading of static pmufw config can be disabled by appending the following arguments: `-global xlnx,zynqmp-boot.load-pmufw-cfg=false`

2. Set up a PMU instance by typing the following commands:

   ```
   qemu-system-microblazeel -M microblaze-fdt -nographic \
   -dtb <proj>/pre-built/linux/images/zynqmp-qemu-multiarch-pmu.dtb \
   -kernel <proj>/pre-built/linux/images/pmu_rom_qemu_sha3.elf \
   -device loader,file=<proj>/pre-built/linux/images/pmufw.elf \
   -machine-path <any-folder-path> -gdb tcp::9005
   ```

You can connect to Arm QEMU using the XSDB `connect` command. See Building QEMU from Source for more information.

Building QEMU from Source

QEMU source code is available on the github link. The following are the build steps:

1. Clone QEMU and update the sub-modules `dtc` and `pixman`.

IMPORTANT! *Make sure to install the build dependencies before starting the build.*

   For Ubuntu, use `apt-get build-dep qemu`. It is recommended to do an out-of-tree build.

2. Make an empty folder outside of the `source` folder, and change into the new folder.
3. Run the following `configure` command:

```
<QEMU_SOURCE_PATH>/configure
--target-list=”aarch64-softmmu,microblazeel-softmmu,arm-softmmu" \
--enable-debug \
--enable-fdt \
--disable-kvm
```

4. Run `make -j16`.

   QEMU binaries are available at:

   - `<build_path>/aarch64-softmmu/qemu-system-aarch64`
   - `<build_path>/microblazeel-softmmu/qemu-system-microblazeel`

   You might need to install additional libraries based on your configuration inputs.

Using XDB with QEMU

Connecting QEMU

XSDB connects to QEMU GDB by using remote ports. While passing arguments to load and run QEMU, attach a GDB client by appending the option: `-gdb tcp::<port-num>` to QEMU command line. The following is the command:

```
qemu-system-aarch64 -M arm-generic-fdt -nographic \
-dtb ./images/linux/zynqmp-qemu-arm.dtb -gdb tcp::1440
```

***Note*:** Port number is not specific; it can be any free port.

With QEMU running, connect the QEMU using the XSDB `gdbremote` command shown in the following command line:

```
gdbremote connect <hostname>:<port-num>
```

***Note*:** The hostname can be localhost or the name of the server or an IP address upon which

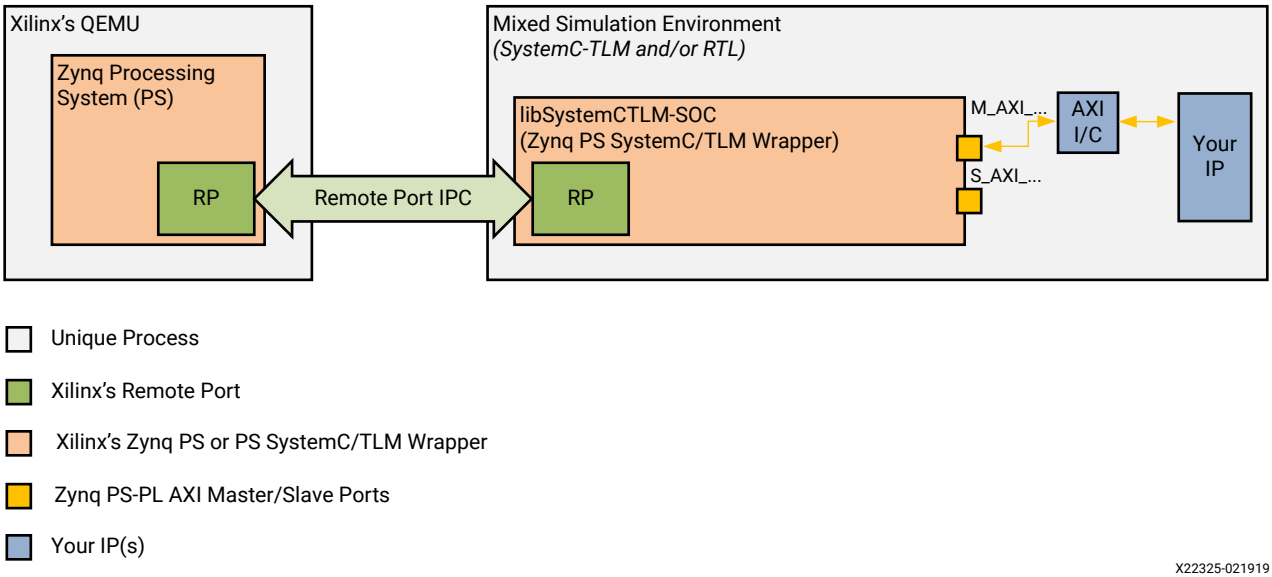
Table 11: **Processors Out of Reset Command**

Processor	Command and Address
Cortex™-A53	`mwr 0xfd1a0104 0`
Cortex-R5F	`mwr 0xff5e023c 0x00008fde` `mwr 0xff9a0000 0x00000218`

Co-Simulating with QEMU

You can use the Xilinx® QEMU to connect and drive mixed simulation environments. This feature enables you to model large and complex systems right from the beginning of the system design. As of 2016.4 release, Xilinx exposes a SystemC/TLM interface to connect QEMU which models the hardened Processing System (PS) of any Zynq® based product to a model of your own IP written in Verilog/VHDL or SystemC instantiated in the Programmable Logic (PL), see the following figure.

Figure 2: **Xilinx QEMU Mixed Simulation Environment**



QEMU and the RTL or SystemC Simulator run on different processes enabling a less intrusive and much more flexible integration between your existing mixed simulation environment and QEMU.

Note: This feature is predominantly suitable for experienced developers in SystemC/TLM and integration with Mixed Simulation environments. This feature is provided "as-is" and under open source license model. Feel free to use our libSystemCTLM-SOC to interface your simulation environment to Xilinx QEMU. Please see the SystemC page of Accellera's website for further details and demo with Accellera's Open Source SystemC Reference Simulation Environment (Accellera's SystemC Reference Simulation Environment is free and under the Apache v2 License as of 2016).

Remote-Port

The underlying mechanism that QEMU uses to connect to external simulation environments is through remote-port (RP). Remote-Port is a protocol/framework that uses sockets and shared-memory to communicate transactions and synchronize time between simulators.

libsystemctlm-soc

You do not need to interface directly with Remote-Port. Xilinx provides abstractions for SystemC/TLM-2.0 that encapsulate your SystemC/TLM-2.0 module, allowing it to connect to remote QEMU instances. These modules use Remote-Port. SystemC/TLM-2.0 users can, therefore, treat QEMU as any other standard SystemC/TLM-2.0 module. These abstractions are in the libsystemctlm-soc repository. For an example of wrapping your SystemC application, see the SystemC/TLM-2.0 co-simulation demonstration.

SystemC/TLM-2.0 Co-Simulation Demo

This demo is written using standard, compliant SystemC/TLM-2.0 APIs. You can run the demo on any SystemC/TLM-2.0 simulator that is compliant with Accellera Systems Initiative (ASI) industry standard specifications. This open source reference implementation of the simulator is tested and verified with Accellera's standard.

The SystemC/TLM-2.0 co-simulation demonstration provides an example project that demonstrates how to use `libsystemctlm-soc` to connect custom SystemC/TLM-2.0 and RTL models to QEMU.

The examples in this demonstration show how QEMU models the PS aspect of both Zynq®-7000 devices and Zynq UltraScale+™ devices while SystemC/TLM-2.0 and RTL models can be used to model the custom PL logic.

Co-Simulating with QEMU

Generating Required Device Trees

You need to instruct QEMU to co-simulate with other simulation environments. This can be done by editing the hardware device tree passed into QEMU using the `-hw-dtb` option.

Note: The hardware device tree is specific to QEMU and should not be confused with Linux guest device tree.

For more information, see QEMU Wiki and device tree repository.

Extra Command Line Options

`-machine-path`: Specifies a directory where QEMU creates shared memory files and named UNIX sockets.

`-sync-quantum`: Specifies the TLM synchronization quantum in nanoseconds (only used if `-icount` is enabled).

`-icount`: Enables virtual instruction counter with `2^N` clock ticks per instruction; enables aligning the host and virtual clocks or disables real-time CPU sleeping.

Useful starting values for `icount`:

- Zynq UltraScale+ MPSoC devices: 1
- Zynq-7000 devices: 7

As you lower the `sync-quantum`, the modeling speed decreases, but the accuracy increases. A general starting value is 100000.

Example Extra Options

```
-icount 1 -sync-quantum 100000
```

Example QEMU Command

```
qemu-system-aarch64 -M arm-generic-fdt -nographic \
   -dtb <path to dtb build form qemu-devicetrees repo> \
   -device loader,file=<proj_dir>images/linux/bl31.elf,cpu=0 \
   -device loader,addr=0xfd1a0104,data=0x8000000e,data-len=4 \
   -device loader,file=<proj_dir>/images/linux/u-boot..elf \
   -tftp images/linux/ -machine-path <soc_dir> -icount 1 \
   -sync-quantum 100000
```

Note: Use the same sync-quantum number for the other simulators.

Example Simulator Command

When the QEMU command shown above runs successfully, QEMU waits for SystemC/TLM-2.0 connection on the socket created in the directory that was supplied by the `-machine-path argument`. You must use the same socket path while running the SystemC application, as follows:

```
./demo-app unix:<socket path> <sync-quantum number>
```

More details on how to build and run the SystemC demo application are in systemctlm-cosim-demo repository.

Using Boot Images on QEMU

This section details some end-to-end image generation and QEMU boot flows. The standard FSBL, Arm® Trusted firmware (ATF), U-Boot, and Linux boot flow is the example in each case. This specific use case is similar to PetaLinux, and you can access it more simply using the PetaLinux tool suite. This section details the lower-level tools available for complex boot flows should they be required for greater customization. This section does not cover building the boot products. See the *Zynq UltraScale+ MPSoC: Software Developers Guide* (UG1137) for information on how to build the following:

- FSBL
- U-Boot
- ATF
- Linux image (Kernel plus RAM disk)
- Device tree binary
- BOOT.bin

Note: This is different from the hardware DTB that is passed to QEMU command lines.

It is assumed all of these boot products are available. You can build all of the listed images in a standard PetaLinux project. In the example, FSBL, ATF `(bl31.elf)`, U-Boot runs on Cortex A53 core as shown in the BIF file. U-Boot loads the Kernel onto A53. The following run commands point to the `<proj_dir>/images/linux/` folder for all the boot products. You can also use a pre-built area.

Using SD for Boot

1. Create the SD image:

```
dd if=/dev/zero of=qemu_sd.img bs=128M count=1
mkfs.vfat -F 32 qemu_sd.img
mcopy -i qemu_sd.img BOOT.BIN ::/
mcopy -i qemu_sd.img Image ::/
mcopy -i qemu_sd.img system.dtb ::/
```

2. Boot the image on QEMU:

```
qemu-system-aarch64 -M arm-generic-fdt -nographic \
-device loader,file=<proj_dir>/images/linux/ron_a53_fsbl.elf,cpu-num=0 \
-dtb <proj_dir>/images/linux/zynqmp-qemu-arm.dtb \
-device loader,addr=0xfd1a0104,data=0x8000000e,data-len=4 \
-drive file=qemu_sd.img,if=sd,format=raw,index=0\
-boot mode=3
```

Note: Even though the FSBL is packed in the SD image, it should be passed over the command line as a runnable ELF because QEMU does not contain the boot ROM.

Using QSPI for Boot

1. Create the QSPI boot image(s) for either or both single flash mode and dual parallel mode.

   Single Flash Mode:

```
dd if=/dev/zero of=qemu_qspi.bin bs=64M count=1
dd if=BOOT.BIN of=qemu_qspi.bin bs=1 seek=0 conv=notrunc
```

   Dual Parallel Mode:

```
dd if=/dev/zero bs=128M count=1 of=qemu_qspi_tmp.bin
dd if=BOOT.BIN of=qemu_qspi_tmp.bin bs=1 seek=0 conv=notrunc
flash_strip_bw qemu_qspi_tmp.bin qemu_qspi_low.bin qemu_qspi_high.bin
```

2. Boot either the single or dual image(s) on QEMU.

   Single Flash Mode:

```
qemu-system-aarch64 -M arm-generic-fdt -nographic \
-dtb <proj_dir>/images/linux/zynqmp-qemu-arm.dtb \
-device loader,file=<proj_dir>/ images/linux/zynqmp_a53_fsbl.elf,cpu-
num=0 \
-device loader,addr=0xfd1a0104,data=0x8000000e,data-len=4 \
-drive file=qemu_qspi.bin,if=mtd,format=raw,index=0 \
-boot mode=1
```

   Dual Parallel Mode:

```
qemu-system-aarch64 -M arm-generic-fdt -nographic \
-dtb <proj_dir>/images/linux/zynqmp-qemu-arm.dtb \
-device loader,file=<proj_dir>/images/linux/zynqmp_a53_fsbl.elf,cpu=0 \
-device loader,addr=0xfd1a0104,data=0x8000000e,data-len=4 \
-drive file=qemu_qspi_low.bin,if=mtd,format=raw,index=0\
-drive file=qemu_qspi_high.bin,if=mtd,format=raw,index=1
-boot mode=1
```

TIP: *See the Storage Media for more information on* `flash_strip_bw`.

Using TFTP for Boot

In the normal JTAG boot mode, `petalinux-build` command has the required images in the `images/linux/` directory (or) a prebuilt path (for example: `pre-built/linux/images/`).

1. Point QEMU to the `./images/linux` directory for `tftp` boot.
2. Use the following command for `TFTP` boot.

```
qemu-system-aarch64 -M arm-generic-fdt -nographic \
-dtb <proj_dir>/images/linux/zynqmp-qemu-arm.dtb \
-device loader,file=<proj_dir>images/linux/bl31.elf,cpu-num=0 \
-device loader,addr=0xfd1a0104,data=0x8000000e,data-len=4 \
-device loader,file=<proj_dir>/images/linux/u-boot..elf \
-net nic -net nic -net nic -net nic \
-net user,tftp=images/linux/
```

IMPORTANT! *The `/TFTP` folder should contain the image and the `system.dtb`.*

TIP: *The image creation steps are the same for SD and EMMC .*

SD-Card Partitioning and Loading an Ubuntu-core File System

The following are the steps to create an SD card image.

1. Create a dummy container using `qemu-img`. The `qemu-img` is a utility used to create disk images for using with QEMU. It comes with Ubuntu packages and is also packaged with PetaLinux tools.

   ```
   qemu-img create <Image name> <size>
   ```

   For example, the command, `qemu-img create sd.img 2G`, creates a 2G raw disk, with no partitions present.

2. Create the network back end with `qemu-nbd`.

   The `qemu-nbd` command connects `sd.imge` file to NBD device.

   For example, `sudo qemu-nbd -c /dev/nbd0 sd.img` makes the `sd.img` connect to a NBD device.

   TIP: *If `/dev/nbd0` is not found in the machine, `nbd` driver is probably not installed.*

3. Create partitions using `fdisk`, a text-based tool used to create partitions on a disk.

You can also use `gparted`, a GUI-based partitioning tool.

`fdisk /dev/nbd0` connects to the block device.

4. Create partitions, mostly two primary partitions are required.
   - One is bootable partition for keeping `BOOT.BIN`,`Image`, and the `system.dtb`.
   - Another is the partition for rootfs.
5. Write the partition table and exit.
6. Format the partitions. Always format the bootable partition using FAT file systems. The second partition can be `ext2/ext4`.

   For example:

   `mkfs.vfat -F 32 /dev/nbd0p1` formats the first partitions using FAT.

   `mkfs.ext4 /dev/nb0p2` formats the second partition using `ext4`.
7. Mount the partitions and copy the necessary files. To load the `Image` file without `initramfs`, de-select **[]Initial RAM filesystem and RAM disk (initramfs/initrd) support** located in General Setup in menuconfig.

   ***Note*:** This step in not required if performing switch-root.
8. Extract the `ubutu-core` available for Arm64 in to the second partition.

TIP: *The `ubuntu-core` is located on the `Ubuntu-Core` release page.*

9. Un-mount the partition and disconnect the `nbd` connection using the following command: `sudo qemu-nbd -d /dev/nbd0`.
10. Ensure that the `bootargs` points to appropriate device for root, which is the following: `root=/dev/mmcblk0p2 rw rootfstype=ext4`.

Adding New Devices to the Design

QEMU has limited device models; you can use the device model if it is available in QEMU source by directly adding the bindings into the device tree.

The following is an example of adding a SI57X I2c controlled clock generator to one of the I2C buses:

```
&i2cswitch{
i2c@3 {
            #address-cells = <1>;
            #size-cells = <0>;
            reg = <3>;
            si570_21 clock-generator@5e {
               compatible = “silabs,si57x”;
```

```
                reg = <0x5d>;
                temperature-stability=<50>;
            };
        };
        };
```

The compatible string is QEMU device, `TYPE_NAME`.

- `reg` represents the I2C address of the slave.
- `temperature-stability` is the property that the device expects to be set before device initialization happens.

See the `Property` structure in the device model to know what properties must be added.

```
static Property si57x_properties[] = {
    DEFINE_PROP_UINT16("temperature-stability", Si57xState, temp_stab,
      TEMP_STAB_50PPM),
   DEFINE_PROP_END_OF_LIST(),
```

Troubleshooting

FSBL Hangs on QEMU

First stage boot loader (FSBL) uses `psu-init.c`, which is dynamically generated code, and changes it according to the design. `psu_init` functions generally make clock configurations for the SoC, which QEMU does not emulate. Due to such missing emulation, sometimes `psu_init` calls may hang during `fsbl` boot.

Possible Solution: Build a customized `fsbl` by commenting out the functions that cause the hangs in `psu_init.c`.

For additional information on possible solutions, see the Xilinx Developer Forum .

QEMU CPU Stall Messages

When running Linux on top of QEMU, Linux sometimes warns about CPU stalls. Following are the stalls and their causes:

- vCPU in QEMU has hung.

  ***Note*:** This is a bug in QEMU, please report it to your Xilinx representative if you encounter this.

- vCPU in QEMU has not been scheduled for enough time and Linux thinks it has hung, when in fact it has not.

  ***Note*:** This may cause due to multiple reasons related to vCPUs not getting scheduled to run. For example, due to an overloaded host, or low priority scheduling assignments to QEMU.

Possible Solutions to Avoid Scheduling Issues

- If possible, avoid running QEMU in a Linux VM like Virtualbox. VMs are subject to CPU scheduling by the host (for example Windows host). This adds more scheduling latency, increasing the probability of a vCPU stall.

- Avoid running a lot of other CPU intensive programs while running QEMU. These other programs use up CPU resources and increase the scheduling latency and the probability of vCPU stalls.
- Avoid giving QEMU a low scheduling priority (e.g., nice QEMU).
- Disable the RCU CPU stalls check in the guest Linux kernel.
- Disable SMP in the guest to decrease the amount of CPU processing needed by QEMU.
- Try using an MTTCG enabled QEMU to avoid all CPUs depending on one single host thread.

XILINX

Appendix A

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help → Documentation and Tutorials**.
- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

Note: For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

References

These documents provide supplemental material useful with this guide:

Zynq Documentation:

1. *Zynq UltraScale+ MPSoC: Software Developers Guide* (UG1137)
2. *SDK Online Help* (UG782) (Includes XSDB)
3. *OS and Libraries Document Collection* (UG643)
4. *Xilinx Third-Party End User License Agreement*
5. *UltraScale Architecture and Product Data Sheet: Overview* (DS890)
6. *Zynq UltraScale+ Device Technical Reference Manual* (UG1085)
7. *Zynq UltraScale+ MPSoC Register Reference* (UG1087)
8. PetaLinux Tools
9. Vivado Design Suite Documentation

Wiki Sites and GitHub Resources

1. QEMU wiki
2. Zynq MPSoC XEN wiki
3. GNU FTP
4. Zynq MPSoC Non-Secure Boot and Zynq MPSoC Secure Boot
5. Arm Information Center
6. Using Git
7. GitHub
8. OpenAMP wiki
9. QEMU Linux Kernel Logbuf Extraction
10. libsystemctlm-soc Repository
11. SystemC/TLM 2.0 Co-simulation Demo Repository
12. Device Tree Repository

Please Read: Important Legal Notices

negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright