# Xilinx Quick Emulator User Guide

## QEMU

November 3, 2020

# Table of Contents

User the search bar below to quickly find items of interest in this guide:

The selected root page could not be found.

Search

For our new users we would suggest starting from Chapter 1 - Introduction to QEMU.

- Chapter 1 - Introduction to QEMU
- Chapter 2 - Building and Running QEMU
- Chapter 3 - Developing with QEMU Virtual Machines
- Chapter 4 - Advanced
- Chapter 5 - Troubleshooting and Known Issues
- Chapter 6 - Additional Resources and References

# 1 Versions

| Release |
| --- |
| latest |
| 2020.2 |
| 2020.1 |
| 2019.2 |

# 2 What is QEMU

This page gives a bird eye view of what QEMU is, why should you use it and how it works at a high level.

This is an ideal starting point for new QEMU users.



## 2.1 What is QEMU

QEMU (Quick EMUlator) is an open source, cross-platform, system emulator. It is an executable that runs on an x86 Linux or Windows operating systems.

QEMU can emulate a full system (commonly referred to as the guest), such as a Xilinx ZCU102 or VCK190 board.

> When this document uses examples that are ran on the guest, the example shell prompt will say something similar to `xilinx-zcu102-2020_2:~#` or `xilinx-vck190-2020_2:~#`.

The emulation includes the processors, peripherals, and other hardware on the development board; allowing you to launch an operating system or other applications on the virtualized hardware.

These applications can be developed using the exact same toolchains that would be used on physical hardware.

QEMU can also interact with the host machine through interfaces, such as CAN, Ethernet and USB; allowing real-world data from the host to be used in the guest machine in real time.

## 2.2 Why Use QEMU

### 2.2.1 Remote Development

Using QEMU allows developers to develop without the need for physical hardware, while still being able to use real-world data for testing.

### 2.2.2 Easier Debugging

QEMU contains a GDB server which the host can connect to and debug their applications through, as if they were running natively.
See Debugging Guest Applications with QEMU and GDB for more information on how to debug using QEMU.

QEMU also contains commands and tools, such as module debug printing, that developers can use to debug their programs.

Since QEMU emulates the entire guest platform, this allows debugging from the bootloader up to the user application. This also means that emulation can be paused and resumed at any time, even in the middle of a data transfer.

### 2.2.3 Easier Testing

Since QEMU is entirely software, it integrates much easier with testing utilities (e.g. PyTest) than hardware does.

### 2.2.4 Developing and Running an OS

QEMU allows people to run systems that use Linux, FreeRTOS, a custom OS, or bare metal applications.
When doing system-level programming with QEMU, it is easy for developers to cross-compile their kernel and run it in QEMU for testing.

See What is QEMU for how to get started with QEMU.

### 2.2.5 Hardware Modeling and Verification

QEMU can be used to model and verify hardware, but it can also connect and drive mixed simulation environments, called co-simulation.
With co-simulation, Xilinx exposes a SystemC/TLM interface to connect the processing system (PS) of any Zynq-based and Versal products to a model of your own IP.

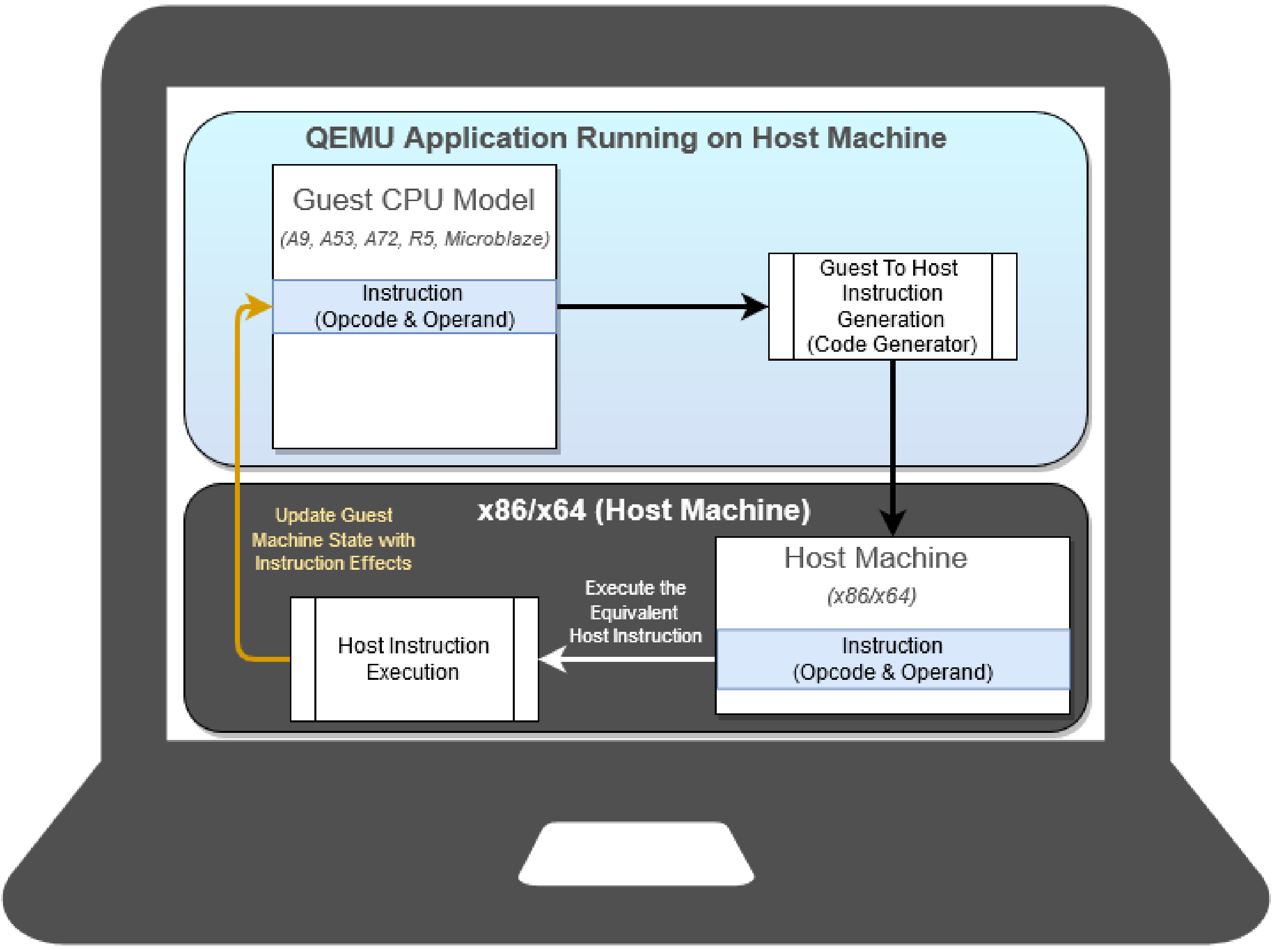
### 2.2.6 Safety and Security

With QEMU, you have access to non-intrusive fault injection, allowing you to validate the safety, security, reliability, and robustness of your system.

## 2.3 How it Works

QEMU works by using dynamic translation. Instructions are translated from the guest's instruction set to the equivalent host machine instructions.

The equivalent host instructions are then executed on the host, and the results of those instructions are then pushed back into the guest machine.
A diagram of this is shown below.




Since the emulation is working at an instruction level, your software is running exactly as it would run on the final target hardware platform; completely unchanged.

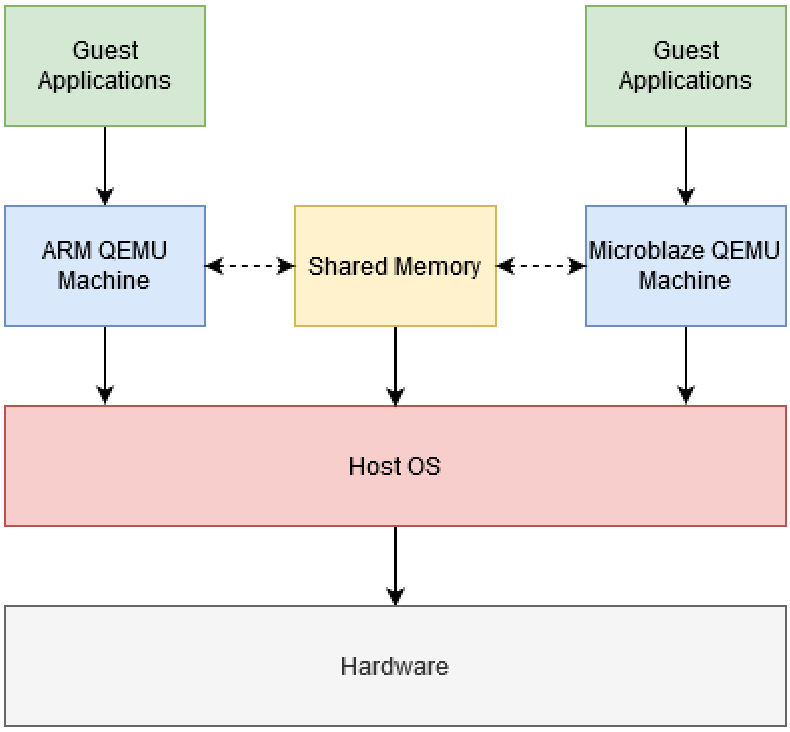
This is one of the key benefits of running your software on QEMU, you have complete instruction parity between the final target hardware platform and QEMU.

In order to take full advantage of the host execution throughput, instructions are executed as fast as the host will allow.  As such, QEMU is not a cycle accurate emulator.

In addition, due to some key innovations from Xilinx, QEMU can be connected and interoperate with clock cycle-accurate emulation environments to drive Verilog/SystemC based IP. See the Co-simulation page for more information.

In multi-architecture environments, such as Zynq UltraScale+ MPSoC/RFSoC and Versal ACAP, QEMU uses shared memory to communicate between the different architectures (i.e. ARM and MicroBlaze).

For example, a block diagram of a Zynq UltraScale+ MPSoC/RFSoC or Versal ACAP machine running on QEMU may look like this:




Note that if running a standalone (single-architecture environment) ARM or MicroBlaze machine on Zynq UltraScale+ MPSoC/RFSoC or Versal ACAP, the shared memory is not necessary, as only one QEMU guest is running. This is also true if running any other standalone ARM or MicroBlaze platform, such as Zynq-7000 or MicroBlaze.

## 2.4 QEMU and Xilinx

Xilinx provides a QEMU emulation platform to support the software developers targeting MicroBlaze, Zynq-7000, Zynq UltraScale+ MPSoC/RFSoC, and Versal ACAP development platforms.

Xilinx QEMU is distributed as part of the Petalinux and Yocto toolchains and is already integrated in Vitis.

QEMU can emulate Xilinx development boards, such as VCU118, AC701, ZCU102, VCK190, etc. See QEMU Supported Platforms for a full list of what Xilinx supports on QEMU.

## 2.5 Next Steps

See Chapter 2 on how to Build and Run QEMU.

# 3 QEMU Supported Platforms

This section covers what Xilinx boards and peripherals are implemented in QEMU.



## 3.1 Supported Boards

The following table shows what boards are supported by QEMU, and the names of the BSPs in PetaLinux and Yocto.

| Platform | Board Name | PetaLinux-specific Board Name | Yocto-specific board name |
|---|---|---|---|
| Artix-7 w/ Microblaze | AC701 | xilinx-ac701 | |
| Kintex-7 w/ Microblaze | KC705 | xilinx-kc705 | |
| Kintex-7 w/ Microblaze | KCU105 | xilinx-kcu105 | |
| Spartan-7 w/ Microblaze | SP701 | xilinx-sp701 | |
| Virtex Ultrascale+ w/ Microblaze | VCU118 | xilinx-vcu118 | |
| Zynq7000 SoC | ZC702 | xilinx-zc702 | zc702-zynq7 |
| Zynq7000 SoC | ZC706 | xilinx-zc706 | zc706-zynq7 |
| Zynq Ultrascale+ | ZC1751 | xilinx-zc1751 | |
| Zynq Ultrascale+ | ZCU102 | xilinx-zcu102 | zcu102-zynqmp |
| Zynq Ultrascale+ | ZCU104 | xilinx-zcu104 | zcu104-zynqmp |

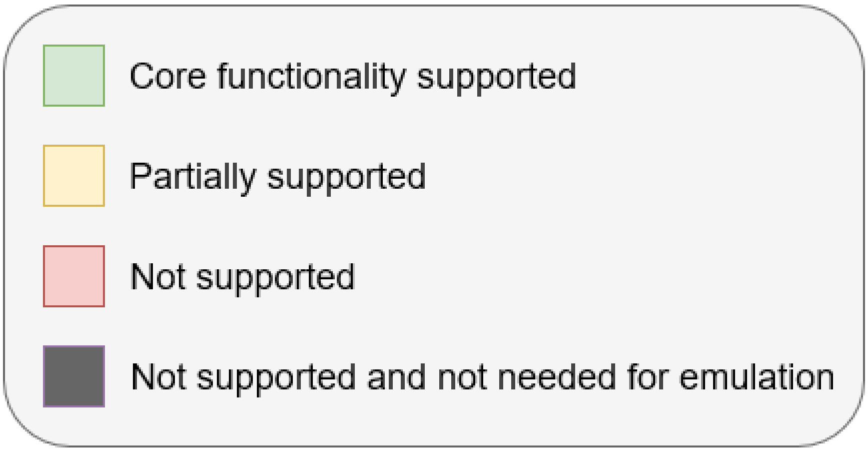
| Platform | Board Name | PetaLinux-specific Board Name | Yocto-specific board name |
|---|---|---|---|
| Zynq Ultrascale+ | ZCU111 | xilinx-zcu111 | zcu111-zynqmp |
| Zynq Ultrascale+ | ZCU1275 | xilinx-zcu1275 | zcu1275-zynqmp |
| Zynq Ultrascale+ | ZCU1285 | xilinx-zcu1285 | zcu1285-zynqmp |
| Zynq Ultrascale+ | ZCU208 | xilinx-zcu208 | zcu216-zynqmp |
| Zynq Ultrascale+ | ZCU216 | xilinx-zcu216 | zcu216-zynqmp |
| Zynq Ultrascale+ | Ultra96 | xilinx-ultra96 | ultra96-zynqmp |
| Versal | VCK190 | xilinx-vck190 | vck190-versal |
| Versal | VCK5000 | xilinx-vck5000 | |
| Versal | VMK180 | xilinx-vmk180 | vmk180-versal |

## 3.2 Supported Hardware

This section will cover what hardware and peripherals are supported in QEMU.

The block diagrams provide a broad overview of what is supported in each hardware block in QEMU, while the tables provide more detail on the support scope.

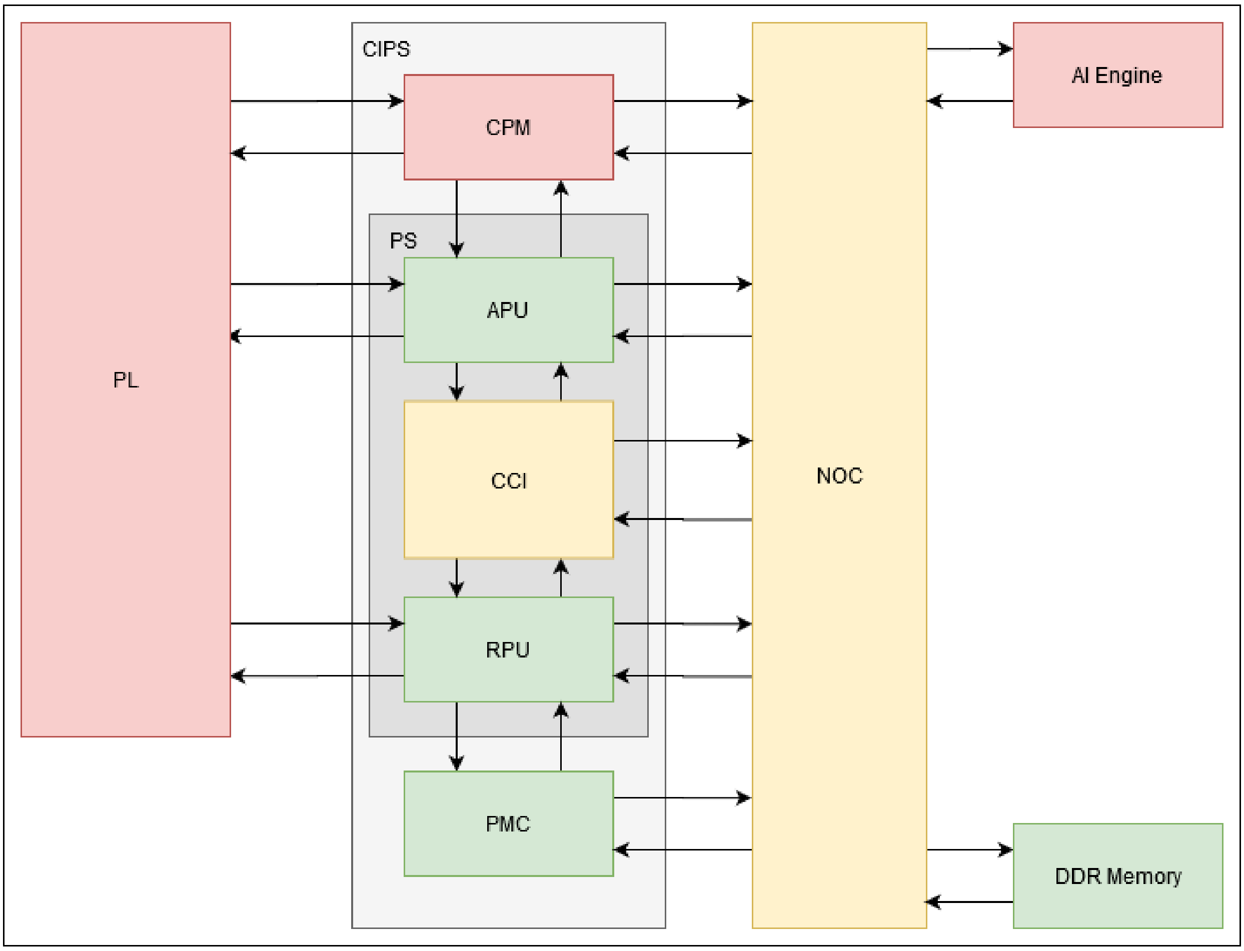
### 3.2.1 Block Diagrams



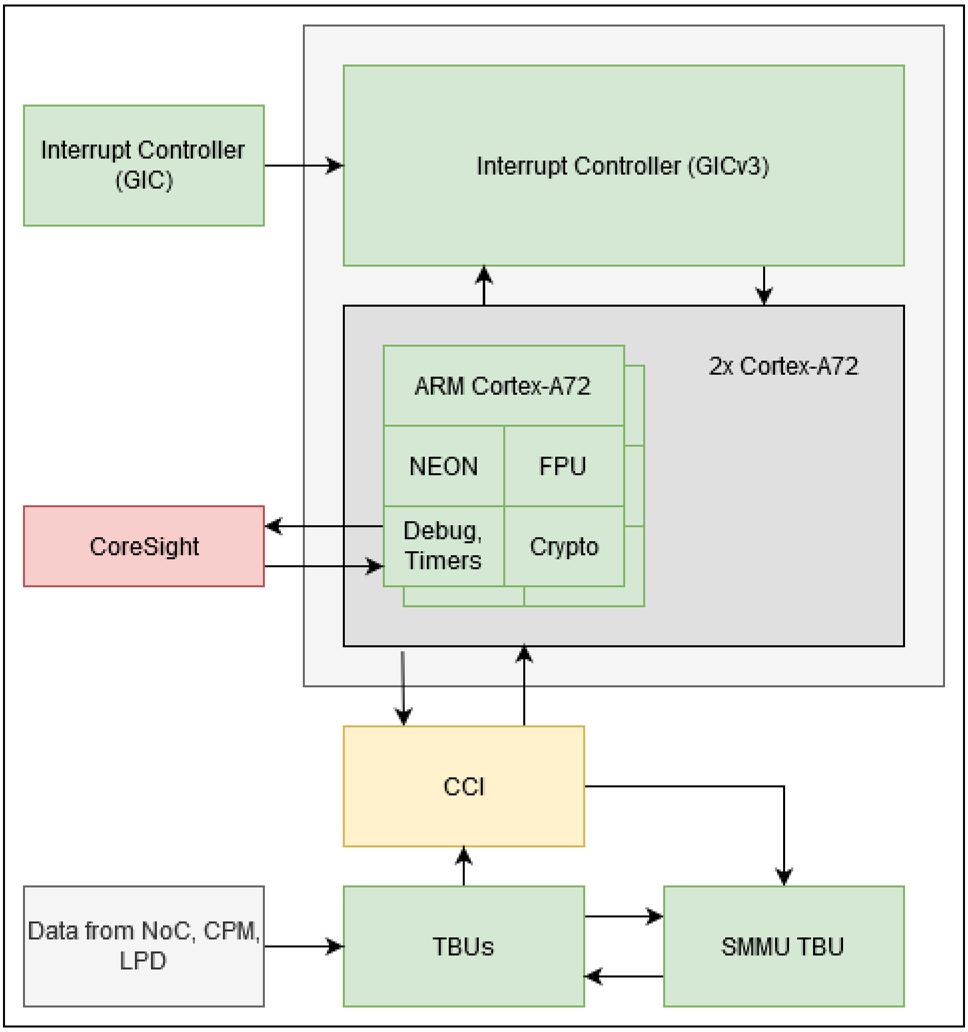

Versal

**Versal Block Diagram**

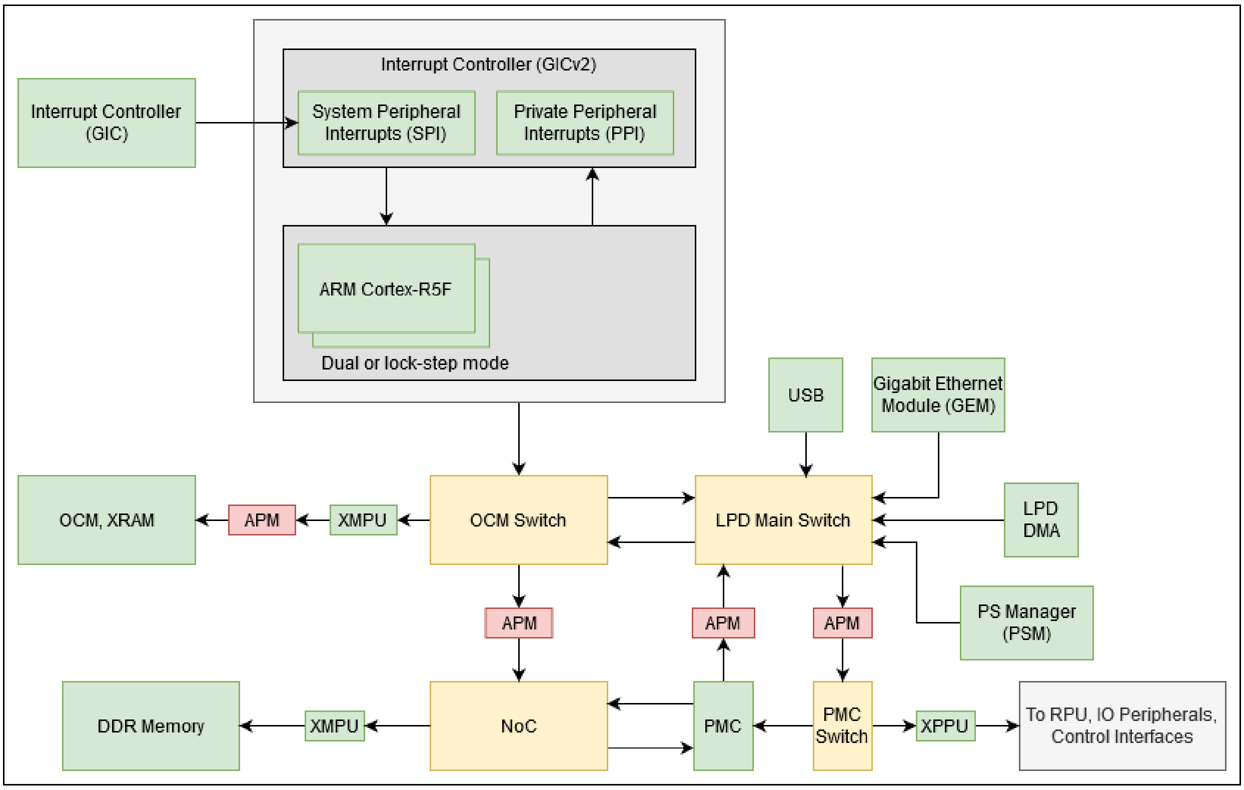
# Versal Architecture

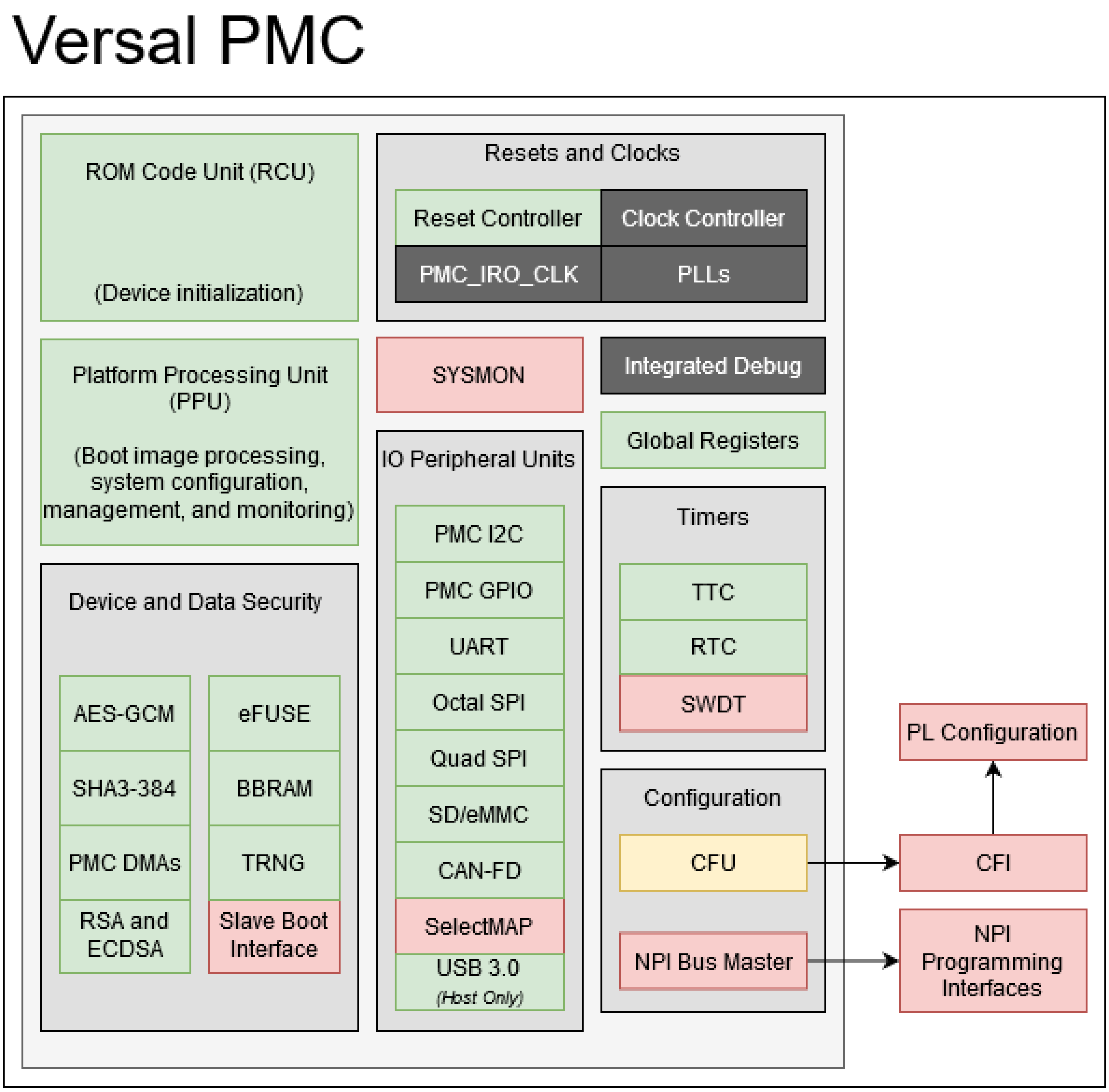


# Versal APU


Interrupt Controller (GIC)
Interrupt Controller (GICv3)
ARM Cortex-A72
2x Cortex-A72
NEON
FPU
CoreSight
Debug, Timers
Crypto
CCI
Data from NoC, CPM, LPD
TBUs
SMMU TBU

# Versal RPU

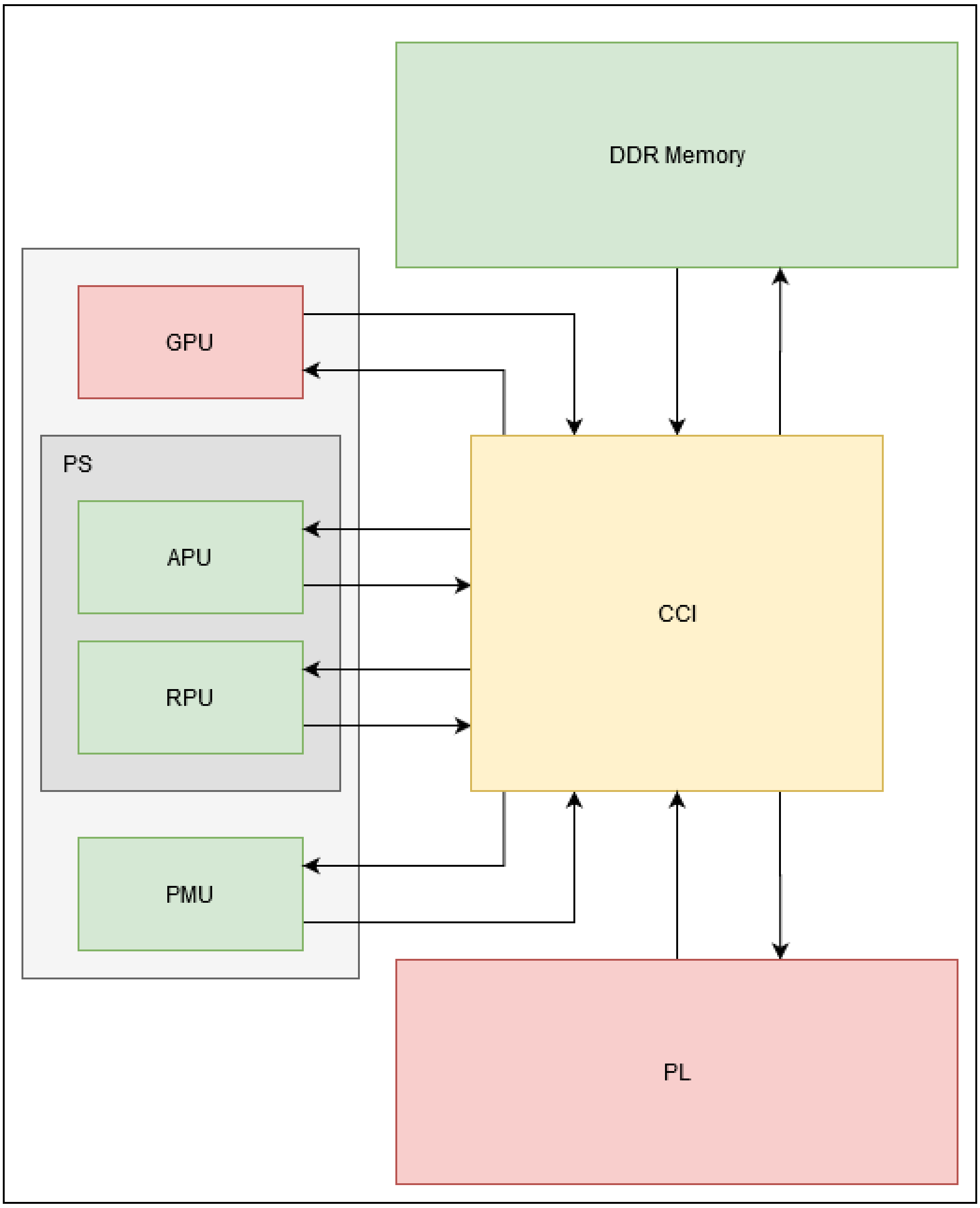

Interrupt Controller (GIC)
Interrupt Controller (GICv2)
System Peripheral Interrupts (SPI)
Private Peripheral Interrupts (PPI)
ARM Cortex-R5F
Dual or lock-step mode
USB
Gigabit Ethernet Module (GEM)
OCM, XRAM
APM
XMPU
OCM Switch
LPD Main Switch
LPD DMA
APM
APM
APM
PS Manager (PSM)
DDR Memory
XMPU
NoC
PMC
PMC Switch
XPPU
To RPU, IO Peripherals, Control Interfaces



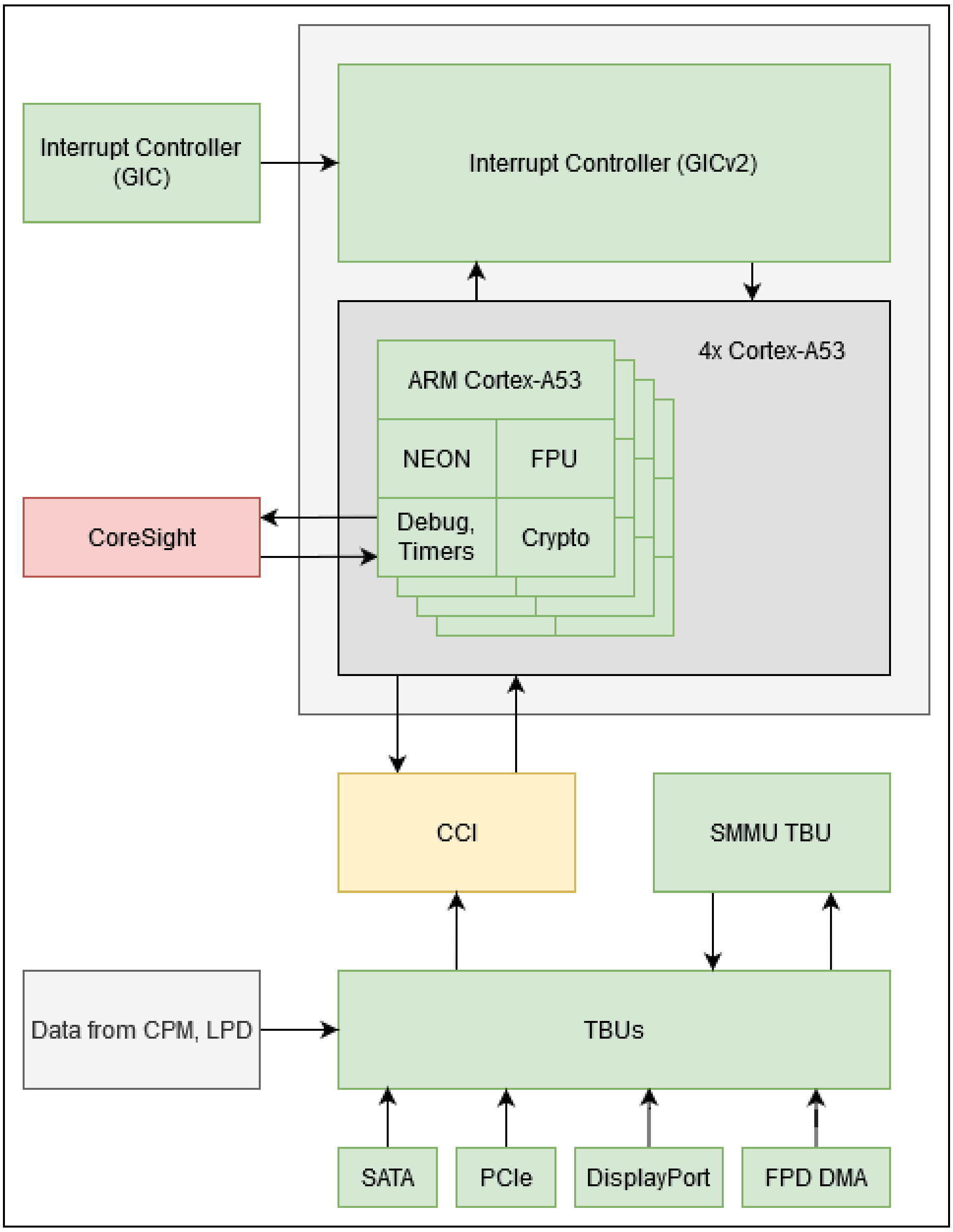

ZynqMP

**ZynqMP Block Diagram**

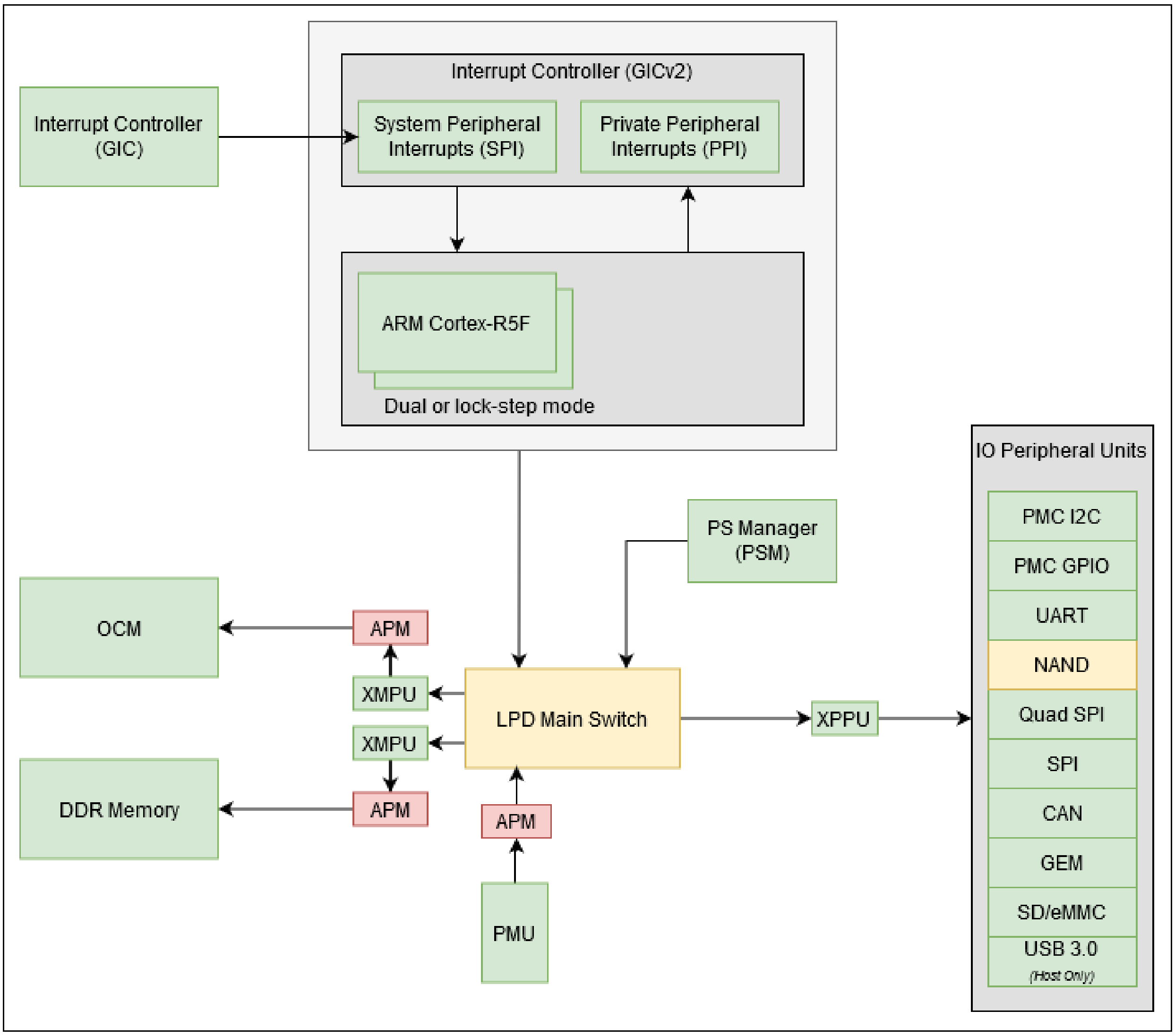
# ZynqMP Architecture

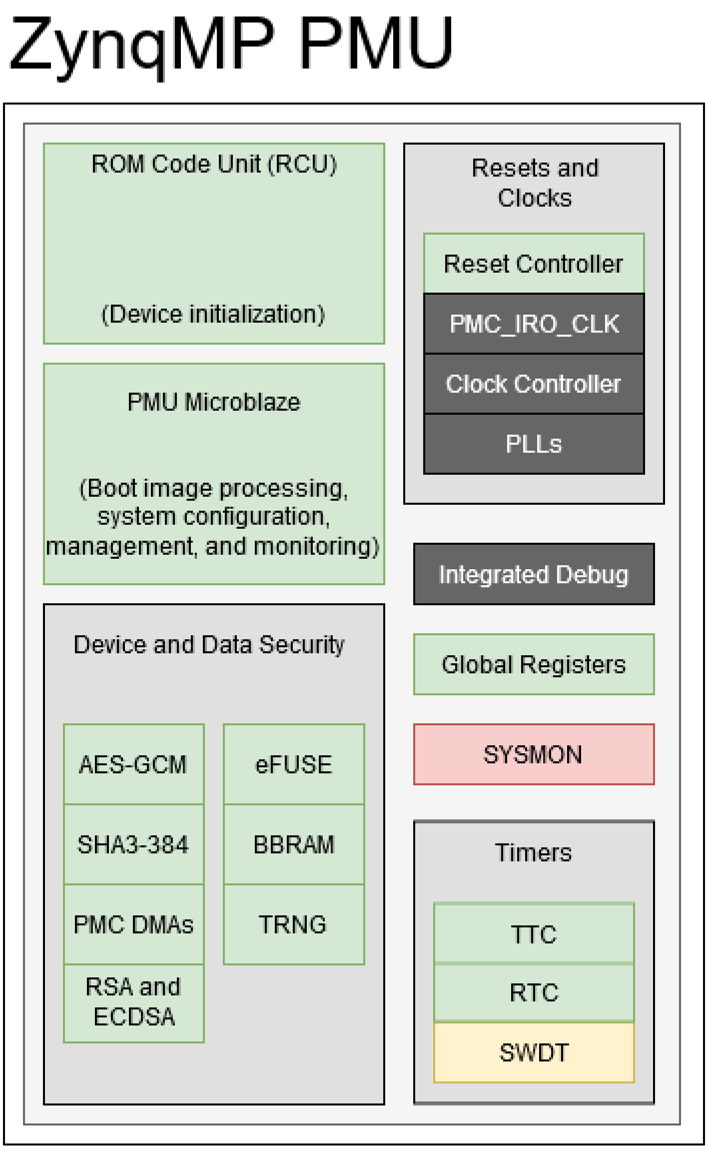


# ZynqMP FPD


Interrupt Controller (GIC)
Interrupt Controller (GICv2)
4x Cortex-A53
ARM Cortex-A53
NEON
FPU
Debug, Timers
Crypto
CoreSight
CCI
SMMU TBU
Data from CPM, LPD
TBUs
SATA
PCIe
DisplayPort
FPD DMA

# ZynqMP LPD


Interrupt Controller (GIC)
Interrupt Controller (GICv2)
System Peripheral Interrupts (SPI)
Private Peripheral Interrupts (PPI)
ARM Cortex-R5F
Dual or lock-step mode
IO Peripheral Units
PMC I2C
PMC GPIO
UART
NAND
Quad SPI
SPI
CAN
GEM
SD/eMMC
USB 3.0
(Host Only)
PS Manager (PSM)
OCM
APM
XMPU
XMPU
APM
LPD Main Switch
XPPU
DDR Memory
APM
PMU




### 3.2.2 Xilinx QEMU Features

**Supported Features Table**

Application Processing Units

| Description | Support Scope |
|---|---|
| ARM Interrupt Controller (GICv2) | Supported |
| ARM v8 (A53) Implementation. | Little Endian Only |
| ARM v8 EL0 Support | AArch64 and AArch32 |

| Description | Support Scope |
| --- | --- |
| ARM v8 EL1 Support | AArch64 and AArch32 |
| ARM v8 EL2 Support | AArch64 |
| ARM v8 EL3 Support | AArch64 |
| ARM v8 Crypto Instruction | Supported |
| Vector Floating Point (VFP) | 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. |

## Real-Time Processing Units

| Description | Support Scope |
| --- | --- |
| Dual Core ARM-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 | Only globally accessible TCM memory region is accessible.<br>Flat memory only, no control register implementation |
| Interrupt Controller | Supported |
| SLCRs | Very limited functionality. Only dummy registers, except SD `is_MMC` control. |

## PMU (ZynqMP and Versal)

| Description | Support Scope |
| --- | --- |
| IPI | Limited Connectivity specific to PMU functionality |
| Global Registers | Supported |

| Description | Support Scope |
|---|---|
| PMU MicroBlaze | Supported |
| PMU Interrupt Controller | Supported |

## I/O Peripherals and Devices

| Description | Support Scope |
|---|---|
| I/O Peripherals | Not all peripherals are implemented.<br>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) | Supported |
| SD Card model | No SDXC |
| QSPI controller (excludes Linear and Generic) | Supported |
| QSPI linear region | No XIP. Slow emulation performance. |
| QSPI NOR flash devices | Incomplete but reasonable selection of parts including many modern QSPI capable devices. |
| OSPI | Supported |
| UART Controller | Supported |
| SPI controller | Master mode only. |
| I2C controller | Master mode only. |
| DDR | Simple flat RAM model, no ECC. |
| CAN | Supported |
| CAN-FD | Supported |
| XADC | Not supported |
| GPIO | Limited functionality, connects to remote port. |
| MDIO and Ethernet PHY | Dummy models, show link up on requested PHY using MDIO |

| Description | Support Scope |
|---|---|
| USB | Supported - Host mode only. |
| SATA | Supported |
| PCI | Supported |

## DisplayPort

| Description | Support Scope |
|---|---|
| DP Model | AUX Communication.<br>DPCP: DisplayPort Configuration Information.<br>EDID. |
| DPDMA | Supported |
| 2 Layers | Supported |
| Alpha Blending | Supported |
| Audio | With some unexpected behavior |
| Dynamic resolution changes | Supported |
| Multiple pixel formats | Not all |
| Mali GPU | Not supported |

## AMBA AXI Bus

| Description | Support Scope |
|---|---|
| AMBA/AXI bus interconnect system | Simple bus model, no AXI/AMBA-specific features (such as MIDs).<br>Master IDs and Trustzone (secure versus non-secure) transactions supported. |
| Bus quality of service monitoring and control | N/A |
| On Chip Memory | Supported |
| AXI Performance Monitor (APM)<br>AXI Trace Monitor (ATM) | N/A |

### Additional ZynqMP and Versal Capabilities

| Description | Support Scope |
|---|---|
| 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 controller | Supported |
| PL-based AMS block | N/A |

### Miscellaneous QEMU Non-IP Related Feature

| Description | Support Scope |
|---|---|
| Ability to boot multiple software in different CPUs. | Supported |
| 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 | Supported |

### Timers and Clock Generators

| Description | Support Scope |
|---|---|
| Triple Timer Counter (TTC) | Supported |
| SWDT, WDT | Not Supported |
| Si570/71 | I2C device. Dummy emulation of clock generator. |

| Description | Support Scope |
|---|---|
| RTC | Supported |

Cryptographic modules

| Description | Support Scope |
|---|---|
| AES-GCM | Supported; all key sources |
| SHA3-384 | Supported |
| PMC DMAs | Supported |
| RSA | Supported |
| ECDSA | Supported |
| eFUSE | Supported; all documented fields |
| BBRAM | Supported |
| PUF | Supported. Limited to XilPuf API (Versal) and XilSkey API (ZynqMP).<br>Helper-data usable across all QEMU sessions, all user credentials, and all hosts. |
| TRNG | Supported. Caveat, TRNG generation is statistically unsecured. |

# 4 Launching QEMU Using Xilinx PetaLinux

This page will take you through the basic steps of running QEMU using Xilinx's PetaLinux.



Although the instructions here are targeting Xilinx's VCK190, the same steps can be repeated for any of the supported pre-built BSPs that can be found at the Embedded Development Downloads page.

To get you up and running quickly in this part we will focus on booting a pre-built image of QEMU for Xilinx's ZCU102 Development Board up to the first Linux prompt.

## 4.1 Download and Install PetaLinux

Go to the Embedded Development Downloads page. Please check the PetaLinux Tools documentation for installation instructions.

## 4.2 Download a Pre-Built PetaLinux VCK190 BSP

To get started quickly, Xilinx strongly recommends that you download the pre-built VCK190 board support package (BSP) from the Embedded Development Downloads page.

## 4.3 Create a Project Based on the VCK190 BSP

Open a terminal/console and type the following:

```
source <petalinux-install-path>/settings.sh
```

Select settings.sh for bash-based shells or settings.csh for C-based shells.

```
petalinux-create -t project -s <path to bsp>/xilinx-vck190-v2020.2-final.bsp -n
xilinx-qemu-first-run
cd xilinx-qemu-first-run
```

## 4.4 Boot to Linux Prompt

```
petalinux-boot --qemu --prebuilt 3
```

After you enter the final command above, first it will print all the commands used for booting QEMU, followed by QEMU boot sequence which loads the pre-built Linux image. At the prompt login, enter root as the username and root as the password.

> You can pass additional arguments to QEMU using the qemu-args "...." option. Enter any additional argument within the double-quotes. To quit the emulation, press **CTRL+A** followed by **X**. To switch between the serial port and the monitor, use **CTRL+A** followed by **C**.

> If you want to load the pre-built U-Boot image instead of Linux, you can use `--prebuilt 2` instead of `--prebuilt 3`.

# 5 Launching QEMU Using Xilinx Yocto toolchains

To get you up and running quickly, in this part we will focus on booting a pre-built image of QEMU for Xilinx's VCK190 Development Board, up to the first Linux prompt.

For more information on Yocto please follow this link.



## 5.1 Install the repo

Repo is a tool that enables the management of many git repositories given a single manifest file. The repo will fetch the git repositories specified in the manifest and, by doing so, sets up a Yocto Project build environment for you.

```
# Download the Repo script.
curl -k https://storage.googleapis.com/git-repo-downloads/repo > repo
# Generally, curl is install at /usr/bin/curl. If it is installed at custom location
please point it to correct location.

# Make it executable.
chmod a+x repo

# If it is correctly installed, you should see a Usage message when invoked with the
help flag.
repo --help
```

## 5.2 Fetch all sources

```
# initialize the repo.
repo init -u git://github.com/Xilinx/yocto-manifests.git -b <release-version>

# repo sync to get all sources
repo sync
```

Example of release-version: rel-v2020.2, rel-v2020.1 and rel-v2019.2 etc.

## 5.3 Source environment

```
# source the environment to build using bitbake.
source setupsdk
```

## 5.4 Build using bitbake

```
# provide machine name to build for.
MACHINE=<machine-name> bitbake petalinux-image-minimal
```

Examples of machine-name: vck190-versal, zcu102-zynqmp, zcu702-zynqmp etc.

> This step can take a lot of time depending upon host machine processing power. It will also required significant disk space, please see Yocto for more details.

## 5.5 Running QEMU

```
MACHINE=<machine-name> runqemu petalinux-image-minimal
```

After you enter the above command, the QEMU boot sequence loads the Linux image built with bitbake (previous step). At the prompt login, enter root as the username and root as the password.

> To quit the emulation, press **CTRL+A** followed by **X**. To switch between the serial port and the monitor, use **CTRL+A** followed by **C**.

# 6 Building and Running QEMU from Source Code

This page will take you through the steps you need to follow to build and run QEMU from Source Code.



## 6.1 Building QEMU Source Code On a Linux Host:

### 6.1.1 Downloading QEMU from Xilinx

The Xilinx QEMU source code is available on the Xilinx Git server and can be downloaded using the following command.

```
git clone git://github.com/Xilinx/qemu.git
cd qemu
```

The command above will by default clone the master branch of QEMU. This generally is ahead of the version of QEMU released with PetaLinux. This means it has improvements and new features compared to the released version but is also is less thoroughly tested and could have unknown bugs.

If you want to build the source that was used for the released version of QEMU, please checkout the appropriate tag instead of the master branch.
This can be done by first finding a list of tags:

```
git tag -l
```

And then checking out the release you want to use:

```
git checkout tags/xilinx-v2020.1
```

### 6.1.2 QEMU Linux Dependencies

If the configure or build steps fail, it is possible some build dependencies are missing. On Ubuntu use the below command to install most of the dependencies needed for building QEMU (please note you may find additional dependencies based on your setup).

```
sudo apt install libglib2.0-dev libgcrypt20-dev zlib1g-dev autoconf automake libtool
bison flex libpixman-1-dev
```

QEMU also includes submodules that will need to be checked out. Use the following command to checkout the appropriate submodules.

```
git submodule update --init dtc
```

### 6.1.3 Configuring QEMU

QEMU must be configured to build on the Linux host. This can be accomplished using the following command line.

```
mkdir build
cd build
../configure --target-list="aarch64-softmmu,microblazeel-softmmu" --enable-fdt --
disable-kvm --disable-xen --enable-gcrypt
```

The command shown above configures QEMU to build aarch64-softmmu and microblazeel-softmmu targets. Use "./configure –help" to know all supported targets and optional features in QEMU. If no "--target-list" provided, QEMU will build all targets.

> ⚠ Configuration and build steps will create a lot of files. To keep things easy, we will create a new folder called */build* and execute the following two steps in that folder.

### 6.1.4 Building QEMU

The following command line builds QEMU to run on the host computer.

```
make -j4
# -j options specifies the number of jobs (commands) to run simultaneously. You may
increase this to 16, 32 or more for faster build.
```

If the build is successful, an executable named `qemu-system-aarch64` and `qemu-system-microblazeel` will be created in the */aarch64-softmmu* and */microblazeel-softmmu* sub-directory respectively.

To validate the build run below commands:

```
./aarch64-softmmu/qemu-system-aarch64 --help
```

## 6.2 Building device tree binaries:

Device trees are used by the QEMU provided by Xilinx to internally generate a machine model.

### 6.2.1 Install device tree compiler

If device-tree-compiler is not installed, please install it using below command:

```
apt-get install device-tree-compiler
```

### 6.2.2 Clone Xilinx QEMU device trees

```
git clone git://github.com/Xilinx/qemu-devicetrees.git
cd qemu-devicetrees
```

### 6.2.3 Build device trees

You must have dtc on your PATH or specify `DTC=<path to device-tree-compiler executable>` with the `make` command below:

```
make
# make DTC=<path to device-tree-compiler executable>
```

This will give you a folder called */LATEST* which contains subdirectories for the different QEMU operating modes. Under these subdirectories are the board-specific device trees.

## 6.3 Running QEMU

QEMU can run in three ways:

1. Run Bare metal(standalone) applications on QEMU.
2. Run Linux on QEMU.

For running Linux on QEMU, please check petalinux boot section and look for the commands it prints when executing boot to linux prompt step.

We will talk about running bare metal applications in the next sections.

# 7 Running Bare Metal Applications on QEMU

This page shows you how to run a simple baremetal application on QEMU.



## 7.1 Let's run your first bare metal application "Hello World"

We will build and run a simple example that runs directly on the A53 out of the OCM memory of the Zynq Ultrascale+™ MPSoC. Click here to check the source code for this example. This example simply prints the line "Hello World on Xilinx's QEMU for ZCU102" and then it quits. Below is a code snippet of this example:

```
/* Main loop. */
int main(int argc, char* argv[]){

    SetUpPsUart0();

    outString("Hello World on Xilinx's QEMU for ZCU102\n");

    return(0);
}
```

### 7.1.1 Compile the bare metal example

1. Download and extract the AArch64 bare-metal (aarch64-none-elf) toolchain from here or use the PetaLinux installed toolchain.
2. Go to Xilinx UG 1169 example page and clone the repository. Examples for building bare metal applications are located under *BareMetal_examples* folder. For this hello-application, we will use *build_bare_metal_zcu102* example.
3. Run below commands on your terminal:

```
cd build_bare_metal_zcu102

# If you extracted the compiler to a folder.
make PATH=<path to extracted folder>/bin

#If you installed aarch64-none-elf-gcc. Installation path can be found using
which aarch64-none-elf-gcc.
make PATH=<installation path>
```

ⓘ If you have installed Xilinx Petalinux, Yocto and Vitis tools, this compiler can be found at: <install directory>/gnu/aarch64/lin/aarch64/bin/aarch64-none-elf-gcc.

### 7.1.2 Run it on QEMU

ⓘ We will use DTBs created in building dtbs section.

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

## 7.2 Running Bare Metal Applications

Below are a few more examples for running a bare metal application on different CPUs.

For more information on what the commands do, see the QEMU options section.

### 7.2.1 Running a bare metal application on Versal ACAP A72

```
qemu-system-aarch64 \
-M arm-generic-fdt \
-serial null -serial null -serial mon:stdio \
-device loader,file=<baremetal_for_versal_a72.elf>,cpu-num=0 \
-device loader,addr=0xFD1A0300,data=0x8000000e,data-len=4 \
-hw-dtb <device tree binary for Versal> \
-m <DDR memory size> \
-display none
```

ⓘ To quit the emulation, press **CTRL+A** followed by **X**. To switch between the serial port and the monitor, use **CTRL+A** followed by **C**.

### 7.2.2 Running a bare-metal application on Versal ACAP R5

```
qemu-system-aarch64 \
-M arm-generic-fdt \
-serial null -serial null -serial mon:stdio \
-device loader,file=<baremetal_for_versal_r5.elf>,cpu-num=2 \
-device loader,addr=0xff5e0300,data=0x16,data-len=4 \
-device loader,addr=0xff9a0100,data=0x1,data-len=4 \
-device loader,addr=0xff9a0000,data=0x48,data-len=4 \
-hw-dtb <device tree binary for Versal> \
-m <DDR memory size> \
-display none
```

### 7.2.3 Running a bare-metal application on Zynq Ultrascale+ MPSoC A53

```
qemu-system-aarch64 \
-M arm-generic-fdt \
-serial mon:stdio \
-device loader,file=<baremetal_for_zynqmp_a53.elf>,cpu-num=0 \
-device loader,addr=0xfd1a0104,data=0x8000000e,data-len=4 \
-hw-dtb <device tree binary for ZynqMP> \
-m <DDR memory size> \
-display none
```

### 7.2.4 Running a bare-metal application on Zynq Ultrascale+ MPSoC r5

```
qemu-system-aarch64 \
-M arm-generic-fdt \
-serial mon:stdio \
-device loader,file=<baremetal_for_zynqmp_r5.elf>,cpu-num=4 \
-device loader,addr=0XFF5E023C,data=0x80088fde,data-len=4 \
-device loader,addr=0xff9a0000,data=0x80000218,data-len=4 \
-hw-dtb <device tree binary for Versal> \
-m <DDR memory size> \
-display none
```

### 7.2.5 Running a bare-metal application on Zynq7000

ⓘ Zynq7000 DTBs are not included in the QEMU DTB repository, but can be found in a Zynq7000 PetaLinux project.

```
qemu-system-aarch64 \
-M arm-generic-fdt-7series \
-machine linux=on \
-serial /dev/null -serial mon:stdio \
-display none \
-kernel <guest image path> \
-dtb <Zynq7000 DTB path> \
-m <DDR memory size> \
-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 \
-device loader,addr=0xf8000910,data=0x0000000F,data-len=4
```

### 7.2.6 Running a bare-metal application on MicroBlaze

MicroBlaze DTBs are not included in the QEMU DTB repository, but can be found in a MicroBlaze PetaLinux project.

```
qemu-system-microblazeel \
-M microblaze-fdt-plnx \
-m <ram_size> \
-serial mon:stdio \
-display none \
-kernel <guest image path> \
-m <DDR memory size> \
-dtb <MicroBlaze DTB path>
```

# 8 QEMU Options and Commands

This section contains commonly used options and commands when using QEMU with virtual Xilinx hardware.

A larger list of options can be found here.

## 8.1 Options

These options are passed by the command line when starting QEMU.

If using PetaLinux tools, these options can be passed in by using the `--qemu-args "<options>"` argument when booting your machine.
If using a multi-architecture system, such as Zynq UltraScale+ MPSoC or Versal ACAP, arguments can be passed into the MicroBlaze QEMU machine by using the `--pmu-qemu-args "<options>"` argument.

| **Option** | **Description** | **Example** |
|---|---|---|
| `-accel tcg,thread=<multi\|single>` | Forces multi-threaded tiny code generator (MTTCG) to run or not run on QEMU. This means each VCPU runs on an individual host CPU thread.<br><br>This is enabled by default on systems where it is stable and does not need to be explicitly passed in.<br><br>Forcing MTTCG may cause incorrect emulation.<br><br>MTTCG can not be enabled with `icount`, since `icount` forces QEMU to run in a single thread | `-accel tcg,thread=multi`<br><br>Enables MTTCG<br><br>`-accel tcg,thread=single`<br><br>Disables MTTCG |

<table>
<tr><th>Option</th><th>Description</th><th>Example</th></tr>
<tr><td><code>-boot [order=<drives>] [,once=<drives>] [mode=<n>]</code></td><td>Specifies boot parameters when using U-Boot.
<table>
<tr><th>Boot pins</th><th>ZynqMP Boot Mode</th><th>Versal Boot Mode</th></tr>
<tr><td>1</td><td>QSPI24</td><td>QSPI24</td></tr>
<tr><td>2</td><td>QSPI32</td><td>QSPI32</td></tr>
<tr><td>3</td><td>SD0</td><td>SD0</td></tr>
<tr><td>4</td><td>NAND (unsupported in QEMU)</td><td>N/A</td></tr>
<tr><td>5</td><td>SD1</td><td>SD1</td></tr>
<tr><td>6</td><td>eMMC</td><td>eMMC</td></tr>
<tr><td>7</td><td>USB (unsupported in QEMU)</td><td>USB (unsupported in QEMU)</td></tr>
<tr><td>8</td><td>N/A</td><td>OSPI</td></tr>
</table></td><td><code>-boot mode=5</code><br>Sets the value of boot mode pins to 5 (boot from SD1).</td></tr>
<tr><td><code>-chardev <backend>,id=<id>[,mux=on|off] [,options]</code><br><code>-chardev help</code></td><td>Creates a character devices that allows communication between a QEMU front-end and back-end.<br>This can be thought of as a file descriptor that routes text from inside QEMU to outside QEMU.</td><td><code>-chardev socket,host=127.0.0.1,port=8887,id=pmu-console,server,nowait,mux=on</code><br>Creates a multiplexed chardev server socket on localhost, with port 8887 that has the name pmu-console.</td></tr>
</table>

| Option | Description | Example |
|---|---|---|
| -d <log item 1, ...> \| help | Enables logging to stderr for all log items specified.<br>If *help* is specified, a list of available log items is printed.<br>See QEMU Module Debug Printing for more information on how to use -d. | -d guest_errors<br>Prints any guest errors that occur during emulation to stderr.<br>-d guest_errors,trace:m25p80_command_decoded<br>Prints any guest errors, and trace output for m25p80 decoded commands to stderr. |
| -D <log file> | Redirects stderr to *log file*. | -D out.log<br>Redirects stderr to *out.log*. |
| -device <driver>[,prop[=<value>][,...]]<br>-device help | Adds the device *driver.* Properties and values depend on the driver being added.<br>Typically, when emulating Xilinx hardware with QEMU, this option is used to load software and bring CPUs out of reset by using the loader driver. See: QEMU Loader Options for more information. | -device loader,file=/path/to/software/bl31.elf,cpu-num=0<br>Loads *bl31.elf* and runs it on CPU 0<br>-device loader,addr=0xFD1A0104,data=0x0000000E,data-len=4<br>Writes 0x0000000E to the register 0xFD1A0104, which brings A53 CPU 0 out of reset on ZymqMP.<br>When writing data to an address, the order must be addr=<addr>,data=<data>,data-len=<len> |
| -display <type> | The type of display to be used | -display none<br>Specifies there's no display. |
| -drive [file=<path>][,if=<interface>]\\[,format=<format>][index=<index>] | Defines a new drive.<br>See: Storage Media for more information. | -drive file=/path/to/img/qemu_boot.img,if=sd,format=raw,index=1<br>Creates an SD drive at index 1 with the image *qemu_boot.img*. |

| Option | Description | Example |
|---|---|---|
| `-dtb <dtb path>` | The hardware description for the QEMU machine. Anything that QEMU cannot emulate is discarded.<br><br>If a Linux kernel is present, the dtb is passed to it through a memory buffer. See: -dtb vs -hw-dtb for more detail. | `-dtb /path/to/dtb/system.dtb`<br><br>Speficies the QEMU hardware description to be *system.dtb* and passes it to the Linux Kernel (if present). |
| `-gdb <host>:<port>` | Opens a GDB server on *host:port*. | `-gdb tcp:127.0.0.1:9001`<br><br>Creates a GDB server on `127.0.0.1:9001` |
| `-global <property>=<val>` | Sets driver properties for devices created by the machine model. | `-global xlnx,zynqmp-boot.cpu-num=0`<br><br>Releases A53 CPU 0 from reset. |
| `-hw-dtb <dtb path>` | The hardware description for the QEMU machine.<br>See: -dtb vs -hw-dtb for how this differs from the `-dtb` parameter. | `-hw-dtb /path/to/dtb/zynqmp-qemu-multiarch-arm.dtb`<br><br>Specifies the QEMU hardware description to be *zynqmp-qemu-multiarch-arm.dtb*. |
| `-kernel <kernel img>` | Specifies an image, such as a kernel or bare-metal image. | `-kernel /path/to/kernel/pmu_rom_qemu_sha3.elf`<br><br>Specifies the image *pmu_rom_qemu_sha3.elf*. |
| `-m <memory size>` | Allocates *memory size* bytes of RAM for the virtual machine.<br><br>By default, QEMU allocates 128MB of RAM. | `-m 4G`<br><br>This machine will have 4GB of RAM allocated<br><br>`-m 512M`<br><br>This machine will have 512MB of RAM allocated |
| `-M <machine>` | Specifies the machine architecture. | `-M arm-generic-fdt`<br><br>Specifies that this is an ARM-generic-fdt machine.<br><br>`-M microblaze-fdt`<br><br>Specifies that this is a microblaze-fdt machine. |

<table>
<tr><th>Option</th><th>Description</th><th>Example</th></tr>
<tr><td>-machine-path <dir></td><td>Specifies a directory where QEMU creates shared memory files and named UNIX sockets.<br><br>When emulating Xilinx hardware with QEMU, this is used to share data between architectures in multiarch environments, or for co-simulation.<br><br>⚠ It is recommended you clear the -machine-path directory between boots. PetaLinux and Yocto tools do this for you. See this section for more information.</td><td>-machine-path /tmp/qemu-shm<br><br>Specifies that /tmp/qemu-shm is where QEMU shared memory and UNIX sockets will be created.</td></tr>
<tr><td>-net nic[,netdev=<nd>][,macaddr=<mac>]<br>-net user|tap|bridge|socket|l2tpv3|vde[,...]</td><td>When nic is specified, this option configures or creates an on-board network interface card and connects it either to the emulated hub, or to the netdev nd.<br><br>When user is specified, this option configures a host network back-end and connects it to the emulated default hub.</td><td>-net nic<br><br>Creates an on-board NIC.<br><br>-net user,id=eth0,tftp=/host/path/for/tftp<br><br>Creates a network back-end with the ID eth0 that has TFTP access to the host path /host/path/for/tftp.</td></tr>
<tr><td>-nographic</td><td>This machine will have no graphic output.<br><br>Also passes in the -serial stdio option.</td><td>-nographic<br><br>Specifies there's no graphics and passes in -serial stdio as well.</td></tr>
<tr><td>-S</td><td>Pauses the machine on the first instruction.<br><br>Typically this is used with the gdb option to debug the boot sequence of your machine.</td><td>-S<br><br>Pauses the machine on the first instruction.</td></tr>
</table>

| Option | Description | Example |
|---|---|---|
| `-s` | Shorthand for `-gdb tcp::1234` | `-s`<br>Creates a GDB server on `127.0.0.1:1234` |
| `-serial <dev>` | Connects the serial device to *dev*. | `-serial mon:stdio`<br>Connect this serial device to the QEMU monitor and STDIO.<br>`-serial null`<br>Don't connect this serial device<br>`-serial chardev:pmu-console`<br>Connect this serial device to the chardev *pmu-console*. |

## 8.1.1 -dtb vs -hw-dtb

Xilinx QEMU supports two device tree options:

- -hw-dtb is used for the hardware device tree binary that QEMU uses to generate the model. Hardware device tree binaries will have the name of the device it represents. For example, `board-versal-ps-vc-p-a2197-00.dtb` is for Versal and `zcu102-arm.dtb` for the ZCU102 board.
- -dtb is generally a Linux device tree binary used for Linux kernel boots. With the DTB passed in with `-dtb` option, QEMU removes the nodes that it cannot emulate and later copies them to RAM for the kernel. Xilinx Linux kernel device tree binaries are typically named `system.dtb`.

For booting Linux on multi-arch platforms, such as Zynq Ultrascale+ MPSoC and Versal ACAP, we use the `-hw-dtb` option, since guest device tree loading is skipped by QEMU.
If both `-hw-dtb` and `-dtb` are used in multi-arch environments, the DTB passed in with `-dtb` is ignored.

For standalone (single-arch) flows, these two arguments are fully interchangeable; specify only one or the other.

> ⚠ This procedure is applicable only when `-kernel` is passed on QEMU command line.
>
> For Zynq Ultrascale+ MPSoC and Versal ACAP in a multi-architecture environment, the QEMU DTB is different from the kernel `system.dtb`.

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

### Zynq UltraScale+ MPSoC

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

### Versal ACAP

- Single-arch: `versal-qemu-ps.dtb`

- Multi-arch: `versal-qemu-multiarch-ps.dtb`, `versal-qemu-multiarch-pmc.dtb`

If building DTBs from source, the single-arch and multi-arch device trees will appear under `path/to/dts-repo/LATEST/SINGLE_ARCH` and `path/to/dts-repo/LATEST/MULTI_ARCH` respectively.

### 8.1.2 QEMU Loader 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 loader driver can perform the following tasks:

- Load software or data into RAM sections
- Set the CPU entry points
- Release CPUs from reset
- Write to registers

In Zynq Ultrascale+ MPSoC, by default, the six ARM CPUs (four ARM-A53 and two ARM-R5) 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 only one mode for each `-device` argument.
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.

When loading raw images, the address is specified with the `addr=<value>` argument. The default address 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. Specifying `cpu-num` also sets the Program Counter (PC).

The CPU has a set entry point in the following situations:

- For ELFs and U-Boot images, the entry point 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.

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).
In this case, you can pass the `force-raw=true` sub-option to instruct the loader to treat the image as raw.
You must specify the `addr`, since the section information in the ELF and U-Boot images are ignored.

#### Single Transaction Mode

In single transaction mode, a single bus transaction occurs.

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

- `addr` must be 32-bit aligned.

Before the initial system reset for Zynq Ultrascale+ MPSoC or Versal ACAP, QEMU performs the specified bus transaction.

Optionally, you can specify a CPU using the `cpu-num=<value>` sub-option.
The list of CPU enumeration values for Zynq Ultrascale+ MPSoC and Versal ACAP are given below:

| Value | Zynq Ultrascale+ MPSoC CPU |
|---|---|
| 0 | A53-0 |
| 1 | A53-1 |
| 2 | A53-2 |
| 3 | A53-3 |
| 4 | R5-0 |
| 5 | R5-1 |

| Value | Versal ACAP CPU |
|---|---|
| 0 | A72-0 |
| 1 | A72-1 |
| 2 | R5-0 |
| 3 | R5-1 |

⚠ If you edit the `/cpus` node in the DTB, these enumerations change.

The single transaction occurs from the perspective of the specified CPU. If you do not specify a CPU, then the system assumes `CPU0`.

## 8.1.3 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 persist across multiple sessions.

### Argument Format

```
-drive file=<image-path>, if=(mtd|sd|
pflash),format=<fmt>,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 single or dual-parallel arrangement.
The file size for each should match Flash model size.

If you are using only a single mode QSPI, then only one QSPI argument is needed.

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

Creating a QSPI image and booting with QSPI is outlined here.

### Flash Striper Utility

In parallel mode, the data that QSPI 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 into the two images.
The syntax is as follows:

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

Where:

- `<input>` is a 128MB image for Zynq UltraScale+ or a 256MB image on Versal ACAP.
- `<out0>` and `<out1>` are the two images with half the size of the input image (64MB or 128MB) passable to the `-mtdblock` arguments for QSPI.

The reverse is also possible, taking the two striped images and converting them back to a single 128MB or 256MB image as shown in the following command:

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

### Building the Flash Striper utilities

The complete flash striper utility source code is available here.

### Building the Bit Stripe Utilities

The bit stripe utilities are used for legacy QSPI (LQSPI) on Zynq7000.

Compile the flash striper utility for your host with the following commands:

```
gcc flash_stripe.c -o flash_stripe
gcc flash_stripe.c -o flash_unstripe
```

### Building the Byte Stripe Utilities

The byte stripe utilities are used for generic QSPI (GQSPI) on Zynq UltraScale+ MPSoC and Versal ACAP.

Compile the flash striper utility for your host with the following commands:

```
gcc flash_stripe.c -o flash_stripe_bw -DFLASH_STRIP_BW
gcc flash_stripe.c -o flash_unstripe_bw -DUNSTRIP -DFLASH_STRIP_BW
```

`flash_strip_be_bw` is also available as part of the PetaLinux tools.

Supported QSPI Flash Models

| Vendor | Flash Models |
|---|---|
| Atmel | at25fs010, 25fs040, at25df041a, at25df321a, at25df641, at26f004, at26df081a, at26df161a, at26df321, at45db081d |
| EON | en25f32, en25p32, en25q32b, en25p64, en25q64 |
| GigaDevice | gd25q32, gd25q64 |
| Intel/Numonyx | 160s33b, 320s33b, 640s33b n25q064 |
| Macronix | mx25l2005a, mx25l4005a, mx25l8005, mx25l1606e, mx25l3205d, mx25l6405d, mx25l12805d, mx25l12855e, mx25l25635e, mx25l25655e |
| Micron | n25q032a1, 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.

> ⚠ SDXC (>32GB) sizes do not work.

If an SD argument is not specified, no SD card is modeled, the corresponding SD slot is ejected.
This is different from SPI, where the flash is still modeled even if an image is not provided.

Information for booting with SD can be found here.

### eMMC

QEMU will model an eMMC card for `-drive file=<file_path>,if=sd`. This requires changes in the system-level control register (SLCR) registers to enable eMMC. The table below shows the registers that need to be modified:

| Platform | Register | Fields |
|---|---|---|
| Zynq UltraScale+ MPSoC | IOU_SLCR | CTRL_REG_SD |
| Versal ACAP | PMC_IOU_SLCR | SD0_CTRL_REG<br>SD1_CTRL_REG |

For Zynq Ultrascale+ MPSoC and Versal `index=2` works as an EMMC card connected to `sdhci0`.

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

### EEPROM

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

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

Users can find the information on the connected EEPROMs in the board's DTS file.

## 8.2 Boot Examples

This section contains boot examples for Zynq Ultrascale+ MPSoC and Versal ACAP and what each parameter means. Your boot parameters may vary slightly, depending on your project.

**Zynq UltraScale+ MPSoC Boot**

```
$ qemu-system-microblazeel \
-M microblaze-fdt \
-serial mon:stdio \
-serial /dev/null \
-display none \
-kernel /scratch/petalinux-images/xilinx-zcu102-2020.1/pre-built/linux/images/pmu_rom_qemu_sha3.elf \
-device loader,file=/scratch/petalinux-images/xilinx-zcu102-2020.1/pre-built/linux/images/pmufw.elf \
-hw-dtb /scratch/petalinux-images/xilinx-zcu102-2020.1/pre-built/linux/images/zynqmp-qemu-multiarch-pmu.dtb \
-machine-path /tmp/tmp.MOj3amuPFs \
-device loader,addr=0xfd1a0074,data=0x1011003,data-len=4 \
-device loader,addr=0xfd1a007C,data=0x1010f03,data-len=4 \
&
$ qemu-system-aarch64 \
-M arm-generic-fdt \
-serial mon:stdio \
-serial /dev/null \
-display none \
-device loader,file=/scratch/petalinux-images/xilinx-zcu102-2020.1/pre-built/linux/images/bl31.elf,cpu-num=0 \
-device loader,file=/scratch/petalinux-images/xilinx-zcu102-2020.1/pre-built/linux/images/Image,addr=0x00080000 \
-device loader,file=/scratch/petalinux-images/xilinx-zcu102-2020.1/pre-built/linux/images/system.dtb,addr=0x15e80000 \
-device loader,file=/scratch/petalinux-images/xilinx-zcu102-2020.1/build/misc/linux-boot/linux-boot.elf \
-gdb tcp::9000 \
-dtb /scratch/petalinux-images/xilinx-zcu102-2020.1/pre-built/linux/images/system.dtb  \
-net nic -net nic -net nic -net nic,netdev=eth0 \
-netdev user,id=eth0,tftp=/scratch/petalinux-images/xilinx-zcu102-2020.1/images/linux \
-hw-dtb /scratch/petalinux-images/xilinx-zcu102-2020.1/pre-built/linux/images/zynqmp-qemu-multiarch-arm.dtb \
-machine-path /tmp/tmp.MOj3amuPFs \
-global xlnx,zynqmp-boot.cpu-num=0 \
-global xlnx,zynqmp-boot.use-pmufw=true \
-m 4G
```

| Microblaze QEMU Parameter | Meaning |
|---|---|
| `-M microblaze-fdt` | Set the machine to be a Microblaze machine |
| `-serial mon:stdio` | Set a serial device to `STDIO` and the QEMU monitor. |

| Microblaze QEMU Parameter | Meaning |
|---|---|
| -serial /dev/null | Set a serial device to /dev/null |
| -display none | Don't use a display |
| -kernel /scratch/petalinux-images/xilinx-zcu102-2020.1/pre-built/linux/images/pmu_rom_qemu_sha3.elf | Make pmu_rom_qemu_sha3.elf the kernel image |
| -device loader,file=/scratch/petalinux-images/xilinx-zcu102-2020.1/pre-built/linux/images/pmufw.elf | Load pmufw.elf |
| -hw-dtb /scratch/petalinux-images/xilinx-zcu102-2020.1/pre-built/linux/images/zynqmp-qemu-multiarch-pmu.dtb | Use zynqmp-qemu-multiarch-pmu.dtb as the DTB for the machine |
| -machine-path /tmp/tmp.MOj3amuPFs | Use /tmp/tmp.MOj3amuPFs as the QEMU shared memory directory |
| -device loader,addr=0xfd1a0074,data=0x1011003,data-len=4 | Write 0x01011003 to address 0xFD1A0074.<br>In this case, 0xFD1A0074 is the DP_AUDIO_REF_CTRL register |
| -device loader,addr=0xfd1a007C,data=0x1010f03,data-len=4 | Write 0x01010f03 to address 0xFD1A007C.<br>In this case, 0xFD1A007C is the DP_STC_REF_CTRL register |

| Aarch64 QEMU Parameter | Meaning |
|---|---|
| -M arm-generic-fdt | Set the machine to be an ARM machine |
| -serial mon:stdio | Set a serial device to STDIO and the QEMU monitor. |
| -serial /dev/null | Set a serial device to /dev/null |
| -display none | Don't use a display |

| Aarch64 QEMU Parameter | Meaning |
|---|---|
| `-device loader,file=/scratch/petalinux-images/xilinx-zcu102-2020.1/pre-built/linux/images/bl31.elf,cpu-num=0` | Load `bl31.elf` and run it on CPU 0 |
| `-device loader,file=/scratch/petalinux-images/xilinx-zcu102-2020.1/pre-built/linux/images/Image,addr=0x00080000` | Load `Image` and put it at address `0x80000` |
| `-device loader,file=/scratch/petalinux-images/xilinx-zcu102-2020.1/pre-built/linux/images/system.dtb,addr=0x15e80000` | Load `system.dtb` and put it at address `0x15E80000` |
| `-device loader,file=/scratch/petalinux-images/xilinx-zcu102-2020.1/build/misc/linux-boot/linux-boot.elf` | Load `linux-boot.elf` |
| `-gdb tcp::9000` | Make the QEMU GDB server listen on `localhost:9000` |
| `-dtb /scratch/petalinux-images/xilinx-zcu102-2020.1/pre-built/linux/images/system.dtb` | Use `system.dtb` as the hardware description for the machine.<br>This DTB is passed to the Linux Kernel |
| `-net nic -net nic -net nic -net nic,netdev=eth0` | Create 4 NICs, the last of which is connected to the network interface `eth0` |
| `-netdev user,id=eth0,tftp=/scratch/petalinux-images/xilinx-zcu102-2020.1/images/linux` | Create a network back-end on `eth0`, and change the default TFTP path to `/scratch/petalinux-images/xilinx-zcu102-2020.1/images/linux` |
| `-hw-dtb /scratch/petalinux-images/xilinx-zcu102-2020.1/pre-built/linux/images/zynqmp-qemu-multiarch-arm.dtb` | Use `zynqmp-qemu-multiarch-arm.dtb` as the hardware description for the machine |
| `-machine-path /tmp/tmp.MOj3amuPFs` | Use `/tmp/tmp.MOj3amuPFs` as the QEMU shared memory directory |
| `-global xlnx,zynqmp-boot.cpu-num=0` | Set `cpu-num` to 0 in the `zynqmp-boot` driver |

| Aarch64 QEMU Parameter | Meaning |
|---|---|
| `-global xlnx,zynqmp-boot.use-pmufw=true` | Set `pmufw` to true in the `zynqmp-boot` driver |
| `-m 4G` | Allocate 4GB of RAM for the machine |

### Versal ACAP Boot

```
$ qemu-system-microblazeel
-M microblaze-fdt \
-serial mon:stdio \
-display none \
-device loader,addr=0xf0000000,data=0xba020004,data-len=4 \
-device loader,addr=0xf0000004,data=0xb800fffc,data-len=4 \
-device loader,file=/scratch/petalinux-images/xilinx-vc-e-a2197-00-2019.2/pre-built/linux/images/pmc_cdo.bin,addr=0xf2000000 \
-device loader,file=/scratch/petalinux-images/xilinx-vc-e-a2197-00-2019.2/pre-built/linux/images/BOOT_bh.bin,addr=0xf201e000,force-raw \
-device loader,file=/scratch/petalinux-images/xilinx-vc-e-a2197-00-2019.2/pre-built/linux/images/plm.elf \
-hw-dtb /scratch/petalinux-images/xilinx-vc-e-a2197-00-2019.2/pre-built/linux/images/versal-qemu-multiarch-pmc.dtb \
-machine-path /tmp/tmp.qp3oHny0iJ \
-device loader,addr=0xF1110624,data=0x0,data-len=4 \
-device loader,addr=0xF1110620,data=0x1,data-len=4 \
&
$ qemu-system-aarch64 \
-M arm-generic-fdt \
-serial null -serial null -serial mon:stdio \
-display none \
-boot mode=5 \
-drive file=/scratch/petalinux-images/xilinx-vc-e-a2197-00-2019.2/pre-built/linux/images/qemu_boot.img,if=sd,format=raw,index=1 \
-device loader,file=/scratch/petalinux-images/xilinx-vc-e-a2197-00-2019.2/pre-built/linux/images/Image,addr=0x00080000 \
-device loader,file=/scratch/petalinux-images/xilinx-vc-e-a2197-00-2019.2/pre-built/linux/images/system.dtb,addr=0x15e80000 \
-gdb tcp::9000 -dtb /scratch/petalinux-images/xilinx-vc-e-a2197-00-2019.2/pre-built/linux/images/system.dtb \
-net nic -net nic,netdev=eth0 \
-netdev user,id=eth0,tftp=/scratch/petalinux-images/xilinx-vc-e-a2197-00-2019.2/pre-built/linux/images/ \
-hw-dtb /scratch/petalinux-images/xilinx-vc-e-a2197-00-2019.2/pre-built/linux/images/versal-qemu-multiarch-ps.dtb \
-machine-path /tmp/tmp.qp3oHny0iJ \
-m 4G
```

| Microblaze QEMU Parameter | Meaning |
|---|---|
| `-M microblaze-fdt` | Set the machine to be a Microblaze machine |
| `-serial mon:stdio` | Set a serial device to `STDIO` and the QEMU monitor |

| Microblaze QEMU Parameter | Meaning |
|---|---|
| `-display none` | Don't use a display |
| `-device loader,addr=0xf0000000,data=0xba020004,data-len=4` | Write `0xBA020004` to address `0xF0000000` |
| `-device loader,addr=0xf0000004,data=0xb800fffc,data-len=4` | Write `0xB800FFFC` to address `0xF0000004` |
| `-device loader,file=/scratch/petalinux-images/xilinx-vc-e-a2197-00-2019.2/pre-built/linux/images/pmc_cdo.bin,addr=0xf2000000` | Load `pmc_cdo.bin` into address `0xF2000000` |
| `-device loader,file=/scratch/petalinux-images/xilinx-vc-e-a2197-00-2019.2/pre-built/linux/images/BOOT_bh.bin,addr=0xf201e000,force-raw` | Load `BOOT_bh.bin` into address `0xF201E0000` |
| `-device loader,file=/scratch/petalinux-images/xilinx-vc-e-a2197-00-2019.2/pre-built/linux/images/plm.elf` | Load `plm.elf` |
| `-hw-dtb /scratch/petalinux-images/xilinx-vc-e-a2197-00-2019.2/pre-built/linux/images/versal-qemu-multiarch-pmc.dtb` | Use `zynqmp-qemu-multiarch-pmu.dtb` as the DTB for the machine |
| `-machine-path /tmp/tmp.MOj3amuPFs` | Use `/tmp/tmp.MOj3amuPFs` as the QEMU shared memory directory |
| `-device loader,addr=0xfd1a0074,data=0x1011003,data-len=4` | Write `0x01011003` to address `0xFD1A0074`. In this case, `0xFD1A0074` is the `DP_AUDIO_REF_CTRL` register |
| `-device loader,addr=0xfd1a007C,data=0x1010f03,data-len=4` | Write `0x01010f03` to address `0xFD1A007C`. In this case, `0xFD1A007C` is the `DP_STC_REF_CTRL` register |

| Aarch64 QEMU Parameter | Meaning |
|---|---|
| `-M arm-generic-fdt` | Set the machine to be an ARM machine |

| Aarch64 QEMU Parameter | Meaning |
|---|---|
| `-serial null -serial null -serial mon:stdio` | Set a serial device to `STDIO` and the QEMU monitor |
| `-display none` | Don't use a display |
| `-boot mode=5` | Set the boot pins to 5, which sets up the board to boot via SD |
| -drive file=/scratch/petalinux-images/xilinx-vc-e-a2197-00-2019.2/pre-built/linux/images/qemu_boot.img,if=sd,format=raw,index=1 | Create a new drive, which is an SD image that contains `qemu_boot.img` |
| `-device loader,file=/scratch/petalinux-images/xilinx-vc-e-a2197-00-2019.2/pre-built/linux/images/Image,addr=0x00080000` | Load `Image` and put it at address `0x80000` |
| `-device loader,file=/scratch/petalinux-images/xilinx-vc-e-a2197-00-2019.2/pre-built/linux/images/system.dtb,addr=0x15e80000` | Load `system.dtb` and put it at address `0x15E80000`<br>`microblazeel \`<br>`-M microblaze-fdt \`<br>`-serial mon:stdio \`<br>`-serial /dev/nul` |
| `-gdb tcp::9000` | Make the QEMU GDB server listen on `localhost:9000` |
| `-dtb /scratch/petalinux-images/xilinx-vc-e-a2197-00-2019.2/pre-built/linux/images/system.dtb` | Use `system.dtb` as the hardware description for the machine.<br>This DTB is passed to the Linux Kernel |
| `-net nic -net nic,netdev=eth0` | Create 2 NICs, the last of which is connected to the network interface `eth0` |
| `-netdev user,id=eth0,tftp=/scratch/petalinux-images/xilinx-vc-e-a2197-00-2019.2/pre-built/linux/images/` | Create a network back-end on `eth0`, and change the default TFTP path to `/scratch/petalinux-images/xilinx-vc-e-a2197-00-2019.2/pre-built/linux/images/` |
| `-hw-dtb /scratch/petalinux-images/xilinx-vc-e-a2197-00-2019.2/pre-built/linux/images/versal-qemu-multiarch-ps.dtb` | Use `zynqmp-qemu-multiarch-arm.dtb` as the hardware description for the machine |

| Aarch64 QEMU Parameter | Meaning |
| --- | --- |
| `-machine-path /tmp/tmp.MOj3amuPFs` | Use `/tmp/tmp.MOj3amuPFs` as the QEMU shared memory directory |
| `-m 4G` | Allocate 4GB of RAM for the machine |

As an side note, these boot commands were created by going into a PetaLinux project and booting with the following command:

```
$ petalinux-boot --qemu --prebuilt 3
```

## 8.3 Booting with an Application

The following examples use the FSBL as the application being booted, however the command-line formats used in these examples are applicable to other standalone applications as well.
The FSBL is bundled with PetaLinux or Yocto BSPs.

> ⚠ QEMU does not model the DDRMC, for performance reasons. Because of this, using the FSBL can cause hangs in QEMU.
> See the known issues page for what to do if this problem occurs.

### 8.3.1 A53 Application (Zynq UltraScale+ MPSoC)

#### A53-0 FSBL in JTAG Mode

```
qemu-system-aarch64 \
-M arm-generic-fdt \
-m 4G \
-nographic \
-serial null \
-serial null \
-serial mon:stdio \
-hw-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 \
-m 4G \
-nographic \
-serial null \
-serial null \
-serial mon:stdio \
-hw-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 \
-m 4G \
-nographic \
-serial null \
-serial null \
-serial mon:stdio \
-hw-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
```

⚠ Default ZCU102 Petalinux design has QSPI in a dual parallel Configuration.

### A53-0 FSBL in SD0 Boot Mode

```
qemu-system-aarch64 \
-M arm-generic-fdt \
-m 4G \
-nographic \
-serial null \
-serial null \
-serial mon:stdio \
-hw-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
```

⚠ Default ZCU102 board supports SD1. Index should be set to 1 for SD drive argument.

## 8.3.2 A72 Application (Versal ACAP)

### A72-0 FSBL in JTAG Mode

```
qemu-system-aarch64
-M arm-generic-fdt \
-m 8G \
-nographic \
-serial null \
-serial null \
-serial mon:stdio \
-device loader,addr=0xFD1A0300,data=0x8000000e,data-len=4 \
-hw-dtb /path/to/dtb/versal-qemu-ps.dtb \
-kernel /path/to/image/image.elf
```

### 8.3.3 Zynq UltraScale+ MPSoC R5 Application

R5-0 FSBL in JTAG Mode

```
qemu-system-aarch64 \
-M arm-generic-fdt \
-m 4G \
-nographic \
-serial null \
-serial null \
-serial mon:stdio \
-hw-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 \
-m 4G \
-nographic \
-serial null \
-serial null \
-serial mon:stdio \
-hw-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 \
-m 4G \
-nographic \
-serial null \
-serial null \
-serial mon:stdio \
-hw-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 SD0 Boot Mode

```
qemu-system-aarch64 \
-M arm-generic-fdt \
-m 4G \
-nographic \
-serial null \
-serial null \
-serial mon:stdio \
-hw-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 \
-serial null \
-serial null \
-serial mon:stdio \
-hw-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
```

### 8.3.4 Versal ACAP R5 Application

#### R5-0 Application in JTAG Mode

```
/scratch/proj/qemu/build/aarch64-softmmu/qemu-system-aarch64 \
-M arm-generic-fdt \
-device loader,file=/path/to/image.elf,cpu-num=2 \
-device loader,addr=0xff5e0300,data=0x16,data-len=4 \
-device loader,addr=0xff9a0100,data=0x1,data-len=4 \
-device loader,addr=0xff9a0000,data=0x8,data-len=4 \
-serial null \
-serial null \
-serial stdio \
-display none \
-hw-dtb /path/to/dtb/board-versal-ps-virt.dtb
```

#### R5 Lockstep

```
/scratch/proj/qemu/build/aarch64-softmmu/qemu-system-aarch64 \
-M arm-generic-fdt \
-device loader,file=/path/to/image/image.elf,cpu-num=2 \
-device loader,addr=0xff5e0300,data=0x14,data-len=4 \
-device loader,addr=0xff9a0000,data=0x50,data-len=4 \
-device loader,addr=0xff9a0100,data=0x1,data-len=4 \
-serial null \
-serial null \
-serial stdio \
-display none \
-hw-dtb /path/to/dtb/board-versal-ps-virt.dtb
```

## 8.4 Terminal Commands

These are some of the terminal commands that can be used when QEMU is started with the `-nographic` option.

| Command | Description |
|---|---|
| `CTRL+A C` | Switch between the QEMU monitor and console. |
| `CTRL+A X` | Exit QEMU. |
| `CTRL+A H` | Prints the terminal command help message. |

## 8.5 QEMU Monitor Commands

This is a short list of useful QEMU monitor commands, for a full list, use the `help` and `help info` commands in the QEMU Monitor.
To get to the QEMU monitor, do `CTRL+A C` while in QEMU. To exit the QEMU monitor, do `CTRL+A C` while inside the QEMU monitor.

| Command | Description |
|---|---|
| `help` | Prints a list of monitor commands and a description of each one. |
| `help info` | Prints a list of info monitor commands and a description of each one. |
| `info qtree` | Prints the device tree. |
| `info mtree` | Prints the memory tree. |
| `info cpus` | Shows information for each CPU. |
| `info registers [-a]` | Shows the CPU registers.<br>Passing in -a shows register info for all CPUs. |
| `memsave <addr> <len> <file>` | Reads *len* bytes at memory address *addr* and saves it to file *file* as raw binary data.<br>See also: Linux Kernel Logbuf Extraction |
| `system_reset` | Resets the system. |
| `x /fmt <addr>` | Prints the memory at the virtual address *addr*, with format dictated by *fmt*. |
| `xp /fmt <addr>` | Prints the memory at the physical address *addr*, with format dictated by *fmt*. |
| stop | Stops emulation |

| Command | Description |
| --- | --- |
| c | Resumes emulation |
| q | Quits QEMU. |

## 8.6 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.

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]
```

The options are the same as the ones described in QEMU Loader Options.
The emulation can then be resumed (with the new memory and CPU state from the loading operations) using the `c` command:

```
(qemu) c
```

## 8.7 Linux Kernel Logbuf Extraction

Using the QEMU monitor command `memsave`, it is possible to extract the Linux kernel logbuf and read it.

### 8.7.1 Get the logbuf address and size

Use `readelf` to get the address and size of the logbuf.

```
$ readelf -a image.elf | grep __log_buf
     1: c1324c44 0x20000 OBJECT  LOCAL  DEFAULT   21 __log_buf
```

`image.elf` is the image you pass into QEMU that contains the Linux kernel whose logbuf you want to extract.

The numbers we care about in this case are `c1324c44` and `0x20000`, which are the virtual address and the size of the logbuf respectively.

### 8.7.2 Dump the logbuf from QEMU

In the QEMU monitor window, type:

```
(qemu) memsave 0xc1324c44 0x20000 dumpmem.logbuf
```

### 8.7.3 Read the contents

Parts of the logbuf will have binary data in it, which can be fixed by using a logbuf reader program, found here.

Compile the reader.

```
$ gcc logbuf-reader.c -o logbuf-reader
```

Then pipe the logbuf through the reader and `cat` it.

```
$ cat dumpmem.logbuf | logbuf-reader
Ramdisk addr 0x00000000,
FDT at 0x813ae998
Linux version 4.19.0-xilinx-v2019.2 (oe-user@oe-host) (gcc version 8.2.0 (GCC)) #1
Thu Oct 3 22:42:39 UTC 2019
setup_memory: max_mapnr: 0x7ffff
setup_memory: min_low_pfn: 0x80000
setup_memory: max_low_pfn: 0xb0000
setup_memory: max_pfn: 0xfffff
Zone ranges:
  DMA      [mem 0x0000000080000000-0x00000000afffffff]
  Normal   empty
  HighMem  [mem 0x00000000b0000000-0x00000000fffefff]
Movable zone start for each node
Early memory node ranges
  node   0: [mem 0x0000000080000000-0x00000000fffefff]
Initmem setup node 0 [mem 0x0000000080000000-0x00000000fffefff]
On node 0 totalpages: 524287
  DMA zone: 1536 pages used for memmap
  DMA zone: 0 pages reserved
  DMA zone: 196608 pages, LIFO batch:63
  HighMem zone: 327679 pages, LIFO batch:63
earlycon: uartlite_a0 at MMIO 0x40600000 (options '115200n8')
bootconsole [uartlite_a0] enabled
setup_cpuinfo: initialising
setup_cpuinfo: No PVR support. Using static CPU info from FDT
wt_msr
pcpu-alloc: s0 r0 d32768 u32768 alloc=1*32768
pcpu-alloc: [0] 0
Built 1 zonelists, mobility grouping on.  Total pages: 522751
Kernel command line: console=ttyUL0,115200 earlycon
Dentry cache hash table entries: 131072 (order: 7, 524288 bytes)
Inode-cache hash table entries: 65536 (order: 6, 262144 bytes)
Memory: 2059012K/2097148K available (4843K kernel code, 152K rwdata, 1380K rodata,
13129K init, 562K bss, 38136K reserved, 0K cma-reserved, 1310716K highmem)
Kernel virtual memory layout:
  * 0xfffea000..0xfffff000  : fixmap
  * 0xff800000..0xffc00000  : highmem PTEs
  * 0xff7ff000..0xff800000  : early ioremap
  * 0xf0000000..0xff7ff000  : vmalloc & ioremap
# ...
```

# 9 Debugging Guest Applications with QEMU and GDB

This section will cover how to debug a guest application with QEMU and GDB, and will cover different methods of debugging such as:

- Intrusive debugging (debugging so that when a breakpoint is hit, the kernel is paused as well)
- Non-intrusive debugging (debugging so that when a breakpoint is hit, the kernel is not paused)

> Since QEMU emulates the CPU, the breakpoints set by GDB are hardware breakpoints, not software breakpoints.



## 9.1 Differences Between Zynq UltraScale+ MPSoC and Versal ACAP

The examples on this page are done on the Zynq UltraScale+ MPSoC platform.

On Versal ACAP, the differences are:

- 2 ARM-A CPUs instead of 4

## 9.2 Acquiring the Tools

The tools mentioned in this guide are:

- `aarch64-none-elf-gdb`
- `aarch64-linux-gnu-gdb`

- `arm-none-eabi-gdb`
- `gdb-multiarch`
- `mb-gdb`

`aarch64-none-elf-gdb`, `aarch64-linux-gnu-gdb`, `arm-none-eabi-gdb`, and `mb-gdb` are bundled with PetaLinux, Yocto, and Vitis tools.
`gdb-multiarch` can be downloaded through your package manager (e.g. `apt-get`).

## 9.3 Kernel-Intrusive Debugging

Kernel-intrusive debugging allows you to debug the kernel or bare-metal image on your guest, and has the added benefit of being easier to set up.

The main disadvantage of intrusive debugging with QEMU is only being able to capture `SIGINT` and `SIGTRAP` signals.
This means if your program has a segmentation fault and a `SIGSEGV` is emitted, it will not be caught and your program will exit.

Intrusive debugging requires QEMU parameters to connect to its GDB server, and additional GDB commands in order to start a debugging session.

### 9.3.1 Enabling a GDB connection to QEMU

QEMU contains a GDB server that you can connect to, allowing you to debug your QEMU application.

To enable connection to the GDB server, you need to pass in a parameter to QEMU that specify the hostname and port it should listen on.

| QEMU parameter | Info |
|---|---|
| `-gdb tcp:<hostname>:<port>` | Makes QEMU's GDB server listen on host *hostname* on port *port*.<br>⚠ Generally the hostname is "localhost" and the port can be anything, as long as you can connect to it. |
| `-gdb tcp:<hostname>:<port> -S` | Makes QEMU's GDB server listen on host *hostname* on port *port* and makes emulation start in a paused state.<br>This can allow you to debug the boot sequence of your virtual machine.<br>⚠ To un-pause emulation, connect to QEMU using GDB and use the *continue* command. |
| `-s` | Shorthand for `-gdb tcp::1234`. |

```
-gdb tcp:localhost:9000
```

If booting QEMU using PetaLinux, the primary machine will typically listen on `localhost:9000`.
For example, if booting a ZCU102 machine using PetaLinux, the ARM machine will listen on `localhost:9000`, while the Microblaze machine will not have remote debugging enabled.

To simultaneously debug both MicroBlaze and ARM machines in a multi-arch environment with PetaLinux, use the `--pmu-qemu-args='-gdb tcp:<hostname>:<port>'` argument to enable debugging on the MicroBlaze QEMU machine.

## 9.3.2 Connecting GDB to QEMU

To connect GDB to QEMU, you need to use the GDB that corresponds to your target's architecture.

For example, some of the architectures in the Xilinx QEMU environments are:

| GDB | Architecture |
|---|---|
| `aarch64-none-elf-gdb`<br>`aarch64-linux-gnu-gdb` | ARM A53, ARM A72 |
| `arm-none-eabi-gdb` | ARM R5<br>⚠ `arm-none-eabi-gdb` may give a 'g' packet error when connecting to QEMU, depending if it is built to handle the `aarch64` architecture or not.<br>This is because QEMU will give register information for the first CPU cluster, which are the ARM A72 or ARM A53 CPUs.<br>If this happens, you can debug the ARM R5s with `gdb-multiarch`. |
| `gdb-multiarch` | Multiple ARM architectures, and others (excluding Microblaze) |
| `mb-gdb` | Microblaze |

In this guide, we will use `gdb-multiarch` to debug the ARM CPUs.

To connect to QEMU's GDB server using your host GDB, you need to create a remote connection.

Once you are connected, you can debug your emulated environment like you would debug any other program.

| GDB command | Info |
|---|---|
| `target remote <hostname>:<port>`<br>`target remote :<port>` | Attempts to connect to host *hostname* on port *port*.<br>If no hostname is specified, GDB will use *localhost*. |
| `target extended-remote <hostname>:<port>`<br>`target extended-remote :<port>` | Attempts to connect to host *hostname* on port *port*, and will remain connected after the debugged program exits or GDB detaches from it.<br>If no hostname is specified, GDB will use *localhost*. |

```
$ aarch64-none-elf-gdb hello_a53.elf
GNU gdb (Linaro GDB 2018.04) 8.0.50.20171128-git
Copyright (C) 2017 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=x86_64-unknown-linux-gnu --target=aarch64-none-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from /proj/xhdsswstaff/saipava/epdsw1/work/hello_a53.elf...done.
(gdb)
(gdb) target remote:9000
Remote debugging using :9000
_vector_table () at asm_vectors.S:208
208 b _boot
(gdb) b main
Breakpoint 1 at 0x1740: file ../src/main.c, line 75.
(gdb) c
Continuing.

Thread 1 "" hit Breakpoint 1, main () at ../src/main.c:75
75 BootStatus = XPm_GetBootStatus();
(gdb)
```

## 9.4 Non-Kernel-Intrusive Debugging

Rather than running GDB on the host machine and using the GDB server provided by QEMU, it's possible to run GDB on the guest machine as long as the guest supports it.

This can provide a series of advantages, such as:

- Only debugging your program (assuming you're not debugging your kernel)
- Being able to catch more signals, such as `SIGSEGV`
- Not losing control of GDB to other signals, such as `SIGTRAP`

The disadvantages are that it is more work to install GDB on the guest machine, assuming it can be installed at all, and it uses storage space on the guest.

### 9.4.1 Installing GDB on the Guest

For this example, we will run GDB on a Linux guest, on the 64-bit ARM-A CPUs on a Zynq UltraScale+ MPSoC QEMU machine.
The machine's image will be a default PetaLinux ZCU102 image, and we will install GDB by copying the files from the host using SCP.

1. Download the ARM64 GDB package from here and save it somewhere.
2. Unpack the package. This should produce the folders `DEBIAN`, `etc`, and `usr`.

```
$ sudo dpkg-deb -R gdb_7.12-6_arm64.deb .
```

3. Compress the `etc` and `usr` folders. Optionally delete the `DEBIAN`, `etc`, and `usr` folders when you are done. On a default PetaLinux image, `zip` and `tar` files are supported.

```
$ zip -r gdb.zip etc usr
```

4. On the guest, copy the compressed file via SCP and extract it.

```
root@xilinx-zcu102-2020_2:~# scp <host user>@<host IP>:/path/to/gdb.zip .
root@xilinx-zcu102-2020_2:~# unzip gdb.zip
```

5. On the guest, copy the extracted contents to root.

```
root@xilinx-zcu102-2020_2:~# cp -r etc usr /
```

### 9.4.2 Running GDB on the Guest

GDB can be run from the guest as if you would normally run it from the host.

If your binary and source files are not on the guest machine, copy them from the host before running GDB.

```
root@xilinx-zcu102-2020_2:~# scp <host user>@<host IP>:/scratch/proj/gdb-test/segfault.c .
<host user>@<host IP>'s password:
segfault.c                                                100%  171     0.2KB/s   00:00
root@xilinx-zcu102-2020_2:~# scp <host user>@<host IP>:/scratch/proj/gdb-test/segfault.elf .
<host user>@<host IP>'s password:
segfault.elf                                              100%   11KB  10.6KB/s   00:00
root@xilinx-zcu102-2020_2:~# gdb
gdb: /lib/libncurses.so.5: no version information available (required by gdb)
gdb: /lib/libncurses.so.5: no version information available (required by gdb)
gdb: /lib/libncurses.so.5: no version information available (required by gdb)
gdb: /lib/libtinfo.so.5: no version information available (required by gdb)
GNU gdb (Debian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "aarch64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) file segfault.elf
Reading symbols from segfault.elf...done.
(gdb) r
Starting program: /home/root/segfault.elf

Program received signal SIGSEGV, Segmentation fault.
0x00000000004005d8 in main () at segfault.c:8
8       *p_null = 0xAA55AA55;
(gdb)
```

## 9.5 GDB Commands

This section will cover GDB commands used regardless if GDB is used remotely or locally.

This section covers a small subset of what is available.  A full list can be found here.

### 9.5.1 Loading Symbols

GDB requires symbols from the program being executed, otherwise it won't know anything about the program and won't be able to debug.

You can still debug your application without symbols loaded, however it will be much more difficult.

| Command | Info |
|---|---|
| `--args <file.elf>` | Loads the symbols in *file.elf* into GDB<br>If *file.elf* does not contain debugging symbols, it must be recompiled with the *-g* flag passed into gcc.<br>This option is used when starting GDB, for example: `gdb --args file.elf` |
| `symbol-file <file.elf>` | Loads the symbols in *file.elf* into GDB.<br>If *file.elf* does not contain debugging symbols, it must be recompiled with the *-g* flag passed into gcc.<br>⚠ *symbol-file* can only store symbols from one file at a time. |
| `add-symbol-file <file.elf> <addr>` | Loads the symbols in *file.elf* into GDB.<br>If *file.elf* does not contain debugging symbols, it must be recompiled with the *-g* flag passed into gcc.<br>⚠ *addr* is the start address of the .text section in *file.elf*. To find the address of .text you can do:<br>`readelf -WS file.elf \| grep .text` |

```
(gdb) symbol-file test.elf
Reading symbols from test.elf...done.
```

### 9.5.2 Controlling Execution

| Command | Info |
|---|---|
| `r`<br>`run` | Starts execution |

| Command | Info |
| --- | --- |
| CTRL+C | Pauses execution. |
| c<br>continue | Continues execution. |
| kill | Kills the current process. |
| q<br>quit | Quits GDB. |

```
komlodi@machine:/scratch/gdb-test$ gdb test.elf
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from test.elf...done.
(gdb) r
Starting program: /scratch/gdb-test/test.elf
^C
Program received signal SIGINT, Interrupt.
main () at test.c:23
23       while(1);
(gdb) c
Continuing.
^C
Program received signal SIGINT, Interrupt.
main () at test.c:23
23       while(1);
(gdb) kill
Kill the program being debugged? (y or n) y
(gdb) q
komlodi@machine:/scratch/gdb-test$
```

### 9.5.3 Breakpoints and Watchpoints

Breakpoints allow you to pause execution at a specific point in your code. This allows you to observe program state at a specific point in time.

Watchpoints allow you to pause execution when the value of an expression changes. This can include a variable or address being modified, or a more complicated expression such as "var < 10".

| Command | Info |
|---|---|
| `info break` | Gives you information on all breakpoints in your GDB session. |
| `info watch` | Gives you information on all watchpoints in your GDB session. |
| `b <file:line-number\|function-name> [if <cond>]` | Adds a breakpoint at line *line-number* in *file* or at the start of function *function-name*.<br>If *cond* exists, the breakpoint will only trigger when *cond* is met.<br><br>The breakpoint number generated here is used when disabling or deleting a breakpoint. |
| `watch <var>`<br>`watch <addr>`<br>`watch <expr>` | Adds a watchpoint that pauses program when variable var or address addr are modified.<br>Watchpoints can have more complicated expressions with C-like syntax.<br><br>⚠ Watchpoints will trigger after the instruction is executed, meaning GDB will pause on the line after the watchpoint triggers.<br>In a non-local environment, GDB does not support *reverse-step* to see the exact line where the trigger occurred.<br><br>⚠ When using a watchpoint with an expression, GDB will evaluate the expression and pause when it is non-zero. |
| `disable <breakpoint-number>` | Disables breakpoint *breakpoint-number* |
| `enable <breakpoint-number>` | Enables breakpoint *breakpoint-number* |

| Command | Info |
|---|---|
| `d <breakpoint-number>` `delete <breakpoint-number>` | Deletes breakpoint *breakpoint-number* |

```
Reading symbols from test.elf...done.
(gdb) b main
Breakpoint 1 at 0x4005de: file test.c, line 20.
(gdb) watch global_var & 0x00000003
Hardware watchpoint 2: global_var & 0x00000003
(gdb) r
Starting program: /scratch/gdb-test/test.elf

Breakpoint 1, main () at test.c:20
20  {
(gdb) c
Continuing.

Hardware watchpoint 2: global_var & 0x00000003

Old value = 0
New value = 3
foo (s=0x7fffffffe1c0) at test.c:38
38      if (s->str == NULL) {
(gdb) p/x global_var
$1 = 0xcc33cc33
(gdb) info break
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000004005de in main at test.c:20
    breakpoint already hit 1 time
2       hw watchpoint  keep y                      global_var & 0x00000003
    breakpoint already hit 1 time
(gdb) info watch
Num     Type           Disp Enb Address            What
2       hw watchpoint  keep y                      global_var & 0x00000003
    breakpoint already hit 1 time
(gdb) d 1
(gdb) d 2
(gdb) info break
No breakpoints or watchpoints.
(gdb)
```

### 9.5.4 Stepping through your program

When paused, GDB allows you to control the program flow in a variety of ways.

| Command | Info |
|---|---|
| `s` | Steps through one line of your program.<br>This **will** go inside functions, including library functions. |
| `n` | Steps through one line of your program.<br>This **will not** go inside functions. |
| `si` | Steps through one instruction of your program.<br>This **will** go inside functions, including library functions. |
| `ni` | Steps through one instruction of your program.<br>This **will not** go inside functions. |
| `finish` | Executes until the end of the current function. |

```
Reading symbols from test.elf...done.
(gdb) b main
Breakpoint 1 at 0x4005de: file test.c, line 20.
(gdb) r
Starting program: /scratch/gdb-test/test.elf

Breakpoint 1, main () at test.c:20
20  {
(gdb) n
23      foo(&s);
(gdb) s
foo (s=0x7fffffffe1c0) at test.c:30
30      s->str = "This is a string";
(gdb) n
31      memset(s->arr, 0xAA, sizeof(s->arr));
(gdb) n
32      s->var = 1234;
(gdb) c
Continuing.
```

## 9.5.5 Stack and frame information

| Command | Info |
|---|---|
| `backtrace` | Prints the call stack, in the order that functions were called, as a list of frames |

| Command | Info |
| --- | --- |
| `frame <frame-number>` | Changes the current frame being observed to *frame-number*. This does not modify program execution. |
| `info frame` | Gives you information on the current frame. |
| `info variables` | Gives you information of all static/global variables and symbols in your program.<br>⚠ On some systems, this may print a lot of information. |
| `info locals` | Gives you information on local variables in the current frame. |
| `info args` | Gives you information on arguments passed into the current frame. |

```
Breakpoint 1, foo (s=0x7fffffffe1c0) at test.c:30
30        s->str = "This is a string";
(gdb) backtrace
#0  foo (s=0x7fffffffe1c0) at test.c:30
#1  0x00000000004005f9 in main () at test.c:23
(gdb) frame 0
#0  foo (s=0x7fffffffe1c0) at test.c:30
30        s->str = "This is a string";
(gdb) info frame
Stack level 0, frame at 0x7fffffffe1c0:
 rip = 0x400620 in foo (test.c:30); saved rip = 0x4005f9
 called by frame at 0x7fffffffe200
 source language c.
 Arglist at 0x7fffffffe1b0, args: s=0x7fffffffe1c0
 Locals at 0x7fffffffe1b0, Previous frame's sp is 0x7fffffffe1c0
 Saved registers:
  rbp at 0x7fffffffe1b0, rip at 0x7fffffffe1b8
(gdb) info locals
No locals.
(gdb) info args
s = 0x7fffffffe1c0
(gdb) frame 1
#1  0x00000000004005f9 in main () at test.c:23
23        foo(&s);
(gdb) info locals
s = {var = 0, arr = "\000\000\000\000\000\000\000",
  str = 0x4006b0 <__libc_csu_init> "AWAVA\211\377AUATL\215%N\a ", c = -32
'\340', num = 0}
(gdb) info args
No arguments.
(gdb)
```

## 9.5.6 Printing Variables

gdb allows you to print out variable information using expressions. The expressions use C-like syntax.

| **Command** | **Info** |
|---|---|
| `print <var>`<br>`p <var>`<br>`p <file>::<var>`<br>`p '<file>'::<file>` | Prints *var*.<br>By default, GDB will print var based off of what type it is. |
| `p *<arr>@<len>` | Prints the first *len* values of *arr*. |
| `p/x <var>` | Prints *var* as a hexadecimal number |

| Command | Info |
|---|---|
| p/d <var> | Prints *var* as a signed integer. |
| p/t <var> | Prints *var* as a binary number. |
| p/c <var> | Prints *var* as a character. |
| ptype <var><br>ptype <type> | Prints type definition of *var*, or the type definition of *type*. This is useful when getting information of a struct. |

```
Reading symbols from test.elf...done.
(gdb) b foo
Breakpoint 1 at 0x400620: file test.c, line 30.
(gdb) r
Starting program: /scratch/gdb-test/test.elf

Breakpoint 1, foo (s=0x7fffffffe1c0) at test.c:30
(gdb) ptype s
type = volatile struct {
    uint32_t var;
    uint8_t arr[8];
    char *str;
    char c;
} *
30      s->str = "This is a string";
(gdb) n
31      memset(s->arr, 0xAA, sizeof(s->arr));
(gdb) n
32      s->c = 'z';
(gdb) n
33      s->var = 1;
(gdb) n
34      global_var = 0xCC33CC33;
(gdb) p s->str
$1 = 0x400734 "This is a string"
(gdb) p *s->str@4
$2 = "This"
(gdb) p/c s->str[0]
$3 = 84 'T'
(gdb) p/x s->arr
$4 = {0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa, 0xaa}
(gdb) p/t s->arr
$5 = {10101010, 10101010, 10101010, 10101010, 10101010, 10101010,
10101010, 10101010}
(gdb) p/d s->var
$7 = 1
```

### 9.5.7 Modifying Variables

| Command | Info |
|---|---|
| `set <var>=<val>` | Sets the variable *var* to the value *val*. |

```
foo (s=0x7ffffffe1d0) at test.c:38
38      if (s->str == NULL) {
(gdb) p s->str
$1 = 0x400734 "This is a string"
(gdb) set s->str=0x00
(gdb) n
39          puts("s contains a NULL string");
```

### 9.5.8 Macros

GDB provides commands to expand, list, and define macros.
In this case, 'macro' refers to any preprocessor definition, not just one that takes arguments. For example, each of the following lines would be considered a macro:

```
#define REG_FIELD_0_MASK    0x01
#define REG_FIELD_1_MASK    (1 << 1)
#define REG_FIELD_X_MASK(x) (1 << x)
```

> ⚠ Most compilers don't create macro information by default. For example, on gcc, you would create macro information by passing in the `-g3` flag at compile time.

| Command | Info |
|---|---|
| `macro expand <expr>`<br>`macro exp <expr>` | Expands, but does not parse, all preprocessor macros in *expr*. |
| `info macro [-a\|-all] <macro>` | Shows the current definition of macro *macro*, and where it was declared.<br>If the `-a` flag is passed, all definitions of *macro* will be shown. |
| `info macros <function>` | Shows all macros that are currently defined in *function*.<br>ⓘ This will also print **all** macro definitions defined in *function*, including ones from standard libraries. |

```
Breakpoint 1, main () at gdb-macro.c:13
13      for (i=0; i<sizeof(reg) * 8; ++i) {
(gdb) n
14              printf("bit %.2d: %d\n", i, REG_BIT_EXTRACT(reg, i));
(gdb) info macro REG_BIT_EXTRACT
Defined at /scratch/test-code/gdb-macro.c:5
#define REG_BIT_EXTRACT(reg, bit) (!!GET_BIT(reg, bit))
(gdb) info macro GET_BIT
Defined at /scratch/test-code/gdb-macro.c:4
#define GET_BIT(val, bit) (val & (1 << bit))
(gdb) macro exp REG_BIT_EXTRACT(reg, i)
expands to: (!!(reg & (1 << i)))
(gdb) macro exp REG_BIT_EXTRACT(0xAA55AA55, 0)
expands to: (!!(0xAA55AA55 & (1 << 0)))
(gdb) p REG_BIT_EXTRACT(0xAA55AA55, 0)
$1 = 1
```

### 9.5.9 Signals

When debugging your code it may be useful to manage signals, particularly if they keep interrupting your GDB session.

> ⚠ QEMU GDB server is only capable of handling SIGINT and SIGTRAP signals. Meaning if your program receives a SIGSEGV signal, it will not be passed to GDB.

| **Command** | **Info** |
|---|---|
| `info signals [sig]` | Prints information on all symbols, or only signal *sig* if specified |
| `catch signal [sig1 sig2 \| all ]` | Sets a catchpoint at signals *sig1* and *sig2*, or all signals if *all* is specified. |

| Command | Info |
|---|---|
| `handle sig [nostop stop print noprint noignore ignore]` | Handles signal *sig* depending on the following keywords:<br>• nostop<br>GDB should not stop your program when this signal happens<br>• stop<br>GDB should stop your program when this signal happens<br>• print<br>GDB should print when this signal happens<br>• noprint<br>GDB should not print when this signal happens<br>• noignore<br>GDB should allow your program to see this signal<br>• ignore<br>GDB should not allow your program to see this signal |

```
(gdb) symbol-file myapp
Reading symbols from myapp...done.
(gdb) b main
Breakpoint 6 at 0x4008a4: file myapp.c, line 85.
(gdb) c
Continuing.
[Switching to Thread 1.1]

Thread 1 hit Breakpoint 6, main () at myapp.c:85
85 puts("Program start");
(gdb) n

Thread 1 received signal SIGTRAP, Trace/breakpoint trap.
0xffffff800810f814 in ?? ()
(gdb)
Cannot find bounds of current function
(gdb)
Cannot find bounds of current function
(gdb) c
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
0xffffff800809c088 in ?? ()
(gdb) info signals SIGTRAP
Signal Stop Print Pass to program Description
SIGTRAP Yes Yes Yes Trace/breakpoint trap
(gdb) handle SIGTRAP nostop print
SIGTRAP is used by the debugger.
Are you sure you want to change it? (y or n) y
Signal Stop Print Pass to program Description
SIGTRAP No Yes Yes Trace/breakpoint trap
(gdb) c
Continuing.
[Switching to Thread 1.2]

Thread 2 hit Breakpoint 6, main () at myapp.c:85
85 puts("Program start");
(gdb) n

Thread 2 received signal SIGTRAP, Trace/breakpoint trap.
# Execution resumes
```

## 9.5.10 Threads

| **Command** | **Info** |
| --- | --- |
| `info threads` | Gives you information on all current threads in the program.<br>The current thread of execution will have an asterisk (*) by it. |
| `thread <thread-ID>` | Changes the current thread to *thread-ID,* and prints out the current frame in *thread-ID* |
| `set print thread-events <on\|off>` | Prints when a thread is created or deleted. |
| `show print thread-events` | Shows if thread events are printed or not |
| `thread apply [thread-ids \| all] <cmd>` | Applies *cmd* to threads *thread-ids* or to all threads if *all* is specified. |

```
(gdb) info threads
  Id   Target Id         Frame
* 1    Thread 1.1 (Cortex-A72 #0 [running]) 0xffffff800809c088 in ?? ()
  2    Thread 1.2 (Cortex-A72 #1 [running]) 0xffffff800809c088 in ?? ()
(gdb) thread 2
[Switching to thread 2 (Thread 1.2)]
#0  0xffffff800809c088 in ?? ()
(gdb) thread 1
[Switching to thread 1 (Thread 1.1)]
#0  0xffffff800809c088 in ?? ()
(gdb) set print thread-events
(gdb) show print thread-events
Printing of thread events is on.
(gdb) thread apply all catch signal all

Thread 2 (Thread 1.2):
Catchpoint 1 (any signal)

Thread 1 (Thread 1.1):
Catchpoint 2 (any signal)
(gdb)
```

## 9.5.11 Debugging Multiple Processes

In some situations you may want to debug multiple processes simultaneously, or a process that already exists but GDB does not have immediate knowledge of.

On platforms with multiple ARM architectures, these same commands can be used to switch between the different ARM processors.

| Command | Info |
|---|---|
| `attach <id>` | Attaches to a running process with ID *id*.<br>⚠ When remote debugging, you must target gdbserver using the `extended-remote` command in order to attach to a new process. |
| `add-inferior` | Adds an executable to the current debug session |
| `inferior <inf>` | Makes inferior *inf* the current inferior to be debugged by GDB |
| `info inferior` | Prints a list of all inferiors currently managed by GDB. |
| `set print inferior-events <on\|off>` | Enables or disables printing messages when GDB notices new inferiors have started or stopped.<br>By default inferior events are not printed. |
| `show print inferior-events` | Show if inferior event messages are printed or not. |

```
(gdb) target extended-remote :9000
Remote debugging using :9000
0xffffff8008112eb4 in ?? ()
(gdb) info thread
  Id   Target Id         Frame
* 1    Thread 1.1 (Cortex-A53 #0 [running]) 0xffffff8008112eb4 in ?? ()
  2    Thread 1.2 (Cortex-A53 #1 [running]) 0xffffff8008112eb4 in ?? ()
  3    Thread 1.3 (Cortex-A53 #2 [running]) 0xffffff8008112eb4 in ?? ()
  4    Thread 1.4 (Cortex-A53 #3 [running]) 0xffffff8008113440 in ?? ()
(gdb) add-inferior
Added inferior 2
(gdb) inferior 2
[Switching to inferior 2 [<null>] (<noexec>)]
(gdb) attach 2
Attaching to process 2
[New Thread 2.6]
0xffff0000 in ?? ()
(gdb) info thread
  Id   Target Id         Frame
  1.1  Thread 1.1 (Cortex-A53 #0 [running]) 0xffffff8008112eb4 in ?? ()
  1.2  Thread 1.2 (Cortex-A53 #1 [running]) 0xffffff8008112eb4 in ?? ()
  1.3  Thread 1.3 (Cortex-A53 #2 [running]) 0xffffff8008112eb4 in ?? ()
  1.4  Thread 1.4 (Cortex-A53 #3 [running]) 0xffffff8008113440 in ?? ()
* 2.1  Thread 2.5 (Cortex-R5 #0 [halted ]) 0xffff0000 in ?? ()
  2.2  Thread 2.6 (Cortex-R5 #1 [halted ]) 0xffff0000 in ?? ()
(gdb) info inferior
  Num  Description       Executable
  1    process 1
* 2    process 2
(gdb) inferior 1
[Switching to inferior 1 [process 1] (<noexec>)]
[Switching to thread 1.1 (Thread 1.1)]
#0  0xffffff8008112eb4 in ?? ()
(gdb) set print inferior-events on
(gdb) show print inferior-events
Printing of inferior events is on.
(gdb) c
Continuing.
```

To simplify this process you can add a function to GDB. To do this edit your .gdbinit file (located in your home directory) and add the following lines:

```
define rdo_arm
target extended-remote :9000
add-inferior
inferior 2
attach 2
info threads
end
```

This will allow you to use the command `rdo_arm` to connect to QEMU's GDB server and automatically discover the RPU.

### 9.5.12 Lower Level Examining

| Command | Info |
|---|---|
| `info registers` | Prints out your CPU's registers. |
| `x/x <addr>` | Prints out the value at *addr* interpreted as hex. |
| `x/4x <addr>` | Prints out the first 4 values at *addr* interpreted as hex. |
| `x/s <addr>` | Prints out the value at *addr* interpreted as a string. |
| `disassemble <addr>`<br>`disassemble <function-name>` | Prints out the disassembly at address *addr* or at function *function-name*. |

```
Reading symbols from test.elf...done.
(gdb) b foo
Breakpoint 1 at 0x400620: file test.c, line 32.
(gdb) r
Starting program: /scratch/gdb-test/test.elf

Breakpoint 1, foo (s=0x7fffffffe1c0) at test.c:32
32        s->str = "This is a string";
(gdb) n
33        memset(s->arr, 0xAA, sizeof(s->arr));
(gdb) n
34        s->var = 1234;
(gdb) n
36        global_var = 0xCC33CC33;
(gdb) n
38        if (s->str == NULL) {
(gdb) x/x &global_var
0x601054 <global_var>:  0xcc33cc33
(gdb) x/4x &global_var
0x601054 <global_var>:  0xcc33cc33  0x00000000  0x00000000  0x00000000
(gdb) x/s s->str
0x400724:    "This is a string"
(gdb) info registers
rax            0x7fffffffe1c0   140737488347584
rbx            0x0  0
rcx            0x0  0
rdx            0x8  8
rsi            0x7fffffffe1cc   140737488347596
rdi            0x7fffffffe1c4   140737488347588
rbp            0x7fffffffe1b0   0x7fffffffe1b0
rsp            0x7fffffffe1a0   0x7fffffffe1a0
r8             0x400710 4196112
r9             0x7ffff7de7ab0   140737351940784
r10            0x34e    846
r11            0x7ffff7b7f970   140737349417328
r12            0x4004e0 4195552
r13            0x7fffffffe2d0   140737488347856
r14            0x0  0
r15            0x0  0
rip            0x40065a 0x40065a <foo+70>
eflags         0x246    [ PF ZF IF ]
cs             0x33 51
ss             0x2b 43
ds             0x0  0
es             0x0  0
fs             0x0  0
gs             0x0  0
k0             0x0  0
k1             0x0  0
k2             0x0  0
k3             0x0  0
```

```
k4             0x0  0
k5             0x0  0
k6             0x0  0
k7             0x0  0
(gdb) n
42      disassemble_me();
(gdb) disassemble disassemble_me
Dump of assembler code for function disassemble_me:
   0x0000000000400679 <+0>: push   %rbp
   0x000000000040067a <+1>: mov    %rsp,%rbp
   0x000000000040067d <+4>: movq   $0x0,-0x8(%rbp)
   0x0000000000400685 <+12>:    mov    -0x8(%rbp),%rax
   0x0000000000400689 <+16>:    movzbl (%rax),%eax
   0x000000000040068c <+19>:    mov    %al,-0x9(%rbp)
   0x000000000040068f <+22>:    nop
   0x0000000000400690 <+23>:    pop    %rbp
   0x0000000000400691 <+24>:    retq
End of assembler dump.
```

## 9.6 Debugging Examples

### 9.6.1 Zynq UltraScale+ MPSoC and Versal ACAP PS + PMU simultaneous debugging

To debug The PS and PMU simultaneously you need to boot up the Microblaze and ARM64 QEMU instances with the -gdb flag.
Once the virtual machines are started, you can remotely target them with two separate instances of GDB.

> ⚠ Make sure the ports you pass in to each QEMU instance are different from each other.

> ⓘ Most QEMU arguments are omitted for brevity. A list of example arguments used when booting a Zynq UltraScale+ MPSoC or Versal ACAP machine can be found here.

```
# Most QEMU arguments omitted for brevity
qemu-system-microblazeel \
-M microblaze-fdt \
# 8<-----------------------------------
-gdb tcp:127.0.0.1:9090 -S

qemu-system-aarch64 \
-M arm-generic-fdt \
# 8<-----------------------------------
-gdb tcp:127.0.0.1:9000
```

Connect mb-gdb to Microblaze QEMU

```
$ mb-gdb
GNU gdb (crosstool-NG 1.20.0) 8.2.1.20190121-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/
gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-build_unknown-linux-gnu --
target=microblaze-xilinx-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) target remote :9090
Remote debugging using :9090
warning: No executable has been specified and target does not support
determining executable automatically.  Try using the "file" command.
0x0000d0ff in ?? ()
(gdb) c
Continuing.
```

Connect `gdb-multiarch` to AArch64 QEMU and discover the ARM-R processors.

```
 gdb-multiarch
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) target extended-remote :9000
Remote debugging using :9000
0xffffff8008112eb4 in ?? ()
(gdb) info thread
  Id   Target Id         Frame
* 1    Thread 1.1 (Cortex-A53 #0 [running]) 0xffffff8008112eb4 in ?? ()
  2    Thread 1.2 (Cortex-A53 #1 [running]) 0xffffff8008112eb4 in ?? ()
  3    Thread 1.3 (Cortex-A53 #2 [running]) 0xffffff8008112eb4 in ?? ()
  4    Thread 1.4 (Cortex-A53 #3 [running]) 0xffffff8008113440 in ?? ()
(gdb) add-inferior
Added inferior 2
(gdb) inferior 2
[Switching to inferior 2 [<null>] (<noexec>)]
(gdb) attach 2
Attaching to process 2
[New Thread 2.6]
0xffff0000 in ?? ()
(gdb) info thread
  Id   Target Id         Frame
  1.1  Thread 1.1 (Cortex-A53 #0 [running]) 0xffffff8008112eb4 in ?? ()
  1.2  Thread 1.2 (Cortex-A53 #1 [running]) 0xffffff8008112eb4 in ?? ()
  1.3  Thread 1.3 (Cortex-A53 #2 [running]) 0xffffff8008112eb4 in ?? ()
  1.4  Thread 1.4 (Cortex-A53 #3 [running]) 0xffffff8008113440 in ?? ()
* 2.1  Thread 2.5 (Cortex-R5 #0 [halted ]) 0xffff0000 in ?? ()
  2.2  Thread 2.6 (Cortex-R5 #1 [halted ]) 0xffff0000 in ?? ()
(gdb) info inferior
  Num  Description       Executable
  1    process 1
* 2    process 2
```

Now you are able to debug the Microblaze, ARM-A, and ARM-R CPUs simultaneously.

## 9.7 Related articles

http://www.yolinux.com/TUTORIALS/GDB-Commands.html
https://sourceware.org/gdb/current/onlinedocs/gdb/index.html#SEC_Contents
https://sourceware.org/gdb/onlinedocs/gdb/Concept-Index.html

# 10 Debugging Guest Applications with QEMU, XSDB, and XSCT

This page will cover the commands that can be used when debugging a guest application with QEMU and XSDB. Some content from debugging with GDB will be restated here for convenience.

Some commands will be omitted from this page; the full documentation for XSDB can be found here.



## 10.1 Differences Between Zynq UltraScale+ MPSoC and Versal ACAP

The examples on this page are done on the Zynq UltraScale+ MPSoC platform.

On Versal ACAP, the differences are:

- 2 ARM-A72 CPUs instead of 4 ARM-A53 CPUs.

## 10.2 Acquiring the Tools

XSDB is bundled with Vitis and can be downloaded here.

## 10.3 Enabling an XSDB connection to QEMU

QEMU contains a GDB server that you can connect to, allowing you to debug your QEMU application.

To enable connection to the GDB server, you need to pass in a parameter to QEMU that specify the hostname and port it should listen on.

| QEMU parameter | Details |
|---|---|
| -gdb tcp:<hostname>:<port> | Makes QEMU's GDB server listen on host *hostname* on port *port*.<br>⚠ Generally the hostname is "localhost" and the port can be anything, as long as you can connect to it. |

| QEMU parameter | Details |
| --- | --- |
| -gdb tcp:<hostname>:<port> -S | Makes QEMU's GDB server listen on host *hostname* on port *port* and makes emulation start in a paused state.<br>This can allow you to debug the boot sequence of your virtual machine.<br>⚠ To un-pause emulation, connect to QEMU using GDB and use the *continue* command. |

```
-gdb tcp:localhost:9000
```

If booting QEMU using Petalinux, the primary machine will typically listen on `localhost:9000`.
For example, if booting a ZCU102 machine using Petalinux, the ARM machine will listen on `localhost:9000`, while the Microblaze machine will not have remote debugging enabled.

To simultaneously debug both Microblaze and ARM machines in a multi-arch environment, you must build QEMU from source.
Once built from source, pass in the `-gdb` argument for each machine when booting QEMU.

## 10.4 Connecting XSDB to QEMU

| XSDB Command | Details |
| --- | --- |
| `gdbremote connect <hostname>:<port>`<br>`gdbremote connect :<port>` | Connects to a GDB remote server with host *hostname* and port *port*.<br>If hostname is left blank, it will connect to *localhost.* |
| `gdbremote disconnect` | Disconnects from a GDB remote server. |
| `exit` | Exits XSDB |

```
xsdb% gdbremote connect :9000
attempting to launch tcfgdbclient
xsdb% Info: Cortex-A53 #0 (target 3) Stopped at 0xffffff80086824a0
(Suspended)
xsdb% Info: Cortex-A53 #1 (target 4) Stopped at 0xffffff8008112eb4
(Suspended)
xsdb% Info: Cortex-A53 #2 (target 5) Stopped at 0xffffff8008112eb4
(Suspended)
xsdb% Info: Cortex-A53 #3 (target 6) Stopped at 0xffffff8008112eb4
(Suspended)
xsdb% Info: Cortex-R5 #0 (target 8) Stopped at 0xffff0000 (Suspended)
xsdb% Info: Cortex-R5 #1 (target 9) Stopped at 0xffff0000 (Suspended)
```

## 10.5 Loading Debugging Symbols

XSDB requires symbols from the program being executed, otherwise it won't know anything about the program and won't be able to debug.

| XSDB Command | Details |
|---|---|
| `memmap -file <file.elf>` | Loads the symbols from *file.elf* into XSDB.<br>If *file.elf* does not contain debugging symbols, it must be recompiled with the *-g* flag passed into gcc. |

```
xsdb% memmap -file test-arm.elf
xsdb%
```

## 10.6 Connecting to a Target

| XSDB Command | Details |
|---|---|
| `target [id]`<br>`ta [id]` | Lists all available targets.<br>If *id* is specified, XSDB connects to target *id*. |

```
xsdb% ta
  1  GdbClient (127.0.0.1:9000)
     2  p1
        3* Cortex-A53 #0 (Suspended)
        4  Cortex-A53 #1 (Suspended)
        5  Cortex-A53 #2 (Suspended)
        6  Cortex-A53 #3 (Suspended)
     7  p2
        8  Cortex-R5 #0 (Suspended)
        9  Cortex-R5 #1 (Suspended)
xsdb% ta 3
```

## 10.7 Controlling Execution

| XSDB Command | Details |
|---|---|
| `state` | Gives the current execution state. |
| `stop` | Stops execution. |
| `con` | Resumes execution. |
| `stp [count]` | Steps through one line of your program.<br>If count is specified, it will go step through *count* lines.<br>This **will** go inside functions, including library functions. |
| `nxt [count]` | Steps through one line of your program.<br>If count is specified, it will go step over *count* lines.<br>This **will not** go inside functions. |
| `stpi [count]` | Steps through one instruction of your program.<br>If count is specified, it will go step through *count* lines.<br>This **will** go inside functions, including library functions. |
| `nxti [count]` | Steps through one instruction of your program.<br>If count is specified, it will go step over *count* lines.<br>This **will not** go inside functions. |
| `stpout [count]` | Executes until the end of the current function.<br>If count is specified, this will be repeated *count* times. |

```
xsdb% Info: Cortex-A53 #0 (target 3) Stopped at 0x400730 (Breakpoint)
main() at test.c: 22
22: {
xsdb% Info: Cortex-A53 #1 (target 4) Stopped at 0xffffff8008802be8
(Suspended)
xsdb% Info: Cortex-A53 #2 (target 5) Stopped at 0xffffff8008112eb4
(Suspended)
xsdb% Info: Cortex-A53 #3 (target 6) Stopped at 0xffffff8008101aac
(Suspended)
xsdb% Info: Cortex-R5 #0 (target 8) Stopped at 0xffff0000 (Suspended)
xsdb% Info: Cortex-R5 #1 (target 9) Stopped at 0xffff0000 (Suspended)
xsdb% nxt
Info: Cortex-A53 #0 (target 3) Stopped at 0x40074c (Step)
25:     foo(&s);
Info: Cortex-A53 #1 (target 4) Stopped at 0xffffff800809da98 (Suspended)
Info: Cortex-A53 #2 (target 5) Stopped at 0xffffff80087d82a8 (Suspended)
Info: Cortex-A53 #3 (target 6) Stopped at 0xffffff8008103fe8 (Suspended)
Info: Cortex-R5 #0 (target 8) Stopped at 0xffff0000 (Suspended)
Info: Cortex-R5 #1 (target 9) Stopped at 0xffff0000 (Suspended)
xsdb% stp
Info: Cortex-A53 #0 (target 3) Stopped at 0x400780 (Breakpoint)
foo() at test.c: 31
31: {
Info: Cortex-A53 #1 (target 4) Stopped at 0xffffff800809da98 (Suspended)
Info: Cortex-A53 #2 (target 5) Stopped at 0xffffff80087d82a8 (Suspended)
Info: Cortex-A53 #3 (target 6) Stopped at 0xffffff8008103fe8 (Suspended)
Info: Cortex-R5 #0 (target 8) Stopped at 0xffff0000 (Suspended)
Info: Cortex-R5 #1 (target 9) Stopped at 0xffff0000 (Suspended)
xsdb% nxt 3
Info: Cortex-A53 #0 (target 3) Stopped at 0x4007b0 (Step)
34:     s->var = 1234;
Info: Cortex-A53 #1 (target 4) Stopped at 0xffffff8008113ed8 (Suspended)
Info: Cortex-A53 #2 (target 5) Stopped at 0xffffff80087d7a90 (Suspended)
Info: Cortex-A53 #3 (target 6) Stopped at 0xffffff8008113458 (Suspended)
Info: Cortex-R5 #0 (target 8) Stopped at 0xffff0000 (Suspended)
Info: Cortex-R5 #1 (target 9) Stopped at 0xffff0000 (Suspended)
xsdb% nxt
Info: Cortex-A53 #0 (target 3) Stopped at 0x4007bc (Step)
36:     global_var = 0xCC33CC33;
Info: Cortex-A53 #1 (target 4) Stopped at 0xffffff80087d7954 (Suspended)
Info: Cortex-A53 #2 (target 5) Stopped at 0xffffff80087d7b10 (Suspended)
Info: Cortex-A53 #3 (target 6) Stopped at 0xffffff8008113458 (Suspended)
Info: Cortex-R5 #0 (target 8) Stopped at 0xffff0000 (Suspended)
Info: Cortex-R5 #1 (target 9) Stopped at 0xffff0000 (Suspended)
xsdb% stpout
Info: Cortex-A53 #0 (target 3) Stopped at 0x400754 (Step)
main() at test.c: 27
27:     return 0;
Info: Cortex-A53 #1 (target 4) Stopped at 0xffffff800809da98 (Suspended)
Info: Cortex-A53 #2 (target 5) Stopped at 0xffffff8008112eb4 (Suspended)
Info: Cortex-A53 #3 (target 6) Stopped at 0xffffff8008112eb4 (Suspended)
```

```
Info: Cortex-R5 #0 (target 8) Stopped at 0xffff0000 (Suspended)
Info: Cortex-R5 #1 (target 9) Stopped at 0xffff0000 (Suspended)
xsdb% con
Info: Cortex-A53 #0 (target 3) Running
Info: Cortex-A53 #1 (target 4) Running
Info: Cortex-A53 #2 (target 5) Running
Info: Cortex-A53 #3 (target 6) Running
Info: Cortex-R5 #0 (target 8) Running
Info: Cortex-R5 #1 (target 9) Running
xsdb% stop
Info: Cortex-A53 #0 (target 3) Stopped at 0xffffff80080ced88 (Suspended)
Info: Cortex-A53 #1 (target 4) Stopped at 0xffffff8008802c00 (Suspended)
Info: Cortex-A53 #2 (target 5) Stopped at 0xffffff80080817c8 (Suspended)
Info: Cortex-A53 #3 (target 6) Stopped at 0xffffff8008112eb4 (Suspended)
Info: Cortex-R5 #0 (target 8) Stopped at 0xffff0000 (Suspended)
Info: Cortex-R5 #1 (target 9) Stopped at 0xffff0000 (Suspended)
xsdb%
```

## 10.8 Breakpoints and Watchpoints

<table>
<tr><th>XSDB Command</th><th>Details</th></tr>
<tr><td>bpadd <-addr <addr> | -file <name> -line <lineno>> [ -target-id <id> -type <type>|-mode <mode> | -enable <mode>]</td><td>Adds a breakpoint to either address addr or line lineno in file name, depending if the -addr or -file option is provided.<br><br>⚠ addr can also be the address of a function.<br><br>If -target-id is specified, id corresponds to a target ID.<br>To add a breakpoint that targets all targets, use all as the target ID.<br><br>If -type is specified, type can be one of the following values:<br>• auto - The breakpoint type is determined by the hw_server or TCF agent. (default)<br>• hw - hardware breakpoint<br>• sw - software breakpoint<br>If -mode is specified, mode is a bitfield that consists of the following values:<br>• 0x01 - Triggered by a read from the breakpoint location<br>• 0x02 - Triggered by a write to the breakpoint location<br>• 0x04 - Triggered by an instruction execution at the breakpoint location (default)<br>• 0x08 - Triggered by a data change at the breakpoint location<br>If -enable is specified, mode can be one of the following values:<br>• 0 - The breakpoint is disabled<br>• 1 - The breakpoint is enabled (default)</td></tr>
</table>

| XSDB Command | Details |
|---|---|
| `bpremove <breakpoints> \| -all` | Removes each breakpoint in the list *breakpoints*.<br>If -*all* is specified, all breakpoints are removed. |
| `bpenable <breakpoints> \| -all` | Enables each breakpoint in the list *breakpoints*.<br>If -*all* is specified, all breakpoints are enabled. |
| `bpdisable <breakpoints> \| -all` | Disables each breakpoint in the list *breakpoints*.<br>If -*all* is specified, all breakpoints are disabled. |
| `bplist` | Lists all breakpoints along with the status of each breakpoint. |
| `bpstatus <id>` | Prints the status of breakpoint *id*. |

```
xsdb% gdbremote connect :9000
attempting to launch tcfgdbclient
xsdb% Info: Cortex-A53 #0 (target 3) Stopped at 0xffffff8008112eb4 (Suspended)
xsdb% Info: Cortex-A53 #1 (target 4) Stopped at 0xffffff8008112eb4 (Suspended)
xsdb% Info: Cortex-A53 #2 (target 5) Stopped at 0xffffff8008112eb4 (Suspended)
xsdb% Info: Cortex-A53 #3 (target 6) Stopped at 0xffffff8008112eb4 (Suspended)
xsdb% Info: Cortex-R5 #0 (target 8) Stopped at 0xffff0000 (Suspended)
xsdb% Info: Cortex-R5 #1 (target 9) Stopped at 0xffff0000 (Suspended)
xsdb% ta 3
xsdb% memmap -file test.elf
xsdb% bpadd -addr main
0
xsdb% Info: Breakpoint 0 status:
   target 3: {Address: 0x400730 Type: Hardware}
xsdb% con
Info: Cortex-A53 #0 (target 3) Running
Info: Cortex-A53 #1 (target 4) Running
Info: Cortex-A53 #2 (target 5) Running
Info: Cortex-A53 #3 (target 6) Running
Info: Cortex-R5 #0 (target 8) Running
Info: Cortex-R5 #1 (target 9) Running
xsdb% Info: Cortex-A53 #0 (target 3) Stopped at 0x400730 (Breakpoint)
main() at test.c: 22
22: {
xsdb% Info: Cortex-A53 #1 (target 4) Stopped at 0xffffff8008112eb4 (Suspended)
xsdb% Info: Cortex-A53 #2 (target 5) Stopped at 0xffffff8008112eb4 (Suspended)
xsdb% Info: Cortex-A53 #3 (target 6) Stopped at 0xffffff8008112eb4 (Suspended)
xsdb% Info: Cortex-R5 #0 (target 8) Stopped at 0xffff0000 (Suspended)
xsdb% Info: Cortex-R5 #1 (target 9) Stopped at 0xffff0000 (Suspended)
xsdb%
```

## 10.9 Stack and Frame Information

| XSDB Command | Details |
|---|---|
| `locals [-defs\|-dict] [var [val]]` | Returns the values of all local variables.<br>If *var* is specified, the value of local variable *var* is returned.<br>If *var* and *val* are specified, local variable *var* is set to *val*.<br>If *-defs* is specified, the definition (type, size, address, RW flags) of the locals are returned.<br>If *-dict* is specified, the local variables are returned in Tcl dict format, with variable names as dict keys and values as dict values. |
| `backtrace` | Prints the call stack, in the order that functions were called, as a list of frames. |

```
xsdb% backtrace
    0  0x400730 main(): test.c, line 22
    1  0x7f7fa90ce4
xsdb% locals
s             : <Structure>
   var        : 4196440
   arr        : uint8_t[8]
   str        : 549723915792
   p          : 547602631848
   c          : 88
   num        : 0.0
xsdb%
```

## 10.10 Printing and Modifying Variables

<table>
<tr><th>XSDB command</th><th>Details</th></tr>
<tr><td><code>print [-add|-defs|-dict|-remove|-set <var>] <expr></code></td><td>
Prints the expression expr. expr can be a single variable or multiple variables combined with operators in a way that's syntactically valid.
If -add is specified, expr is added to the auto expression list. Expressions in the auto expression list are printed every time print is called.
If -defs is specified, the definition (type, size, address, RW flags) of expr is returned.
If -dict is specified, the result of the expression is returned in Tcl dict format, with variable names as dict keys and values as dict values.
If -remove is specified, an expression that was previously added to the auto expression list via add is removed.
If -set is specified, var is set to the value of expr.
</td></tr>
<tr><td><code>mrd [-force|-size <access-size>|-value|-bin -file <name>|-address-space <name>|-unaligned-access] <addr> [num]</code></td><td>
Prints 1 word from address addr.
If num is specified, num values are printed.
If -force is specified, access protection is overrided, allowing access to reserved and invalid address ranges.
If -size is specified, the amount of data read is determined by access-size, where access-size is one of the following:
<ul>
<li>b - Read a byte</li>
<li>h - Read a half word</li>
<li>w - Read a word (default)</li>
<li>d - Read a double word</li>
</ul>
If -value is specified, a Tcl list of values is returned.
If -bin is specified, the data is written in binary format to the file name on the host machine.
If -address-space is specified, the address space name is accessed instead of the default address space. For ARM DAP targets, the address spaces are as follows:
<ul>
<li>DPR - DP registers</li>
<li>APR - AP registers</li>
<li>AP<n> - MEM-AP<n> registers</li>
</ul>
If unaligned-access is specified, memory access is not aligned to access size.
</td></tr>
</table>

<table>
<tr><th>XSDB command</th><th>Details</th></tr>
<tr><td>mwr [-force|-size<br><access-size>|-bin<br>-file <name>|<br>-address-space <name>|-<br>unaligned-access]<br><addr> <values> [num]</td><td>Writes a list of values values to address addr sequentially.<br><br>If num is specified, num values are written.<br><br>If -force is specified, access protection is overrided, allowing access to reserved and invalid address ranges.<br><br>If -size is specified, the amount of data written is determined by access-size, where access-size is one of the following:<br><br>• b - Read a byte<br>• h - Read a half word<br>• w - Read a word (default)<br>• d - Read a double word<br>If -bin is specified, the data is read from file name and written to addr in binary format.<br><br>If -address-space is specified, the address space name is accessed instead of the default address space.  For ARM DAP targets, the address spaces are as follows:<br><br>• DPR - DP registers<br>• APR - AP registers<br>• AP<n> - MEM-AP<n> registers<br>If unaligned-access is specified, memory access is not aligned to access size.</td></tr>
</table>

```
xsdb% nxt
Info: Cortex-A53 #0 (target 3) Stopped at 0x40079c (Step)
33:     memset(s->arr, 0xAA, sizeof(s->arr));
Info: Cortex-A53 #1 (target 4) Stopped at 0xffffff80087d6a58 (Suspended)
Info: Cortex-A53 #2 (target 5) Stopped at 0xffffff80080d4a40 (Suspended)
Info: Cortex-A53 #3 (target 6) Stopped at 0xffffff8008082280 (Suspended)
Info: Cortex-R5 #0 (target 8) Stopped at 0xffff0000 (Suspended)
Info: Cortex-R5 #1 (target 9) Stopped at 0xffff0000 (Suspended)
xsdb% nxt
Info: Cortex-A53 #0 (target 3) Stopped at 0x4007b0 (Step)
34:     s->var = 1234;
Info: Cortex-A53 #1 (target 4) Stopped at 0xffffff80087d7b84 (Suspended)
Info: Cortex-A53 #2 (target 5) Stopped at 0xffffff80080992c8 (Suspended)
Info: Cortex-A53 #3 (target 6) Stopped at 0xffffff80080df010 (Suspended)
Info: Cortex-R5 #0 (target 8) Stopped at 0xffff0000 (Suspended)
Info: Cortex-R5 #1 (target 9) Stopped at 0xffff0000 (Suspended)
xsdb% nxt
Info: Cortex-A53 #0 (target 3) Stopped at 0x4007bc (Step)
36:     global_var = 0xCC33CC33;
Info: Cortex-A53 #1 (target 4) Stopped at 0xffffff80087d7b84 (Suspended)
Info: Cortex-A53 #2 (target 5) Stopped at 0xffffff80080992c8 (Suspended)
Info: Cortex-A53 #3 (target 6) Stopped at 0xffffff80080df024 (Suspended)
Info: Cortex-R5 #0 (target 8) Stopped at 0xffff0000 (Suspended)
Info: Cortex-R5 #1 (target 9) Stopped at 0xffff0000 (Suspended)
xsdb% print s->var
s->var    : 1234
xsdb% print s->arr
s->arr    : uint8_t[8]

xsdb% print -defs s->arr
Name      Type                Address             Size      Flags
========  ========            ===========         ====      =====
s->arr    uint8_t[8]          0x7ff62aa424        8         RW

xsdb% mrd 0x7ff62aa424 2
      7FF62AA424:   AAAAAAAA
      7FF62AA428:   AAAAAAAA
```

## 10.11 Lower Level Examining

| XSDB command | Details |
|---|---|
| `rrd [-defs\|-no-bits] [reg]` | Reads all registers.<br>If *reg* is provided, only register *reg* is read.<br>If *-defs* is specified the register definitions are read instead of the values.<br>If *-no-bits* is specified, bit f ields are not shown along with the register values. |

| XSDB command | Details |
|---|---|
| `rwr <reg> <val>` | Writes value *val* to register *reg*. |
| `dis [addr] [num]` | Disassembles 1 instruction at the current PC value.<br>If *addr* and *num* are specified, *num* instructions are decoded at *addr*.<br>The keyword *pc* can be used instead of an address to disassemble *num* instructions at *pc*. |

```
xsdb% dis pc 10
ffffff8008112ec4: ldr     w1, [x22]
ffffff8008112ec8: cmp     w1, w21
ffffff8008112ecc: b.ne    -13      ; addr=0xffffff8008112e98
ffffff8008112ed0: sub     x0, x0, x6
ffffff8008112ed4: mov     w2, w2
ffffff8008112ed8: and     x0, x0, x5
ffffff8008112edc: ldp     x19, x20, [sp, #16]
ffffff8008112ee0: mul     x0, x0, x2
ffffff8008112ee4: ldp     x21, x22, [sp, #32]
ffffff8008112ee8: lsr     x0, x0, x4

xsdb% rrd
       x0: 0000000247ad3bf0
       x1: ffffffc87ff6c040
       x2: 000000003d89d8a
       x3: 00000000009d7053
       x4: 0000000000000016
       x5: 00ffffffffffffff
       x6: 0000000006334e7e
       x7: 0000000000000012
       x8: ffffffc87ff6b2d0
       x9: ffffffc87ff6b2b0
      x10: 071c71c71c71c71c
      x11: 00000000000136f9
      x12: 0000000000000044
      x13: 00000000000001d6
      x14: 00000000000000ff
      x15: 0000000000000400
      x16: 0000000000000000
      x17: 000000001fd9034
      x18: 0000000000000400
      x19: ffffff80145dd008
      x20: 0000000000000000
      x21: 0000000000000004
      x22: ffffff80145dd000
      x23: ffffff80145dd008
      x24: 0000000000000028
      x25: ffffffc86e8e4000
      x26: 0000000000000000
      x27: ffffffc86e8e4400
      x28: 0000000001050018
      x29: ffffff80145c3e50
      x30: ffffff8008112eb4
       sp: ffffff80145c3e50
       pc: ffffff8008112ec4
# ...
xsdb% rwr pc 0xffffff8008112ec0
```

# 11 Example Development Flow

This page will contain brief examples of how to compile and load your program into QEMU, and then debug it.



## 11.1 Acquiring the Tools

For this example you will need:

- `aarch64-linux-gnu-gcc`
- `gdb-multiarch` or `aarch64-linux-gnu-gdb`
- device tree compiler (`dtc`)
- `scp` (on the guest)

`aarch64-linux-gnu-gcc`, `aarch64-linux-gnu-gdb`, and `dtc` are bundled with PetaLinux, Yocto, or Vitis tools. `scp` is available on the guest by default if using these tools as well.

## 11.2 Compiling Your Application

For this example, we're going to use an example bare metal program called `myapp`, and we're going to build it for the ARM64 application processing unit (APU) on a ZCU102 development board.

`myapp.c` can be found here.

This application transmits data through UART0 and has a simple menu with three options:

1. Generate data to be printed out to the UART
2. Print out data to the UART
3. Exit

For compiling it, we'll use AArch64 gcc and include debugging symbols.

```
aarch64-linux-gnu-gcc -g -Wall myapp.c -o myapp.elf
```

## 11.3 Loading Your Application

There are a few ways to load your application into the guest image.
For this page we will use the SSH method.

### 11.3.1 TFTP

Files can be copied between the host and guest by using TFTP.
Remember that the gateway IP between the guest and host is `10.0.2.2`.

```
root@xilinx-zcu102-2020_2:~# tftp -g -r myapp.elf 10.0.2.2
root@xilinx-zcu102-2020_2:~# chmod 755 myapp.elf
root@xilinx-zcu102-2020_2:~# ls -la
total 20
drwx------    2 root     root            60 Dec  2 19:29 .
drwxr-xr-x    3 root     root            60 Aug 28 05:26 ..
-rwxr-xr-x    1 root     root         18656 Dec  2 19:29 myapp.elf
```

See also: File Transfer with TFTP.

### 11.3.2 SSH

After booting into Linux, you can copy your application from the host machine to the guest by using SSH.

First, find the IP of the host machine

```
$ ifconfig
eth0 Link encap:Ethernet HWaddr <MAC addr>
inet addr:<host IP> Bcast:<Bcast IP> Mask:255.255.252.0
UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
RX packets:29112513 errors:0 dropped:0 overruns:0 frame:0
TX packets:112423727 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:16274872729 (16.2 GB) TX bytes:163292974163 (163.2 GB)
Memory:90100000-9017ffff

lo Link encap:Local Loopback
inet addr:127.0.0.1 Mask:255.0.0.0
UP LOOPBACK RUNNING MTU:65536 Metric:1
RX packets:31093619 errors:0 dropped:0 overruns:0 frame:0
TX packets:31093619 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1
RX bytes:2366150233 (2.3 GB) TX bytes:2366150233 (2.3 GB)
```

Then on the guest machine, use `scp` to copy the file from the host to the guest.

```
root@xilinx-zcu102-2020_2:~# scp <your host machine username>@<your host
machine IP>:/scratch/doc-example/myapp.elf .

Host '<host IP>' is not in the trusted hosts file.
(ecdsa-sha2-nistp256 fingerprint sha1!!
18:7e:92:d0:33:ed:97:e7:cb:b2:f7:b1:5d:52:5f:a6:34:9a:97:f9)
Do you want to continue connecting? (y/n) y
<host user>@<host IP>'s password:
myapp.elf                                                   100%   18KB
17.7KB/s   00:00
root@xilinx-zcu102-2020_2:~#
```

See also: File Transfer with SSH.

### 11.3.3 Loading your Application to the Guest Image Using PetaLinux

To add your application to the guest by using PetaLinux, it first must be part of a PetaLinux project.

#### Adding an Application to the RootFS of a PetaLinux Project

```
$ cd <petalinux-project-root>
$ petalinux-create -t apps --template install --name myapp --enable
```

This will create a new application that can be added to your PetaLinux project.

The application can be found in: `<petalinux-project-root>/project-spec-meta-user/recipes-apps/<your-application-name>`.

Change directory to the `files` directory and remove the existing `myapp` application; then copy your application to the current directory.

#### Building the Application

```
$ petalinux-config -c rootfs
# apps -> ensure your application is checked.
$ petalinux-build
```

The newly built image will appear under the `<petalinux-project-root>/images/linux/` directory on the host.

Your application will appear in `/usr/bin/<your-application-name>` in the guest after booting your image.

For more information on the above steps, see chapter 8 in the PetaLinux Tools Reference Guide.

## 11.4 Kernel-Intrusive Application Debugging

Now that `myapp.elf` is on the guest machine, let's see if it works.

When we debug `myapp.elf`, we will use GDB on the host machine and connect it to QEMU's GDB server.
This means that we are debugging the QEMU image (the Linux Kernel), along with our application.

More details on intrusive debugging with GDB can be found here.

```
root@xilinx-zcu102-2020_2:~# ./myapp.elf

***************UART TX MENU***************
g: Generate new data to transmit
t: Transmit the UART data
<RETURN>: Exit

g
Gen: 71 a7 5e c9 c7 67 82 8f a7 1d 8b 7a 31 44 ad f5
TX->71
TX->a7
TX->5e
TX->c9
TX->c7
TX->67
TX->82
TX->8f
TX->a7
TX->1d
TX->8b
TX->7a
TX->31
TX->44
TX->ad
TX->f5
g
Gen: 81 4d 91 4b 57 9d b8 08 a1 3d 19 8a ff 11 59 70
TX->81
TX->4d
TX->91
TX->4b
TX->57
TX->9d
TX->b8
TX->08
TX->a1
TX->3d
TX->19
TX->8a
TX->ff
TX->11
TX->59
TX->70
```

It mostly works. It looks like when we generate the UART data it's transmitted right away.

Let's debug it.

```
komlodi@machine:/scratch/doc-example$ gdb-multiarch
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/
gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) symbol-file myapp.elf
Reading symbols from myapp.elf...done.
(gdb) b 103
Breakpoint 1 at 0x400a80: file myapp.c, line 103.
(gdb) target remote :9000
Remote debugging using :9000
0xffffff8008112eb4 in ?? ()
(gdb) c
Continuing.
```

Re-run `myapp.elf` on the guest.

```
root@xilinx-zcu102-2020_2:~# ./myapp.elf

***************UART TX MENU***************
g: Generate new data to transmit
t: Transmit the UART data
<RETURN>: Exit
```

Now our GDB running on the host should hit the breakpoint in the while loop.

```
Breakpoint 1 at 0x400a40: file myapp.c, line 103.
(gdb) c
Continuing.
[Switching to Thread 1.2]

Thread 2 hit Breakpoint 1, prog_loop (uart=0x7f8c055000) at myapp.c:103
103 while (!done) {
(gdb) n
104 fgets(buf, sizeof(buf), stdin);
(gdb)
[Switching to Thread 1.4]

106 switch(buf[0]) {
(gdb)
108 printf("Gen:");
(gdb)

Thread 4 received signal SIGTRAP, Trace/breakpoint trap.
```

This is a perfect situation of when we want to ignore a signal. GDB received a SIGTRAP and lost control of the program.

Let's make sure this doesn't happen again.

```
^C
Thread 4 received signal SIGINT, Interrupt.
0xffffff80087d7b20 in ?? ()
(gdb) thread apply all handle SIGTRAP nostop ignore

Thread 4 (Thread 1.4):
SIGTRAP is used by the debugger.
Are you sure you want to change it? (y or n) y
Signal Stop Print Pass to program Description
SIGTRAP No Yes No Trace/breakpoint trap

Thread 3 (Thread 1.3):
SIGTRAP is used by the debugger.
Are you sure you want to change it? (y or n) y
Signal Stop Print Pass to program Description
SIGTRAP No Yes No Trace/breakpoint trap

Thread 2 (Thread 1.2):
SIGTRAP is used by the debugger.
Are you sure you want to change it? (y or n) y
Signal Stop Print Pass to program Description
SIGTRAP No Yes No Trace/breakpoint trap

Thread 1 (Thread 1.1):
SIGTRAP is used by the debugger.
Are you sure you want to change it? (y or n) y
Signal Stop Print Pass to program Description
SIGTRAP No Yes No Trace/breakpoint trap
(gdb) c
Continuing.
```

Now let's try that again.

```
[Switching to Thread 1.3]

Thread 3 hit Breakpoint 2, prog_loop (uart=0x7f8c055000) at myapp.c:106
106         switch(buf[0]) {
(gdb)
Continuing.

Thread 3 hit Breakpoint 2, prog_loop (uart=0x7f8c055000) at myapp.c:106
106         switch(buf[0]) {
(gdb) n
108                 printf("Gen:");
(gdb)
109                 for (i=0; i<sizeof(buf); ++i) {
(gdb)
110                     buf[i] = rand() & 0xFF;
(gdb)
111                     printf(" %.2x", buf[i]);
(gdb)
109                 for (i=0; i<sizeof(buf); ++i) {
(gdb)
110                     buf[i] = rand() & 0xFF;
(gdb)
111                     printf(" %.2x", buf[i]);
(gdb)
109                 for (i=0; i<sizeof(buf); ++i) {
(gdb) b 113
Breakpoint 3 at 0x400adc: file myapp.c, line 113.
(gdb) c
Continuing.
[Switching to Thread 1.2]

Thread 2 hit Breakpoint 3, prog_loop (uart=0x7f8c055000) at myapp.c:113
113                 puts("");
(gdb) n
115                 uart_tx(uart, buf, sizeof(buf));
(gdb)
```

So after we generate the data, it goes right to the transmit case statement.  Let's double check the switch statement.

```
switch(cmd[0]) {
    case 'g':
        printf("Gen:");
        for (i=0; i<sizeof(buf); ++i) {
            buf[i] = rand() & 0xFF;
            printf(" %.2x", buf[i]);
        }
        puts("");
    case 't':
        uart_tx(uart, buf, sizeof(buf));
        break;
    case '\n':
        done = true;
        break;
    default:
        printf("Unknown command %s", cmd);
        break;
}
```

We forgot to put a `break` at the end of the `'g'` case statement, so our code fell through to the next case.

This is an easy fix.

```
switch(cmd[0]) {
    case 'g':
        printf("Gen:");
        for (i=0; i<sizeof(buf); ++i) {
            buf[i] = rand() & 0xFF;
            printf(" %.2x", buf[i]);
        }
        puts("");
        break; // <--
    case 't':
        uart_tx(uart, buf, sizeof(buf));
        break;
    case '\n':
        done = true;
        break;
    default:
        printf("Unknown command %s", cmd);
        break;
}
```

Now let's put our new code on the guest machine, reload our GDB symbols, and run it.

```
komlodi@machine:/scratch/doc-example$ aarch64-linux-gnu-gcc -g -Wall
myapp.c -o myapp.elf
```

```
^C
Thread 2 received signal SIGINT, Interrupt.
0xffffff80080cedf4 in ?? ()
(gdb) symbol-file myapp.elf
Load new symbol table from "myapp.elf"? (y or n) y
Reading symbols from myapp.elf...done.
(gdb) info b
Num Type Disp Enb Address What
1 breakpoint keep y 0x0000000000400a80 in prog_loop at myapp.c:106
breakpoint already hit 5 times
2 breakpoint keep y 0x0000000000400a40 in prog_loop at myapp.c:102
breakpoint already hit 1 time
3 breakpoint keep y 0x0000000000400ae8 in prog_loop at myapp.c:113
breakpoint already hit 1 time
(gdb) d 1 2 3
(gdb) c
```

```
root@xilinx-zcu102-2020_2:~# scp <host user>@<host IP>:/scratch/doc-example/myapp.elf .
<host user>@1<host IP's password:
myapp.elf                                            100%   18KB   18.2KB/s   00:00
root@xilinx-zcu102-2020_2:~# ./myapp.elf

***************UART TX MENU***************
g: Generate new data to transmit
t: Transmit the UART data
<RETURN>: Exit

g
Gen: 0f 08 45 3e a8 34 72 37 8d 98 8c ae 87 6a 77 e5
t
TX->0f
TX->08
TX->45
TX->3e
TX->a8
TX->34
TX->72
TX->37
TX->8d
TX->98
TX->8c
TX->ae
TX->87
TX->6a
TX->77
TX->e5
g
Gen: 17 69 69 b3 1c f2 46 3b 62 af 08 39 ac 4b c4 bb
t
TX->17
TX->69
TX->69
TX->b3
TX->1c
TX->f2
TX->46
TX->3b
TX->62
TX->af
TX->08
TX->39
TX->ac
TX->4b
TX->c4
TX->bb
```

There, that looks better.

## 11.5 Non-Kernel-Intrusive Application Debugging

For this example we will load another application called `myapp-segfault.c`, which can be found here.
It can be loaded using any of the methods outlined in Loading Your Application, so we will not repeat that here.

> ⚠ In this case, we need to load the source file(s) onto the QEMU guest as well, otherwise GDB will not be able to display what line we're on.

When we debug something non-intrusively in QEMU, we have GDB on the QEMU machine.
Debugging in this way behaves exactly like it would as if you were debugging a program locally on a Linux machine.

> ⚠ This means you cannot debug your kernel using this method.
>
> More information on non-intrusive application debugging can be found here.

First, let's load GDB onto our machine. We'll use `scp` in this example and assume we've already downloaded an ARM64 GDB package.

```
komlodi@machine:/scratch/development-example$ sudo dpkg-deb -R gdb_7.12-6_arm64.deb .
komlodi@machine:/scratch/development-example$ sudo zip -r gdb.zip etc usr
```

Then on the QEMU guest:

```
root@xilinx-zcu102-2020_2:~# scp <host user>@<host IP>:/scratch/development-example/gdb.zip .
root@xilinx-zcu102-2020_2:~# unzip gdb.zip
root@xilinx-zcu102-2020_2:~# cp -rv etc usr /
```

Now let's run `myapp-segfault.c` and see if it works.

```
root@xilinx-zcu102-2020_2:~# ./myapp-segfault.elf

***************UART TX MENU***************
g: Generate new data to transmit
t: Transmit the UART data
<RETURN>: Exit

g
Segmentation fault
root@xilinx-zcu102-2020_2:~# ./myapp-segfault.elf

***************UART TX MENU***************
g: Generate new data to transmit
t: Transmit the UART data
<RETURN>: Exit

t
TX->50
TX->72
TX->6f
TX->67
TX->72
TX->61
TX->6d
TX->20
TX->73
TX->74
TX->61
TX->72
TX->74
TX->21
TX->0a
g
Segmentation fault
root@xilinx-zcu102-2020_2:~#
```

It looks like we consistently have a segmentation fault whenever we generate new data.

Let's debug it.

```
root@xilinx-zcu102-2020_2:~# gdb
gdb: /lib/libncurses.so.5: no version information available (required by gdb)
gdb: /lib/libncurses.so.5: no version information available (required by gdb)
gdb: /lib/libncurses.so.5: no version information available (required by gdb)
gdb: /lib/libtinfo.so.5: no version information available (required by gdb)
GNU gdb (Debian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "aarch64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb) file myapp-segfault.elf
Reading symbols from myapp-segfault.elf...done.
(gdb) r
Starting program: /home/root/myapp-segfault.elf

***************UART TX MENU***************
g: Generate new data to transmit
t: Transmit the UART data
<RETURN>: Exit

g

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400c68 in prog_loop (uart=0x7fbf6fb000) at myapp-segfault.c:89
89                      buf[i] = rand() & 0xFF;
```

We segfault when accessing `buf`, let's look closer.

```
(gdb) backtrace
#0  0x0000000000400c68 in prog_loop (uart=0x7fbf6fb000) at myapp-
segfault.c:89
#1  0x0000000000400d88 in main () at myapp-segfault.c:120
(gdb) info locals
i = 0
buf = 0x400e68 "Program start!\n"
cmd = "g\n\000\000\000\000\000\000\230\r@\000\000\000\000"
done = false
```

`buf` isn't `NULL`, but the address it points to is very close to execution memory.

Since `"Program start!\n"` is a constant, it's most likely going to reside in a read-only section of memory. Let's verify that.

```
(gdb) maintenance info sections
Exec file:
    `/home/root/myapp-segfault.elf', file type elf64-littleaarch64.
 [0]      0x00400200->0x0040021b at 0x00000200: .interp ALLOC LOAD READONLY DATA HAS_CONTENTS
 [1]      0x0040021c->0x0040023c at 0x0000021c: .note.ABI-tag ALLOC LOAD READONLY DATA HAS_CONTENTS
 [2]      0x0040023c->0x00400260 at 0x0000023c: .note.gnu.build-id ALLOC LOAD READONLY DATA HAS_CONTENTS
 [3]      0x00400260->0x00400308 at 0x00000260: .gnu.hash ALLOC LOAD READONLY DATA HAS_CONTENTS
 [4]      0x00400308->0x004004d0 at 0x00000308: .dynsym ALLOC LOAD READONLY DATA HAS_CONTENTS
 [5]      0x004004d0->0x0040059a at 0x000004d0: .dynstr ALLOC LOAD READONLY DATA HAS_CONTENTS
 [6]      0x0040059a->0x004005c0 at 0x0000059a: .gnu.version ALLOC LOAD READONLY DATA HAS_CONTENTS
 [7]      0x004005c0->0x00400600 at 0x000005c0: .gnu.version_r ALLOC LOAD READONLY DATA HAS_CONTENTS
 [8]      0x00400600->0x00400648 at 0x00000600: .rela.dyn ALLOC LOAD READONLY DATA HAS_CONTENTS
 [9]      0x00400648->0x004007c8 at 0x00000648: .rela.plt ALLOC LOAD READONLY DATA HAS_CONTENTS
 [10]     0x004007c8->0x004007dc at 0x000007c8: .init ALLOC LOAD READONLY CODE HAS_CONTENTS
 [11]     0x004007e0->0x00400900 at 0x000007e0: .plt ALLOC LOAD READONLY CODE HAS_CONTENTS
 [12]     0x00400900->0x00400e14 at 0x00000900: .text ALLOC LOAD READONLY CODE HAS_CONTENTS
 [13]     0x00400e14->0x00400e24 at 0x00000e14: .fini ALLOC LOAD READONLY CODE HAS_CONTENTS
 [14]     0x00400e28->0x00400f1f at 0x00000e28: .rodata ALLOC LOAD READONLY DATA HAS_CONTENTS
 [15]     0x00400f20->0x00400f24 at 0x00000f20: .eh_frame ALLOC LOAD READONLY DATA HAS_CONTENTS
 [16]     0x00411de0->0x00411de8 at 0x00001de0: .init_array ALLOC LOAD DATA HAS_CONTENTS
 [17]     0x00411de8->0x00411df0 at 0x00001de8: .fini_array ALLOC LOAD DATA HAS_CONTENTS
 [18]     0x00411df0->0x00411df8 at 0x00001df0: .jcr ALLOC LOAD DATA HAS_CONTENTS
 [19]     0x00411df8->0x00411fd8 at 0x00001df8: .dynamic ALLOC LOAD DATA HAS_CONTENTS
 [20]     0x00411fd8->0x00411fe8 at 0x00001fd8: .got ALLOC LOAD DATA HAS_CONTENTS
 [21]     0x00411fe8->0x00412080 at 0x00001fe8: .got.plt ALLOC LOAD DATA HAS_CONTENTS
 [22]     0x00412080->0x00412090 at 0x00002080: .data ALLOC LOAD DATA HAS_CONTENTS
 [23]     0x00412090->0x004120a8 at 0x00002090: .bss ALLOC
```

```
 [24]     0x00000000->0x0000003b at 0x00002090: .comment READONLY
HAS_CONTENTS
 [25]     0x00000000->0x00000030 at 0x000020cb: .debug_aranges READONLY
HAS_CONTENTS
 [26]     0x00000000->0x000004c7 at 0x000020fb: .debug_info READONLY
HAS_CONTENTS
 [27]     0x00000000->0x0000016c at 0x000025c2: .debug_abbrev READONLY
HAS_CONTENTS
 [28]     0x00000000->0x00000153 at 0x0000272e: .debug_line READONLY
HAS_CONTENTS
 [29]     0x00000000->0x00000120 at 0x00002888: .debug_frame READONLY
HAS_CONTENTS
 [30]     0x00000000->0x000002b5 at 0x000029a8: .debug_str READONLY
HAS_CONTENTS
```

On line `[14]`, we can see the address that `buf` points to, `0x400e68`, resides in that range. That section is called `.rodata`, and is labeled as `READONLY`. This is why we receive a segmentation fault when we write to it.

There are a few ways to fix this, the most straightforward change would be to make `buf` an array and copy `"Program Start!\n"` to it before the while loop.

```c
static void prog_loop(void *uart)
{
    size_t i;
//    char *buf = "Program start!\n";
    char buf[16]; // <--
    char cmd[16];
    bool done;

    strcpy(buf, "Program start!\n"); // <--

    while (!done) {
        fgets(cmd, sizeof(cmd), stdin);

        switch(cmd[0]) {
            case 'g':
                printf("Gen:");
                for (i=0; i<strlen(buf); ++i) {
                    buf[i] = rand() & 0xFF;
                    printf(" %.2x", buf[i]);
                }
                puts("");
                break;
            case 't':
                uart_tx(uart, buf, strlen(buf));
                break;
            case '\n':
                done = true;
                break;
            default:
                printf("Unknown command %s", cmd);
                break;
        }
    }
}
```

Now let's compile our changes, put the new binary on the guest, and run it.

```
komlodi@machine:/scratch/development-example$ aarch64-linux-gnu-gcc myapp-segfault.c -Wall -g -o myapp-segfault.elf
```

On the guest:

```
root@xilinx-zcu102-2020_2:~# scp <host user>@<host IP>:/scratch/
development-example/myapp-segfault.elf .
<host user>@<host IP>'s password:
myapp-segfault.elf                                            100%    18KB
17.7KB/s   00:00
root@xilinx-zcu102-2020_2:~# ./myapp-segfault.elf

***************UART TX MENU***************
g: Generate new data to transmit
t: Transmit the UART data
<RETURN>: Exit

t
TX->50
TX->72
TX->6f
TX->67
TX->72
TX->61
TX->6d
TX->20
TX->73
TX->74
TX->61
TX->72
TX->74
TX->21
TX->0a
g
Gen: c9 ca 06 e6 9b 8b aa c2 d7 69 3d fb 59 27 5f
t
TX->c9
TX->ca
TX->06
TX->e6
TX->9b
TX->8b
TX->aa
TX->c2
TX->d7
TX->69
TX->3d
TX->fb
TX->59
TX->27
TX->5f

root@xilinx-zcu102-2020_2:~#
```

## 11.6 QEMU Module Debug Printing

This section gives a brief example showing how to make QEMU modules print debug information.
A more detailed page showing how to do QEMU module debug printing is available here.

If building QEMU from source, QEMU provides a way to enable debug printing for modules.

Now let's say you wanted to see the register reads and writes of the UART.
To do this we need to find the UART module that QEMU is using and then enable debug printing.

### 11.6.1 Finding the Module

To find out what UART module QEMU is using, we will look in the device tree files.
If you have access to the DTS files, look in those. Otherwise, unflatten the DTB and look at the DTS output.
For this example, we'll unflatten the DTB.

```
komlodi@machine:/scratch/petalinux-images/xilinx-zcu102-2019.2/pre-built/linux/images$ dtc -I dtb -O dts system.dtb -o system.dts
komlodi@machine:/scratch/petalinux-images/xilinx-zcu102-2019.2/pre-built/linux/images$ vim system.dts
```

We're working on a Zynq UltraScale+ MPSoC, so UART0 is at address 0xFF000000.

```
1803            serial@ff000000 {
1804                u-boot,dm-pre-reloc;
1805                compatible = "cdns,uart-r1p12", "xlnx,xuartps";
1806                status = "okay";
1807                interrupt-parent = <0x4>;
1808                interrupts = <0x0 0x15 0x4>;
1809                reg = <0x0 0xff000000 0x0 0x1000>;
1810                clock-names = "uart_clk", "pclk";
1811                power-domains = <0xc 0x21>;
1812                clocks = <0x3 0x38 0x3 0x1f>;
1813                pinctrl-names = "default";
1814                pinctrl-0 = <0x1d>;
1815                cts-override;
1816                device_type = "serial";
1817                port-number = <0x0>;
1818            };
```

Here's the UART in the device tree. What we care about are the values of the `compatible` string.
These values are used by QEMU to determine what module should be used to model the hardware.
QEMU will scan the compatible strings from left to right, and use the first one it is capable of modeling.

It isn't immediately obvious what a `cdns,uart-r1p12` is, aside from it being a UART. Let's look for that.

```
komlodi@machine:/scratch/proj/qemu/build$ find .. -name "*uart*"
../hw/riscv/sifive_uart.c
../hw/char/lm32_juart.c
../hw/char/cmsdk-apb-uart.c
../hw/char/omap_uart.c
../hw/char/grlib_apbuart.c
../hw/char/xilinx_iomod_uart.c
../hw/char/exynos4210_uart.c
../hw/char/milkymist-uart.c
../hw/char/nrf51_uart.c
../hw/char/mcf_uart.c
../hw/char/xilinx_uartlite.c
../hw/char/lm32_uart.c
../hw/char/cadence_uart.c
../hw/char/digic-uart.c
```

cadence_uart.c looks like the most likely file, let's look at that one.

> ⚠ In most situations you can use `grep` on the compatible string and find the module that way.

## 11.6.2 Enabling Module Debug Printing

Most modules will have a sequence of debug code at the top of the file that will look something like this:

```
#ifdef CADENCE_UART_ERR_DEBUG
#define DB_PRINT(...) do { \
    fprintf(stderr,  ": %s: ", __func__); \
    fprintf(stderr, ## __VA_ARGS__); \
    } while (0)
#else
    #define DB_PRINT(...)
#endif
```

or this:

```
#ifndef XLNX_ZYNQMP_GPIO_ERR_DEBUG
#define XLNX_ZYNQMP_GPIO_ERR_DEBUG 1
#endif
```

In `cadence_uart.c`, we're looking at the top code block.
Add debug information by adding a definition for `CADENCE_UART_ERR_DEBUG`.
This will print any time a register in `cadence_uart.c` is accessed (if the module supports it), and any time the module code has a `DB_PRINT` statement.

```
#define CADENCE_UART_ERR_DEBUG // <--

#ifdef CADENCE_UART_ERR_DEBUG
#define DB_PRINT(...) do { \
    fprintf(stderr,  ": %s: ", __func__); \
    fprintf(stderr, ## __VA_ARGS__); \
    } while (0)
#else
    #define DB_PRINT(...)
#endif
```

> ⚠ Some peripherals, such as UART and GPIO, are sometimes used by the guest image for debugging purposes, such as outputting a heartbeat or stdio and stderr output.

This means that enabling debug printing can potentially cause a lot of messages to be printed. This can be disabled when building your image.

After recompiling QEMU, when we re-run `myapp.elf` we will see the UART reads and writes.

```
Gen: 74 0a 00 e7 74 7b 73 2a 6a 52 da 69 a0 b5 ba a2
TX->74
: uart_write:  offset:c0 data:00000074
: uart_read:  offset:b0 data:00000000
TX->0a
: uart_write:  offset:c0 data:0000000a
: uart_read:  offset:b0 data:00000000
TX->00
: uart_write:  offset:c0 data:00000000
: uart_read:  offset:b0 data:00000000
TX->e7
: uart_write:  offset:c0 data:000000e7
: uart_read:  offset:b0 data:00000000
TX->74
: uart_write:  offset:c0 data:00000074
: uart_read:  offset:b0 data:00000000
TX->4b
: uart_write:  offset:c0 data:0000004b
: uart_read:  offset:b0 data:00000000
TX->53
: uart_write:  offset:c0 data:00000043
: uart_read:  offset:b0 data:00000000
TX->2a
: uart_write:  offset:c0 data:0000002a
: uart_read:  offset:b0 data:00000000
TX->6a
: uart_write:  offset:c0 data:0000006a
: uart_read:  offset:b0 data:00000000
TX->52
: uart_write:  offset:c0 data:00000052
: uart_read:  offset:b0 data:00000000
TX->da
: uart_write:  offset:c0 data:000000da
: uart_read:  offset:b0 data:00000000
TX->69
: uart_write:  offset:c0 data:00000069
: uart_read:  offset:b0 data:00000000
TX->a0
: uart_write:  offset:c0 data:000000a0
: uart_read:  offset:b0 data:00000000
TX->b5
: uart_write:  offset:c0 data:000000b5
: uart_read:  offset:b0 data:00000000
TX->ba
: uart_write:  offset:c0 data:000000ba
: uart_read:  offset:b0 data:00000000
TX->a2
: uart_write:  offset:c0 data:000000a2
: uart_read:  offset:14 data:00000000
```

# 12 Advanced QEMU Options

This section contains advanced and less frequently used QEMU options.
Before using these options, make sure you are familiar with the QEMU Options and Commands section.



## 12.1 Display Options

QEMU can provide a virtual monitor for display applications.

This virtual monitor can display using curses, GTK (GNOME ToolKit), or SDL (Simple DirectMedia Layer), libraries. More information on these libraries can be found using the links above.

When using display options to create a display, do not pass the `-nographic` option in the command line.

> Petalinux QEMU does not include SDL support for display emulation. It is recommended that you build QEMU from source with SDL enabled.

| Option | Description | Example |
|---|---|---|
| `-display gtk` | Creates a display output in a GTK window. | `-display gtk` |
| `-display sdl` | Creates a display output via SDL, usually in another window. | `-display sdl` |
| `-display curses` | Creates a display output via curses.<br>**Note**: Nothing is displayed when the graphics device is in graphical mode, or if the graphics device does not support a text mode. | `-display curses` |
| `-vnc <hostname>:<display ID>` | Creates a VNC session on *hostname* through which you can view the display *display ID*.<br>The VNC session will listen on port `5900+X`, where `X` is the display ID. | `-vnc localhost:1`<br>Creates a VNC session on localhost for display 1. The VNC port is 5901. |
| `-vnc <hostname>:<TCP port>,reverse` | Creates a listening VNC session for host *hostname* on port *TCP port*.<br>The port should be `5500+X` to listen for a session with display ID `X`. | `-vnc localhost:5501,reverse`<br>Listens for a VNC session on localhost with display 1. |

### 12.1.1 Connecting to a VNC session

The following section can be done with either TightVNC or RealVNC.

To connect to the monitor on localhost, use the command:

```
vncviewer localhost:<display ID>
```

To connect to the monitor for a different server, enable port forwarding using the command:

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

For example, if the display ID passed into QEMU is 1, the TCP port is 5901. This makes the port forwarding command:

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

Now the previous `vncviewer` command can be used to open the display.

# 13 Using CAN/CAN FD with Xilinx QEMU



## 13.1 Xilinx CAN/CAN FD Introduction

Using Xilinx QEMU, you can stream real or simulated CAN and CAN FD (flexible data-rate) traffic from your host machine to your guest running inside QEMU seamlessly.

Xilinx CAN and CAN FD controllers are developed based on SocketCAN and QEMU CAN bus implementation. Versal ACAP devices support two CAN FDs: CANFD0 and CANFD1. ZynqMP devices support two CANs: CAN0 and CAN1. Bus connection and socketCAN interface for each of the CAN and CAN FD modules can be set through command lines.

⚠ SocketCAN is supported with Linux only. This should already be installed on the host Linux machine if not please install using `sudo apt-get install can-utils`

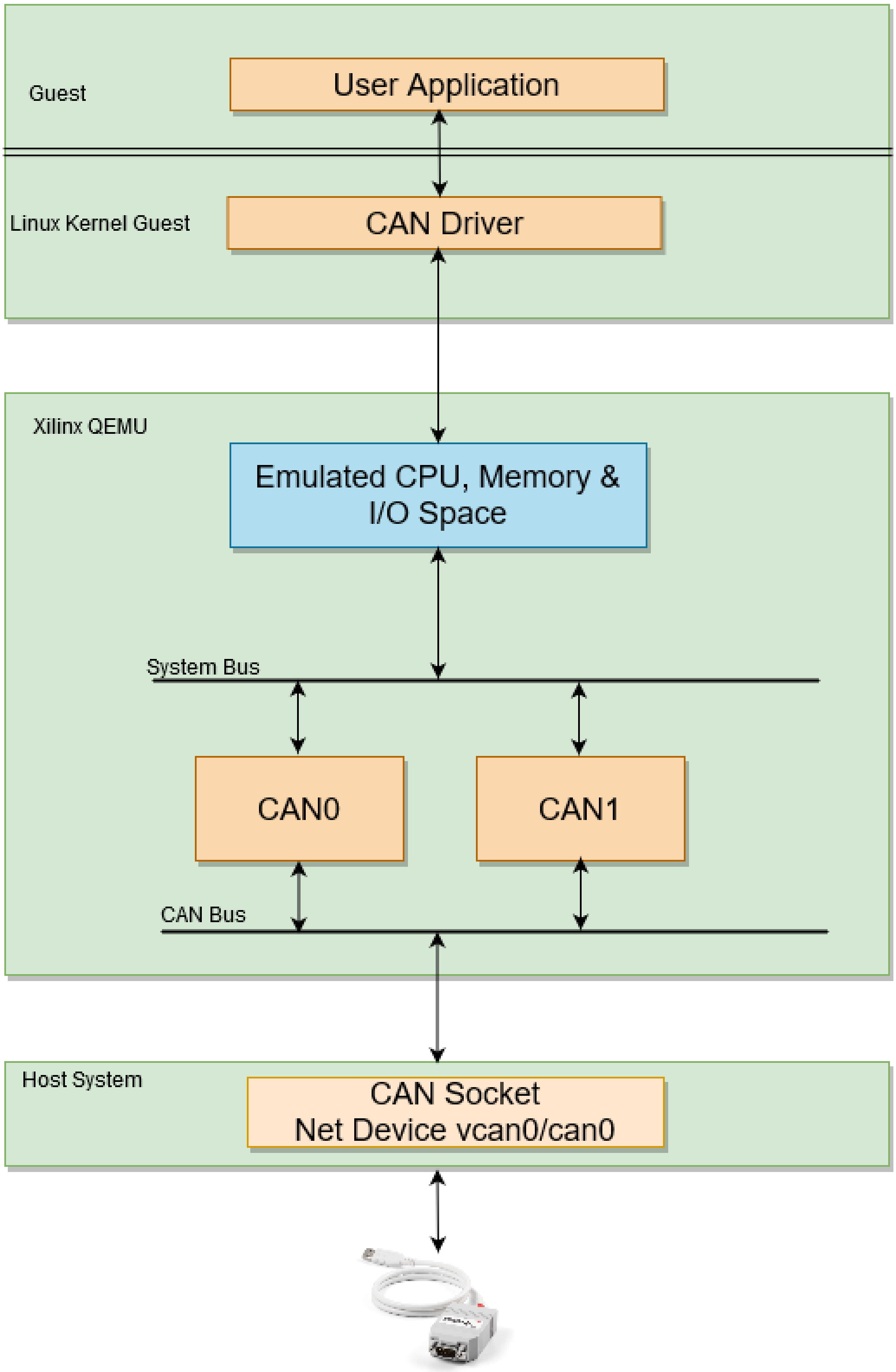
We will be using three commands for initializing a CAN and CAN FD device for Xilinx QEMU. These commands will be appended with ARM instance. Below is an explanation for these commands on supported machines:

| Machine - Device | Command | Description |
|---|---|---|
| ZynqMP - CAN | `-object can-bus,id=canbus0` | This command will create a new `canbus0`. |
| ZynqMP - CAN | `-global driver=xlnx.zynqmp-can,property=canbus0,value=canbus0` | This connects the CAN0 controller with the above-created `canbus0` |
| ZynqMP - CAN | `-object can-host-socketcan,id=socketcan0,if=vcan0,canbus=canbus0` | This connects CAN0 (`canbus0`) to host system CAN bus(which is virtual CAN socket `vcan0` in this example). So, whatever data transferred by CAN0 will be sent to the `vcan0` socket which will go to all devices connected to this `vcan0` interface. |

For CAN FD, we are setting the bus connection in the device tree. We are creating one canfd-bus in DTS and connecting both the CANFD0 and CANFD1 to the common bus. User can create separate buses for both CANFD also. Please check *versal-ps-iou.dtsi* for more on how we are connecting the bus to CANFD controller in DTS.

Before we launch QEMU with CAN or CAN FD devices, we will need to set up CAN interfaces on the host machine. Linux supports virtual and physical interfaces for both CAN and CAN FD. A virtual interface can be created easily. However, the physical CAN interface will need a physical CAN bus/adapter on the host machine to work.

## 13.2 Overview of CAN/CAN FD with QEMU


Guest
User Application
Linux Kernel Guest
CAN Driver
Xilinx QEMU
Emulated CPU, Memory & I/O Space
System Bus
CAN0
CAN1
CAN Bus
Host System
CAN Socket
Net Device vcan0/can0

## 13.3 How to create virtual CAN/CAN FD interface on Linux host machine

Below commands will create a virtual CAN interface:

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

Below commands will create a virtual CAN FD interface:

```
sudo modprobe vcan
sudo ip link add dev vcan0 type vcan
sudo ip link set up vcan0 mtu 72
```

## 13.4 How to create physical CAN/CAN FD interface on Linux host machine

The CAN interface of the host system has to be configured for proper bitrate and set up. The configuration is not propagated from emulated devices through the bus to the physical host device.

Below is an example for a configuration with 1Mbit/s bitrate:

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

## 13.5 Using single CAN with QEMU (for Zynq UltraScale+ MPSoC)

```
# Append the following to ZynqMP ARM instance:
-object can-bus,id=canbus0 \
-global driver=xlnx.zynqmp-can,property=canbus0,value=canbus0 \
-object can-host-socketcan,id=socketcan0,if=vcan0,canbus=canbus0

# MicroBlaze instance is used same way.
```

The above example will create a `canbus0` on the guest, connect CAN0 to `canbus0` and connect CAN0's bus(i.e. `canbus0`) to `vcan0` interface on the host device.

> ⓘ Above QEMU commands can be also used with Petalinux boot. Example: `petalinux-boot --qemu --prebuilt 3 --qemu-args='argument list to append'`

## 13.6 Using both CAN0 and CAN1 devices with QEMU (for Zynq UltraScale+ MPSoC)

```
# Append the following to ZynqMP ARM instance:
-object can-bus,id=canbus0 \
-object can-bus,id=canbus1 \
-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

# MicroBlaze instance is used same way.
```

The above example will create two separate can buses, `canbus0` and `canbus1` on the guest. It will connect CAN0 to canbus0 and CAN1 to `canbus1` and both CAN0 and CAN1 to the `vcan0` interface on the host device.

## 13.7 Using both CANFD0 and CANFD1 devices on separate buses with QEMU (for Versal ACAP)

```
# Add the following canfdbus1 to versal-ps-iou.dtsi after existing canfdbus0 node:

canfdbus1: canfdbus@0 {
        compatible = "can-bus";
    };

# Change canfd1 node to use canfdbus1:
canfd1: can_core@MM_CANFD1 {
        compatible = "xlnx,versal-canfd";
        rx-fifo0 = <0x40>;
        rx-fifo1 = <0x40>;
        enable-rx-fifo1 = <0x1>;
        canfdbus = <&canfdbus1>;
        interrupts = <CAN1_IRQ_0>;
        reg = <0x0 MM_CANFD1 0x0 MM_CANFD1_SIZE 0x0>;
    };

# Compile the DTS as per instruction in chapter 3 and run QEMU versal with new DTB.
```

How to dump random data to CAN through virtual can interface

The below command will pump random data to the `vcan0` interface. Which will be received by CAN0 and CAN1 if they are connected to `vcan0` interface.

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

## 13.8 How to dump random data to CAN FD through virtual CAN FD interface

The below command will pump random data to the `vcan0` interface. Which will be received by CANFD0 and CANFD1 if they are connected to `vcan0` interface.

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

## 13.9 How to analyze data on the host CAN/CAN FD interface

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

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

# 14 Networking in QEMU



## 14.1 Checking the networking interface

QEMU emulates a small sub-network (or LAN if you will) containing a DHCP server, a gateway, and a DNS server; everything you need to access the internet.
There are also some optional components that can be added to this emulated network. The DNS and gateway backs onto your host machine's internet connection.
This means the QEMU VM session has internet access.

Boot QEMU and log in to the system. Use the `ifconfig` utility to check out the networking setup. This will be similar to below:

```
root@xilinx-zcu102-2020_2:~# ifconfig
eth0      Link encap:Ethernet  HWaddr 00:0A:35:00:22:01
          inet addr:10.0.2.15  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::20a:35ff:fe00:2201/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2 errors:0 dropped:0 overruns:0 frame:0
          TX packets:25 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1152 (1.1 KiB)  TX bytes:4732 (4.6 KiB)
          Interrupt:30

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

`eth0`, in this case, is the Cadence GEM. Some network traffic is already accumulated, RX and TX bytes. This is probably for the DHCP acquisition that happened during boot.

## 14.2 Testing the Network

We can use this network connection normally, almost as if it were attached to the host machines network. For example, you can download a file from Xilinx's Github repository. In the booted Linux on QEMU, enter the command:

```
root@xilinx-zcu102-2020_2:~# wget https://github.com/Xilinx/qemu-devicetrees/archive/
master.zip
```

The output should be something like:

```
Connecting to codeload.github.com (192.30.255.120:443)
master.zip           100% |
*********************************************************|  133k  0:00:00 ETA
```

This is a source code tarball for the DTS project retrieved from Xilinx's public Github repository.

`unzip` the downloaded zip file to see if our download worked:

```
root@xilinx-zcu102-2020_2:~# unzip master.zip
```

Output:

```
Archive: master.zip
creating: qemu-devicetrees-master/
inflating: qemu-devicetrees-master/.gitignore
inflating: qemu-devicetrees-master/Makefile
inflating: qemu-devicetrees-master/README
inflating: qemu-devicetrees-master/board-versal-pmc-ddrmc-virt.dts
inflating: qemu-devicetrees-master/board-versal-pmc-vc-p-a2197-00.dts
inflating: qemu-devicetrees-master/board-versal-pmc-vc-pa2197-00.dts
inflating: qemu-devicetrees-master/board-versal-pmc-virt.dts
inflating: qemu-devicetrees-master/board-versal-ps-cosim-vc-p-a2197-00.dts
inflating: qemu-devicetrees-master/board-versal-ps-cosim-virt.dts
inflating: qemu-devicetrees-master/board-versal-ps-vc-p-a2197-00.dts
.....
```

## 14.3 File Transfer with TFTP

QEMU has built-in TFTP capability to allow for easy file transfer to/from the guest machine to the host. Exit QEMU if it is running from before, and on the terminal for the host machine, make a new directory with a file it:

```
mkdir -p /home/${USER}/qemu-training-tftp
echo "hello QEMU world" >> /home/${USER}/qemu-training-tftp/file.txt
```

`file.txt` in this new directory will contain our "hello QEMU world" line of text.

QEMU needs an additional argument to have TFTP access do that directory:

```
-tftp /home/${USER}/qemu-training-tftp
```

If using PetaLinux, restart QEMU with the following modified command:

```
petalinux-boot --qemu --prebuilt 3 --qemu-args "-tftp /home/${USER}/qemu-training-tftp"
```

This will override the default TFTP directory setting to our new directory. Any TFTP request we initiate from the guest will point at this directory we just created.
The built-in TFTP server IP is 10.0.2.2. Log in to the Zynq UltraScale+ MPSoC VM and download the file from the TFTP server:

```
root@xilinx-zcu102-2020_2:~# tftp -g -r file.txt 10.0.2.2
```

cat the file to see if the contents are correct:

```
root@xilinx-zcu102-2020_2:~# cat file.txt
```

## 14.4 File Transfer with SSH

Files can be transferred between the host and guest machines via SSH by using the scp command.

scp syntax is:

scp <source-path> <dest-path>

And the remote path has the syntax of:

user@host:/path/to/file

For example, if copying a file from the host machine to the guest machine, the command might look something like:

```
root@xilinx-zcu102-2020_2:~# scp <your host machine username>@<your host machine IP>:/scratch/doc-example/myapp.elf .

Host '<host IP>' is not in the trusted hosts file.
(ecdsa-sha2-nistp256 fingerprint sha1!! 18:7e:92:d0:33:ed:97:e7:cb:b2:f7:b1:5d:52:5f:a6:34:9a:97:f9)
Do you want to continue connecting? (y/n) y
komlodi@<host IP>'s password:
myapp.elf                                          100%   18KB   17.7KB/s   00:00
root@xilinx-zcu102-2020_2:~#
```

## 14.5 SSH into QEMU

To SSH into QEMU, some additional arguments need to be passed into QEMU.

For example, if using PetaLinux in a Zynq UltraScale+ MPSoC project, add the following arguments with the `petalinux-boot` command:

```
petalinux-boot --qemu --prebuilt 3 --qemu-args "-net nic -net nic -net nic -net
nic,netdev=eth0 -netdev user,id=eth0,hostfwd=tcp::1114-:22"
```

Or if using PetaLinux on a Versal ACAP project:

```
petalinux-boot --qemu --prebuilt 3 --qemu-args "-net nic,netdev=eth0 -netdev
user,id=eth0,hostfwd=tcp::1114-:22 -net nic"
```

Log in to the QEMU machine. From host machine terminal, run the command shown below to access QEMU via SSH:

```
ssh -p 1114 root@localhost
```

If you terminate QEMU and re-run it. Depending on your host machine SSH configuration, you might see the following error when trying to SSH to QEMU:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that a host key has just been changed.
....
....
RSA host key for [localhost]:1115 has changed and you have requested strict checking.
Host key verification failed.
```

This happened due to remote host id changes as shown in the above error message. One way to avoid this is to supply a different port (instead of 1114) each time you boot QEMU.

To completely avoid this, you can skip the host key checking by sending the key to a null known_hosts file. As shown below:

```
ssh -o "UserKnownHostsFile=/dev/null" -o "StrictHostKeyChecking=no" root@localhost -p
1114
```

## 14.6 Connecting to the VM

In the above examples, we did inbound connections to the VM. We will now create another inbound connection, where the VM itself acts as a server. This will be a telnet connection.
Boot QEMU Linux with the following extra port redirection argument:

```
petalinux-boot --qemu --prebuilt 3 --qemu-args "-redir tcp:<port>:10.0.2.15:23"
# Chose your favorite number between 2000 and 10000 for <port>.
# Make sure port number is something unique to avoid port collision with other users
of the same server machine.
```

Log in as normal. There are not many differences to observe in the boot sequence with this change.

The construction of this redirection argument is:

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

The host port is any port of your choosing that QEMU will start listing on. Connections made to this port will be forwarded to the slirp-ip specified and port specified.
The slirp IP is the network address on the small internal network (or LAN) that QEMU creates.

In our case, we use `10.0.2.15` as the IP as this is the IP the DHCP server leases to our VM running Linux. So this will forward connections to port <port> to the VM Linux on port 23 (the commonly used Telnet port).

Open up a new terminal on the same machine you are running QEMU. Telnet into the VM using:

```
telnet localhost <port>
```

This should take you to the login Prompt for VM Linux. Log in with your login credentials. This is a handy way of getting a second (or more) console on a booted Linux system. This is also useful for console access when the UART breaks (but the system still boots).

## 14.7 Setting the TAP network for QEMU

The TAP networking backend makes use of a TAP networking device in the host. It offers very good performance and can be configured to create virtually any type of network topology.
Unfortunately, it requires configuration of that network topology in the host which tends to be different depending on the operating system you are using. Generally speaking, it also requires that you have root privileges.[1]

> ⚠ You might need to install `bridge-utils` and `uml-utilities` on the Linux machine to set up TAP networking. Use the commands shown below to install these tools:

```
sudo apt-get install bridge-utils
sudo apt-get install uml-utilities
```

> ⚠ You might need root privileges. If needed, run the commands shown below with `sudo`

Use the commands shown below to setup TAP network on the host:

```
# Create a bridge named br0
brctl addbr br0
# Add eth0 interface to bridge
brctl addif br0 eth0
# Create tap interface.
tunctl -t tap0 -u `whoami`
# Add tap0 interface to bridge.
brctl addif br0 tap0
# Check/Bring up all interfaces.
ifconfig eth0 up
ifconfig tap0 up
ifconfig br0 up
# Check if bridge is set properly.
brctl show
# Assign IP address to bridge 'br0'.
dhclient -v br0
```

If using Zynq UltraScale+ MPSoC in a PetaLinux project, boot QEMU using the command below:

```
petalinux-boot --qemu --kernel --qemu-args="-net nic -net nic -net nic -net nic -net
tap,ifname=tap0,script=no,downscript=no"
```

Or if using Versal ACAP in a PetaLinux project:

```
petalinux-boot --qemu --kernel --qemu-args="-net nic -net nic -net
tap,ifname=tap0,script=no,downscript=no"
```

## 14.8 NFS mount in QEMU

NFS allows us to share a directory on one device with other devices on the network. NFS only offers close-to-open cache coherence.
This means that the only guarantee provided by the protocol is that if you close a file in a client A and then open the file in another client B, client B will see client A's changes.[862847290[2]]

Prebuilt PetaLinux BSPs have a rootfs and a Linux kernel which are loaded with NFS options. So, no rebuilding/configuring is needed for NFS. Below are simple steps to setup NFS with QEMU on a Linux host:

```
# Check if NFS server is installed on host system or not
dpkg -l | grep nfs-kernel-server
# If its not installed use below command to install:
sudo apt-get install nfs-kernel-server
```

Once the NFS server is installed, add the local directory we want to share. The example below shows how to add the directory `/home/test_nfs` in export settings in the file `/etc/exports`:

```
# For this example, we will add "insecure" option to the nfs entry.
/home/test_nfs *(rw,sync,no_root_squash,insecure)
# Instead of * above we can also assign a IP address for NFS-server host.
```

Boot QEMU using `petalinux-boot` as below:

```
cd <path_to_petalinux_project_created_from_xilinx_bsp>
petalinux-boot --qemu --prebuilt 3
```

Once QEMU boots up, log into the guest. Use below command to mount the host NFS file system to the `/tmp` directory in QEMU.

```
mount -o port=2049,nolock,proto=tcp <host_ip>:/home/test_nfs/ /tmp/
# To find host_ip, use ifconfig or similar ip utility tool.
```

Now, we can see the shared file in `/tmp` directory of the guest.

The above example was a simple configuration for setting up an NFS server. To add more features in NFS server, please go to https://help.ubuntu.com/community/SettingUpNFSHowTo.

## 14.9 References

1. https://wiki.qemu.org/Documentation/Networking#Tap
2. https://wiki.qemu.org/Documentation/Migration_with_shared_storage

# 15 Co-simulation



## 15.1 Prerequisites

SystemC-TLM and RTL experience to compile/develop the examples.

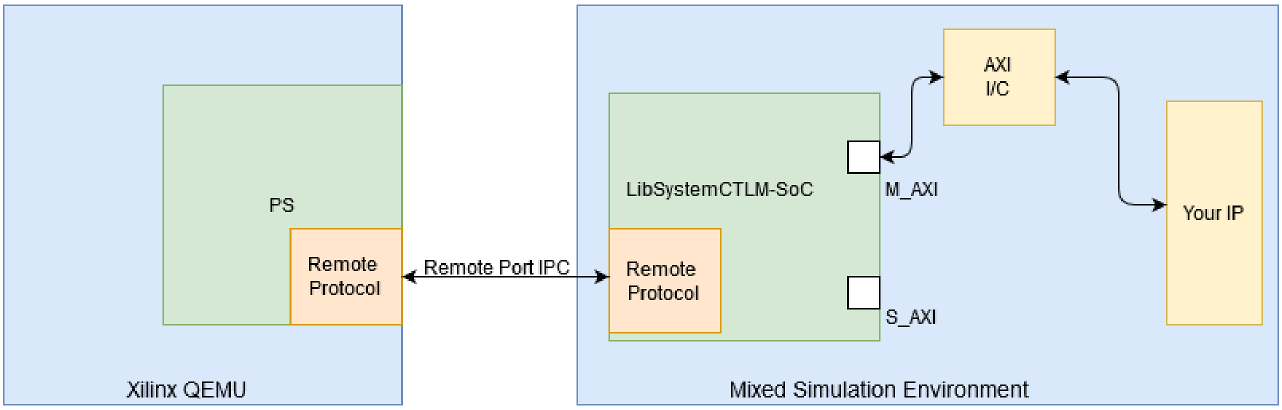
> This feature is predominantly suitable for experienced developers in SystemC/TLM and integration with mixed simulation environments.
> This feature is provided "as-is" and under an open source license model. Feel free to use our libSystemCTLM-SoC to interface your simulation environment to Xilinx's 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).

## 15.2 Overview

You can use Xilinx QEMU to connect and drive mixed simulation environments using the included remote-port framework. This feature enables you to model large and complex systems right from the get-go.

Xilinx exposes a SystemC/TLM interface to connect QEMU, which models the hardened Processing System (PS) of any Zynq-based or Versal ACAP product, to a model of your own IP instantiated in the Programmable Logic (PL). Your IP must be written in either Verilog or SystemC.

The figure below gives a high level overview of the key components.




### 15.2.1 *Figure 1 - Co-Simulation Block Diagram*

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.

## 15.3 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.

## 15.4 libsystemctlm-soc

libSystemCTLM-SoC provides a standard SystemC wrapper around Zynq7000, Zynq Ultrascale+ MPSoC, and Versal ACAP hardened PS (as seen in the right hand side of Figure 1), enabling integrators to connect various IP models just like any other SystemC compatible modeling environment.

These wrappers can be found in libsystemctlm-soc repository. For example see the SystemC/TLM-2.0 co-simulation demonstration.

## 15.5 SystemC/TLM-2.0 Co-Simulation Demo

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.

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.

You need a small configuration file (.config.mk) to be compile the demo source code package. an example of a .config.mk file is shown below for your reference:

```
CXXFLAGS += -std=c++13
HAVE_VERILOG=n
HAVE_VERILOG_VERILATOR=n
HAVE_VERILOG_VCS=n
SYSTEMC = /scratch/tool/systemc-2.3.1/                          //<<== Change
to your local SystemC location.
LD_LIBRARY_PATH=/scratch/tool/systemc-2.3.1/lib-linux64/    //<<== Change
to your local SystemC Library location.
```

## 15.6 Co-Simulating with QEMU

### 15.6.1 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.

> The hardware device tree is specific to QEMU, and should not be confused with the Linux guest device tree.

The device tree repository provides device tree blobs (DTBs) for co-simulation environments.
After building the DTBs, they are available in the `LATEST/SINGLE_ARCH` or `LATEST/MULTI_ARCH` directory and will have `cosim` in their names.

For more information on this process, see the QEMU device tree wiki page and the device tree repository.

### 15.6.2 Extra Command-Line Options

When doing co-simulation, the `-machine-path`, `-sync-quantum`, and `-icount` options are used to allow communication between QEMU and the SystemC/TLM2.0 module.
`icount` is an optional option used to create more deterministic behavior in QEMU, while `machine-path` and `sync-quantum` are required for co-simulation.

More information on what `-machine-path` does can be found here, and the `icount` and `sync-quantum` options are explained below:

| Option | Description | Example |
|---|---|---|
| `-icount <N>[,sleep=on\|off]` | Enables virtual instruction counting with `2^N` virtual nanoseconds per instruction. This enables aligning the host and virtual clocks or disables real-time CPU sleeping.<br><br>When the virtual CPU is sleeping, the virtual time will advance at default speed unless `sleep=off` is specified.<br>With `sleep=off`, the virtual time will jump to the next timer deadline instantly whenever the virtual CPU goes to sleep and will not advance if no timer is enabled. | `-icount 1`<br><br>The virtual CPU will wait 2^1 nanoseconds of virtual time per instruction. |

| Option | Description | Example |
|---|---|---|
| `-sync-quantum <N>` | Specifies the TLM synchronization quantum in nanoseconds.<br>As the `sync-quantum` decreases, the modeling accuracy increases, but its speed decreases.<br>Use the same `sync-quantum` number for any other simulators used in the co-simulation. | `-sync-quantum 100000`<br>The TLM synchronization quantum is set to 100000 nanoseconds |

The following table is a good starting guideline for `icount` and `sync-quantum` values.

| Platform | sync-quantum | icount (optional) |
|---|---|---|
| Zynq UltraScale+ MPSoC | 1000000 | 1 |
| Versal ACAP | 1000000 | 1 |
| Zynq-7000 | 100000 | 7 |

## 15.6.3 Example QEMU Command

### With icount

```
$QEMU_PATH/aarch64-softmmu/qemu-system-aarch64  -M arm-generic-fdt
-nographic \
-dtb $DTS_PATH/zcu102-arm.cosim.dtb \
-device loader,addr=0xfd1a0104,data=0x8000000e,data-len=4 \
-machine-path /tmp/cosim \
-icount 1 \
-sync-quantum 1000000
```

### Without icount

```
$QEMU_PATH/aarch64-softmmu/qemu-system-aarch64  -M arm-generic-fdt
-nographic \
-dtb $DTS_PATH/zcu102-arm.cosim.dtb \
-device loader,addr=0xfd1a0104,data=0x8000000e,data-len=4 \
-machine-path /tmp/cosim \
-sync-quantum 1000000
```

### 15.6.4 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>
```

For example,

```
/scratch/proj/tasks/ug1169_cosim/systemctlm-cosim-demo$ ./
zynqmp_demo  unix:/tmp/cosim/qemu-rport-_amba@0_cosim@0 1000000

        SystemC 2.3.1-Accellera --- Jul 11 2019 10:13:23
        Copyright (c) 1996-2014 by all Contributors,
        ALL RIGHTS RESERVED
connect to /tmp/cosim/qemu-rport-_amba@0_cosim@0
```

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

## 15.7 POSH

As part of the DARPA POSH (**P**osh **O**pen **S**ource **H**ardware) program, Xilinx put in considerable effort to create open source tools to help in the development of complex mixed-simulation environments.

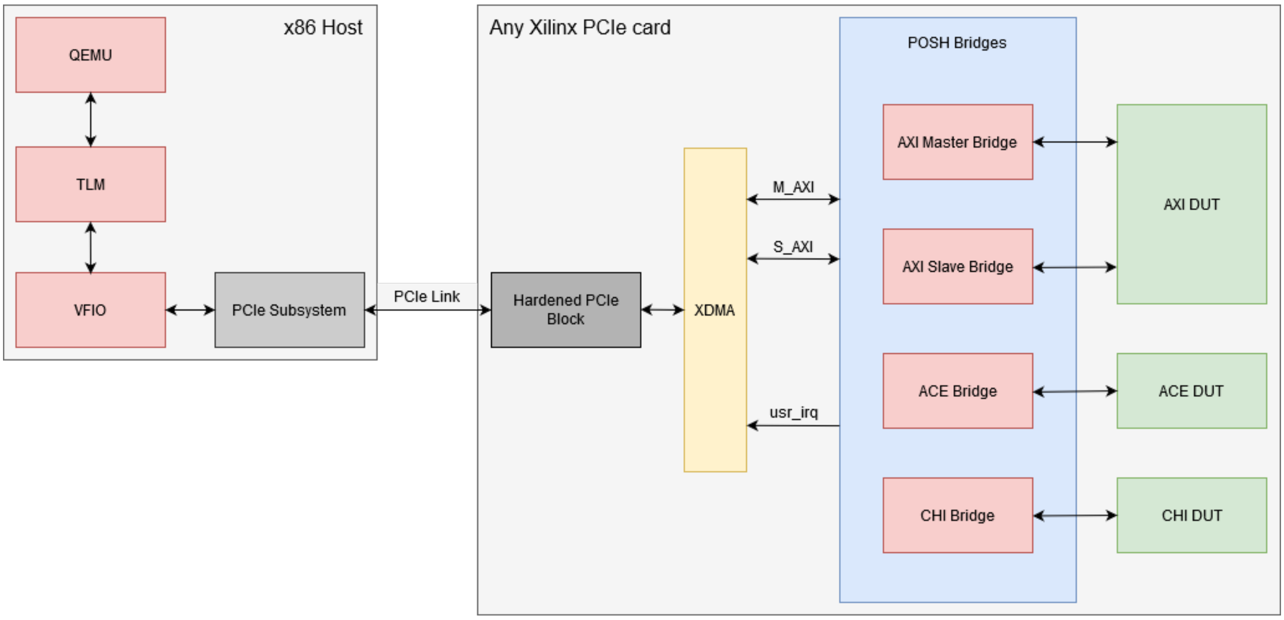
The combinations that can be instantiated include both pure software only simulation environments or software + hardware in the loop (HIL) environments.

An overview of the POSH bridges and utilities can be found libSystemCTLM-SoC and in particular in the documentation section found here:

Some of the tools created are:

- TLM to AXI and AXI to TLM bridges
- TLM to ACE and ACE to TLM bridges
- TLM to CHI and CHI to TLM bridges
- TLM to CXS bridge
- TLM to PCIe bridge
- Protocol checkers
- Traffic generators

A diagram showing how these bridges might be used is shown below:


x86 Host
QEMU
TLM
VFIO
PCIe Subsystem
PCIe Link
Any Xilinx PCIe card
Hardened PCIe Block
XDMA
M_AXI
S_AXI
usr_irq
POSH Bridges
AXI Master Bridge
AXI Slave Bridge
ACE Bridge
CHI Bridge
AXI DUT
ACE DUT
CHI DUT

# 16 Device Trees

This section will detail how to modify a hardware device tree.
It will not tell you how to modify a device tree for your specific hardware and peripherals, but it will provide the tools to do so.



## 16.1 Overview

A device tree is a way to represent hardware. It is comprised of many device tree source (DTS) files and DTS include (DTSI) files.
When the source files are compiled, a flattened device tree (FDT), also known as a device tree blob (DTB), is created. QEMU and Linux use the DTB to understand the structure of the hardware without any hard coding involved.

## 16.2 Acquiring the Tools

Before proceeding, the following programs and files should be installed on your computer

- PetaLinux
- BSPs, on this page we will use a ZCU102 BSP
- Device tree compiler (DTC)

These are available with PetaLinux, Yocto, or Vitis tools.

## 16.3 Modifying a Device Tree

It is strongly recommended to read the device tree specification before modifying a device tree.
The specification will cover the structure, syntax, and good practices of device tree modification.

## 16.4 Device Tree Properties and QEMU

QEMU uses device trees to represent hardware. This device tree is not passed to the guest and is sometimes referred to as the hardware device tree.

Some devices contain device-specific properties that should exist in the hardware device tree.

For example, an SI57X may have a device node that looks like:

```
clock-generator@5e {
    compatible = "silabs,si57x";
    reg = <0x5d>;
    temperature-stability = <0x32>;
};
```

The `temperature-stability` property is one that the device should have present.

If you want to add another device and you're not sure what properties need to be present, you can pass the `-device <device-name>,help` argument along with your other QEMU arguments when starting QEMU.

In this case, the argument will look like:

```
-device si57x,help
```

An alternative is to look at its properties list in the device's `.c` file in the QEMU source.

For example, the SI57X has the properties list:

**qemu/hw/misc/si57x.c**

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

As we can see, it only looks for the `temperature-stability` property.

> If a property is not defined in a device node, QEMU will use a default value. In this case, 50PPM.

## 16.5 Device Tree Properties and the Guest

QEMU guests, such as Linux, can use device trees to understand the hardware it has access to. For instance, a guest device tree may list partitions for a SPI flash.
So for this example, we will modify the device tree to add a partition to SPI flash on a ZCU102 platform using PetaLinux.

Create your ZCU102 project if you haven't already:

```
petalinux-create -t project -s "/path/to/bsp/xilinx-zcu102.bsp" -n "project-name"
```

Boot it once to make sure everything is working properly. Take note of the DTBs used when booting.

```
$petalinux-boot --qemu --prebuilt 3

INFO: sourcing build tools
INFO: No DTB has been specified, use the default one "/scratch/petalinux-images/
xilinx-zcu102-2019.2/pre-built/linux/images/system.dtb".
INFO: Starting microblaze QEMU
INFO: Starting the above QEMU command in the background
INFO:  qemu-system-microblazeel -M microblaze-fdt   -serial mon:stdio -serial /dev/
null -display none -kernel /scratch/petalinux-images/xilinx-zcu102-2019.2/pre-built/
linux/images/pmu_rom_qemu_sha3.elf -device loader,file=/scratch/petalinux-images/
xilinx-zcu102-2019.2/pre-built/linux/images/pmufw.elf      -hw-dtb /scratch/
petalinux-images/xilinx-zcu102-2019.2/pre-built/linux/images/zynqmp-qemu-multiarch-
pmu.dtb -machine-path /tmp/tmp.geJUZ51a7f -device
loader,addr=0xfd1a0074,data=0x1011003,data-len=4 -device
loader,addr=0xfd1a007C,data=0x1010f03,data-len=4
INFO: Set QEMU tftp to /scratch/petalinux-images/xilinx-zcu102-2019.2/images/linux
qemu-system-microblazeel: Failed to connect socket /tmp/tmp.geJUZ51a7f/qemu-rport-
_pmu@0: No such file or directory
qemu-system-microblazeel: info: QEMU waiting for connection on: disconnected:unix:/
tmp/tmp.geJUZ51a7f/qemu-rport-_pmu@0,server
INFO: TCP PORT is free
INFO: Starting aarch64 QEMU
INFO:  qemu-system-aarch64 -M arm-generic-fdt   -serial mon:stdio -serial /dev/null
 -display none -device loader,file=/scratch/petalinux-images/xilinx-zcu102-2019.2/
pre-built/linux/images/bl31.elf,cpu-num=0 -device loader,file=/scratch/petalinux-
images/xilinx-zcu102-2019.2/pre-built/linux/images/Image,addr=0x00080000 -device
loader,file=/scratch/petalinux-images/xilinx-zcu102-2019.2/pre-built/linux/images/
system.dtb,addr=0x15e80000 -device loader,file=/scratch/petalinux-images/xilinx-
zcu102-2019.2/build/misc/linux-boot/linux-boot.elf -gdb tcp::9000 -dtb /scratch/
petalinux-images/xilinx-zcu102-2019.2/pre-built/linux/images/system.dtb  -net nic
-net nic -net nic -net nic,netdev=eth0 -netdev user,id=eth0,tftp=/scratch/petalinux-
images/xilinx-zcu102-2019.2/images/linux   -hw-dtb /scratch/petalinux-images/xilinx-
zcu102-2019.2/pre-built/linux/images/zynqmp-qemu-multiarch-arm.dtb -machine-path /
tmp/tmp.geJUZ51a7f -global xlnx,zynqmp-boot.cpu-num=0 -global xlnx,zynqmp-boot.use-
pmufw=true   -m 4G
QEMU 2.11.1 monitor - type 'help' for more information
#
8<------------------------------------------------------------------------------
-----------
[    7.456218] m25p80 spi0.0: n25q512a (131072 Kbytes)
[    7.457506] 3 fixed-partitions partitions found on MTD device spi0.0
[    7.457958] Creating 3 MTD partitions on "spi0.0":
[    7.458617] 0x000000000000-0x000001e00000 : "boot"
[    7.464772] 0x000001e00000-0x000001e40000 : "bootenv"
[    7.468541] 0x000001e40000-0x000004240000 : "kernel"
# ...
```

For this example, we will be modifying `system.dtb`.

Change directory to where `system.dtb` is and un-flatten `system.dtb` by using DTC.

```
cd pre-built/linux/images
dtc -I dtb -O dts system.dtb -o system.dts
```

Open `system.dts` with your preferred text editor and navigate to the line that says:

```
spi@ff0f0000 {
```

This SPI peripheral has a flash chip as a child. This is the flash chip we will be adding another partition to.

In the flash chip inside the SPI peripheral, there should be three partitions:

```
partition@0x00000000 {
    label = "boot";
    reg = <0x0 0x1e00000>;
};

partition@0x01e00000 {
    label = "bootenv";
    reg = <0x1e00000 0x40000>;
};

partition@0x01e40000 {
    label = "kernel";
    reg = <0x1e40000 0x2400000>;
};
```

Add a fourth partition so it looks like this:

```
partition@0x00000000 {
    label = "boot";
    reg = <0x0 0x1e00000>;
};

partition@0x01e00000 {
    label = "bootenv";
    reg = <0x1e00000 0x40000>;
};

partition@0x01e40000 {
    label = "kernel";
    reg = <0x1e40000 0x2400000>;
};

partition@0x04240000 {
    label = "new-partition";
    reg = <0x4240000 0xc0000>;
};
```

Build the DTB.

```
dtc -I dts -O dtb system.dts -o system.dtb
```

Re-run QEMU.

```
$petalinux-boot --qemu --prebuilt 3

INFO: sourcing build tools
INFO: No DTB has been specified, use the default one "/scratch/petalinux-images/
xilinx-zcu102-2019.2/pre-built/linux/images/system.dtb".
INFO: Starting microblaze QEMU
INFO: Starting the above QEMU command in the background
INFO:  qemu-system-microblazeel -M microblaze-fdt   -serial mon:stdio -serial /dev/
null -display none -kernel /scratch/petalinux-images/xilinx-zcu102-2019.2/pre-built/
linux/images/pmu_rom_qemu_sha3.elf -device loader,file=/scratch/petalinux-images/
xilinx-zcu102-2019.2/pre-built/linux/images/pmufw.elf      -hw-dtb /scratch/
petalinux-images/xilinx-zcu102-2019.2/pre-built/linux/images/zynqmp-qemu-multiarch-
pmu.dtb -machine-path /tmp/tmp.geJUZ51a7f -device
loader,addr=0xfd1a0074,data=0x1011003,data-len=4 -device
loader,addr=0xfd1a007C,data=0x1010f03,data-len=4
INFO: Set QEMU tftp to /scratch/petalinux-images/xilinx-zcu102-2019.2/images/linux
qemu-system-microblazeel: Failed to connect socket /tmp/tmp.geJUZ51a7f/qemu-rport-
_pmu@0: No such file or directory
qemu-system-microblazeel: info: QEMU waiting for connection on: disconnected:unix:/
tmp/tmp.geJUZ51a7f/qemu-rport-_pmu@0,server
INFO: TCP PORT is free
INFO: Starting aarch64 QEMU
INFO:  qemu-system-aarch64 -M arm-generic-fdt   -serial mon:stdio -serial /dev/null
 -display none -device loader,file=/scratch/petalinux-images/xilinx-zcu102-2019.2/
pre-built/linux/images/bl31.elf,cpu-num=0 -device loader,file=/scratch/petalinux-
images/xilinx-zcu102-2019.2/pre-built/linux/images/Image,addr=0x00080000 -device
loader,file=/scratch/petalinux-images/xilinx-zcu102-2019.2/pre-built/linux/images/
system.dtb,addr=0x15e80000 -device loader,file=/scratch/petalinux-images/xilinx-
zcu102-2019.2/build/misc/linux-boot/linux-boot.elf -gdb tcp::9000 -dtb /scratch/
petalinux-images/xilinx-zcu102-2019.2/pre-built/linux/images/system.dtb  -net nic
-net nic -net nic -net nic,netdev=eth0 -netdev user,id=eth0,tftp=/scratch/petalinux-
images/xilinx-zcu102-2019.2/images/linux   -hw-dtb /scratch/petalinux-images/xilinx-
zcu102-2019.2/pre-built/linux/images/zynqmp-qemu-multiarch-arm.dtb -machine-path /
tmp/tmp.geJUZ51a7f -global xlnx,zynqmp-boot.cpu-num=0 -global xlnx,zynqmp-boot.use-
pmufw=true   -m 4G
QEMU 2.11.1 monitor - type 'help' for more information
#
8<-------------------------------------------------------------------------
-----------
[    7.666932] m25p80 spi0.0: n25q512a (131072 Kbytes)
[    7.668616] 4 fixed-partitions partitions found on MTD device spi0.0
[    7.669350] Creating 4 MTD partitions on "spi0.0":
[    7.670832] 0x000000000000-0x000001e00000 : "boot"
[    7.677346] 0x000001e00000-0x000001e40000 : "bootenv"
[    7.681945] 0x000001e40000-0x000004240000 : "kernel"
[    7.685878] 0x000004240000-0x000004300000 : "new-partition"
# ...
```

Verify we can read and write to the new partition

```
root@xilinx-zcu102-2020_2:~# cat /proc/mtd
dev:    size   erasesize  name
mtd0: 01e00000 00002000 "boot"
mtd1: 00040000 00002000 "bootenv"
mtd2: 02400000 00002000 "kernel"
mtd3: 000c0000 00002000 "new-partition"
root@xilinx-zcu102-2020_2:~# dd if=/dev/urandom of=./sample.bin bs=1024 count=64
64+0 records in
64+0 records out
root@xilinx-zcu102-2020_2:~# flashcp -v sample.bin /dev/mtd3
Erasing blocks: 8/8 (100%)
Writing data: 64k/64k (100%)
Verifying data: 64k/64k (100%)
```

## 16.6 Words of Caution

When modifying device trees, your changes must make sense as if you were representing physical hardware. Just because the device tree will build properly doesn't mean it will work properly with hardware, virtualized or otherwise.

For example, if we made our partitions look like:

```
partition@0x00000000 {
    label = "boot";
    reg = <0x0 0x1e00000>;
};

partition@0x01e00000 {
    label = "bootenv";
    reg = <0x1e00000 0x40000>;
};

partition@0x01e40000 {
    label = "kernel";
    reg = <0x1e40000 0x2400000>;
};

partition@0x04240000 {
    label = "new-partition";
    reg = <0x4240000 0x80000000>;
};
```

The flash does not have enough space to have a partition that large.

```
[    7.514727] m25p80 spi0.0: n25q512a (131072 Kbytes)
[    7.516235] 4 fixed-partitions partitions found on MTD device spi0.0
[    7.516899] Creating 4 MTD partitions on "spi0.0":
[    7.518107] 0x000000000000-0x000001e00000 : "boot"
[    7.525290] 0x000001e00000-0x000001e40000 : "bootenv"
[    7.530586] 0x000001e40000-0x000004240000 : "kernel"
[    7.535273] 0x000004240000-0x000084240000 : "new-partition"
[    7.535825] mtd: partition "new-partition" extends beyond the end of device
"spi0.0" -- size truncated to 0x3dc0000
```

In this case, we were warned, but other device tree modifications may have unwanted behavior that won't have warnings.

# 17 Boot Images

This section will cover image generation and boot flows with QEMU.

PetaLinux provides a simpler way to customize boot flow, however this section will cover lower-level tools available for more complex boot flows, should they be needed.

> ⚠ This section does not cover building the files used when creating the boot images. If they are not available, they can be built in a PetaLinux project.



## 17.1 Using SD for Boot

### 17.1.1 Creating the SD Image

We will create the SD image using `dd`, `mkfs`, and `mcopy`.

`mcopy` is a part of `mtools`, and can be downloaded through your package manager (e.g. `apt-get`) or from GNU.org.

```
dd if=/dev/zero of=qemu_sd.img bs=256M 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 ::/
```

### 17.1.2 Booting the Image in QEMU

#### Booting the Image with Zynq UltraScale+ MPSoC

To boot with SD on Zynq UltraScale+ MPSoC, specify:

```
-boot mode=3
-drive index=0
```

or

```
-boot mode=5
-drive index=1
```

For SD0 or SD1 respectively.

**MicroBlaze Machine**

```
qemu-system-aarch64 -M microblaze-fdt \
-nographic \
-hw-dtb ${PROJ_DIR}/images/linux/zynqmp-qemu-multiarch-pmu.dtb \
-kernel ${PROJ_DIR}/images/linux/pmu_rom_qemu_sha3.elf \
-device loader,file=${PROJ_DIR}/images/linux/pmufw.elf \
-device loader,addr=0xfd1a0074,data=0x01011003,data-len=4 \
-device loader,addr=0xfd1a007c,data=0x01010f03,data-len=4 \
-machine-path /tmp/qemu-shm
```

**ARM Machine**

```
qemu-system-aarch64 -M arm-generic-fdt \
-serial mon:stdio \
-m 4G \
-global xlnx,zynqmp-boot.cpu-num=0 \
-global xlnx,zynqmp-boot.use-pmufw=true \
-global xlnx,zynqmp-boot.load-pmufw-cfg=false \
-nographic \
-hw-dtb ${PROJ_DIR}/images/linux/zynqmp-qemu-arm.dtb \
-device loader,file=${PROJ_DIR}/images/linux/zynqmp_fsbl.elf,cpu-num=1 \
-drive file=qemu_sd.img,if=sd,format=raw,index=0 \
-net nic -net nic -net nic -net nic \
-boot mode=5 \
-machine-path /tmp/qemu-shm
```

Even though we put the FSBL in the SD image (as a part of BOOT.BIN), it needs to be passed through the command line as an ELF since QEMU does not have a boot ROM to load the FSBL.

## Booting the Image with Versal ACAP

To boot with SD on Versal ACAP, specify:
`-boot mode=3`
`-drive index=0`

or

`-boot mode=5`
`-drive index=1`

For SD0 or SD1 respectively.

**MicroBlaze Machine**

```
qemu-system-aarch64 -M microblaze-fdt \
-nographic \
-serial mon:stdio \
-hw-dtb ${PROJ_DIR}/images/linux/versal-qemu-multiarch-pmc.dtb \
-device loader,file=${PROJ_DIR}/images/linux/pmc_cdo.bin,addr=0xf2000000 \
-device loader,file=${PROJ_DIR}/images/linux/plm.elf,cpu-num=1 \
-device loader,file=${PROJ_DIR}/images/linux/BOOT_bh.bin,addr=0xf201e000,force-raw \
-device loader,addr=0xf0000000,data=0xba020004,data-len=4 \
-device loader,addr=0xf0000004,data=0xb800fffc,data-len=4 \
-device loader,addr=0xf1110620,data=0x1,data-len=4 \
-device loader,addr=0xf1110624,data=0x0,data-len=4 \
-machine-path /tmp/qemu-shm
```

**ARM Machine**

```
qemu-system-aarch64 -M arm-generic-fdt \
-m 8G \
-nographic \
-serial null \
-serial null \
-serial mon:stdio \
-hw-dtb ${PROJ_DIR}/images/linux/versal-qemu-ps.dtb \
-dtb ${PROJ_DIR}/images/linux/system.dtb \
-drive file=qemu_sd.img,if=sd,format=raw,index=1 \
-net nic -net nic \
-boot mode=5 \
-machine-path /tmp/qemu-shm
```

ⓘ Even though we put the FSBL in the SD image (as a part of BOOT.BIN), it needs to be passed through the command line as an ELF since QEMU needs access to the boot ROM.

## 17.2 Using QSPI for Boot

This section will cover both single flash and dual parallel mode.
Use the one that is more suited to your needs.

⚠ The read and write addresses given in these examples depend on the sizes of your images and how your flash is partitioned.

In this example, Our flash is partitioned in the following way for Zynq UltraScale+ MPSoC:

| Partition Start | Partition End | Partition Name |
|---|---|---|
| 0x0 | 0x1e00000 | Boot |
| 0x1e00000 | 0x1e40000 | DTB |
| 0x1e40000 | 0x4240000 | Kernel |

and has only one partition on Versal ACAP:

| Partition Start | Partition End | Partition Name |
|---|---|---|
| 0x0 | 0x20000000 | Flash |

If you're not sure how your flash is partitioned, you can view the partitions in a few ways:

1. On the Linux guest, run `cat /proc/mtd`

2. In a PetaLinux project, run `petalinux-config` and navigate to `Subsystem AUTO Hardware Settings` → `Flash Settings`, and change your partitions as desired.
   After configuration the partitions, build your project with `petalinux-build`.
3. Unflatten your `system.dtb` file by doing `dtc -I dtb -O dts system.dtb -o system.dts` and find the node for `spi@ff0f0000` on Zynq UltraScale+ MPSoC, or `spi@f1030000` on Versal ACAP. The flash will be a child node of the SPI node.
   The partitions are defined in the `reg` property in the format of `reg = <partition_start partition_size>;`. Modify them as you see fit.
   Once done, reflatten the DTB by doing `dtc -I dts -O dtb system.dts -o system.dtb`
   More information on modifying device trees can be found here.

Note that only methods 2 and 3 allow modification of the partitions.

### 17.2.1 QSPI Boot with Zynq UltraScale+ MPSoC

#### Creating the QSPI boot image

Single Flash Mode

```
dd if=/dev/zero of=qemu_qspi.bin bs=256M count=1
dd if=BOOT.BIN of=qemu_qspi.bin bs=1 seek=0 conv=notrunc
dd if=system.dtb of=qemu_qspi.bin bs=1 seek=31457280 conv=notrunc # Seek offset is
0x1E00000 in hex
dd if=Image of=qemu_qspi.bin bs=1 seek=31719424 conv=notrunc # Seek offset is
0x1E40000 in hex
```

Dual Parallel Mode

If using parallel mode, you must use the `flash_strip_bw` utility.
Information and a download for `flash_strip_bw` can be found here.

Use the same steps as you would for building a QSPI boot image for single flash mode, but after you're done, run:

```
flash_strip_bw qemu_qspi.bin qemu_qspi_low.bin qemu_qspi_high.bin
```

This byte-stripes the boot image so it can be used in dual parallel mode.

> ⚠ This command may take several minutes.

#### Boot the Image in QEMU

Single Flash Mode

To boot with single flash QSPI on Zynq UltraScale+ MPSoC, specify:
`-boot mode=1`
`-drive index=0`

or

```
-boot mode=2
-drive index=1
```

For 24-bit or 32-bit QSPI respectively.

**MicroBlaze Machine**

```
qemu-system-aarch64 -M microblaze-fdt \
-nographic \
-hw-dtb ${PROJ_DIR}/images/linux/zynqmp-qemu-multiarch-pmu.dtb \
-kernel ${PROJ_DIR}/images/linux/pmu_rom_qemu_sha3.elf \
-device loader,file=${PROJ_DIR}/images/linux/pmufw.elf \
-device loader,addr=0xfd1a0074,data=0x01011003,data-len=4 \
-device loader,addr=0xfd1a007c,data=0x01010f03,data-len=4 \
-machine-path /tmp/qemu-shm
```

**ARM Machine**

```
qemu-system-aarch64 -M arm-generic-fdt \
-serial mon:stdio \
-m 4G \
-global xlnx,zynqmp-boot.cpu-num=0 \
-global xlnx,zynqmp-boot.use-pmufw=true \
-global xlnx,zynqmp-boot.load-pmufw-cfg=false \
-nographic \
-hw-dtb ${PROJ_DIR}/images/linux/zynqmp-qemu-arm.dtb \
-device loader,file=${PROJ_DIR}/images/linux/zynqmp_fsbl.elf,cpu-num=0 \
-drive file=qemu_qspi.bin,if=mtd,format=raw,index=0 \
-net nic -net nic -net nic -net nic \
-boot mode=1 \
-machine-path /tmp/qemu-shm
```

Dual Parallel Mode

To boot with dual parallel flash QSPI on Zynq UltraScale+ MPSoC, specify:

```
-boot mode=1
-drive qspi_low,index=0
-drive qspi_high,index=1
```

or

```
-boot mode=2
-drive qspi_low,index=0
-drive qspi_high,index=1
```

For 24-bit or 32-bit QSPI respectively.

**MicroBlaze Machine**

```
qemu-system-aarch64 -M microblaze-fdt \
-nographic \
-hw-dtb ${PROJ_DIR}/images/linux/zynqmp-qemu-multiarch-pmu.dtb \
-kernel ${PROJ_DIR}/images/linux/pmu_rom_qemu_sha3.elf \
-device loader,file=${PROJ_DIR}/images/linux/pmufw.elf \
-device loader,addr=0xfd1a0074,data=0x01011003,data-len=4 \
-device loader,addr=0xfd1a007c,data=0x01010f03,data-len=4 \
-machine-path /tmp/qemu-shm
```

**ARM Machine**

```
qemu-system-aarch64 -M arm-generic-fdt \
-serial mon:stdio \
-m 4G \
-global xlnx,zynqmp-boot.cpu-num=0 \
-global xlnx,zynqmp-boot.use-pmufw=true \
-global xlnx,zynqmp-boot.load-pmufw-cfg=false \
-nographic \
-hw-dtb ${PROJ_DIR}/images/linux/zynqmp-qemu-arm.dtb \
-device loader,file=${PROJ_DIR}/images/linux/zynqmp_fsbl.elf,cpu-num=0 \
-drive file=qemu_qspi_low.bin,if=mtd,format=raw,index=0 \
-drive file=qemu_qspi_high.bin,if=mtd,format=raw,index=1 \
-net nic -net nic -net nic -net nic \
-boot mode=1 \
-machine-path /tmp/qemu-shm
```

U-Boot Commands

In this section we'll use the following U-Boot commands to copy our data from flash into RAM and boot from it.

| Command | Description | Example |
|---|---|---|
| `sf probe [bus[:cs]] [hz] [mode]` | Initializes a flash device on a given bus and chip select. | `sf probe 0 0 0`<br>Probes for a flash device on bus 0 |
| `sf read <memory addr> <flash addr> <len>` | Reads `len` bytes from flash at `flash addr` and stores it in memory at `memory addr`. | `sf read 0x1e00000 0x1e00000 0x20000`<br>Reads 0x20000 bytes from 0x1e00000 on the flash and stores it in memory at 0x1e00000 |

| Command | Description | Example |
|---|---|---|
| `booti [addr [initrd[:size]] [fdt]]` | Boot ARM64 Linux image stored at addr.<br>initrd specifies the address of an initrd in memory. size allows you to specify the size of a raw initrd.<br>If booting a Linux image with a DTB but without an initrd, '-' must be used in place of the initrd argument. | `booti 0x1e40000 - 0x1e00000`<br>Boots using a Linux image stored at 0x1e40000 and a DTB stored at 0x1e00000, with no initrd. |

Use the commands below to copy the data from flash to RAM:

```
sf probe 0 0 0 # Probe the flash so we can access it
sf read 0x1e00000 0x1e00000 0x00020000 # Read system.dtb
sf read 0x01e40000 0x01e40000 0x2100000 # Read Image
booti 0x1e40000 - 0x1e00000 # Boot
```

## 17.2.2 QSPI Boot with Versal ACAP

### Creating the QSPI Boot Image

Single Flash Mode

```
dd if=/dev/zero of=qemu_qspi.bin bs=512M count=1
dd if=BOOT.BIN of=qemu_qspi.bin bs=1 seek=0 conv=notrunc
dd if=system.dtb of=qemu_qspi.bin bs=1 seek=31457280 conv=notrunc # Seek offset is
0x1E00000 in hex
dd if=Image of=qemu_qspi.bin bs=1 seek=31719424 conv=notrunc # Seek offset is
0x1E40000 in hex
dd if=rootfs.cpio.gz.u-boot of=qemu_qspi.bin bs=1 seek=50331648 conv=notrunc # Seek
offset is 0x3000000 in hex
```

Dual Parallel Mode

If using parallel mode, you must use the `flash_strip_bw` utility.
Information and a download for `flash_strip_bw` can be found here.

Use the same steps as you would for building a QSPI boot image for single flash mode, but after you're done, run:

```
flash_strip_bw qemu_qspi.bin qemu_qspi_low.bin qemu_qspi_high.bin
```

This byte-stripes the boot image so it can be used in dual parallel mode.

⚠ This command may take around 10 minutes to finish.

## Boot the Image in QEMU

Single Flash Mode

To boot with single flash QSPI on Versal ACAP, specify:
`-boot mode=1`
`-drive index=0`

or

`-boot mode=2`
`-drive index=1`

For 24-bit or 32-bit QSPI respectively.

**MicroBlaze Machine**

```
qemu-system-aarch64 -M microblaze-fdt \
-nographic \
-serial mon:stdio \
-hw-dtb ${PROJ_DIR}/images/linux/versal-qemu-multiarch-pmc.dtb \
-device loader,file=${PROJ_DIR}/images/linux/pmc_cdo.bin,addr=0xf2000000 \
-device loader,file=${PROJ_DIR}/images/linux/plm.elf,cpu-num=1 \
-device loader,file=${PROJ_DIR}/images/linux/BOOT_bh.bin,addr=0xf201e000,force-raw \
-device loader,addr=0xf0000000,data=0xba020004,data-len=4 \
-device loader,addr=0xf0000004,data=0xb800fffc,data-len=4 \
-device loader,addr=0xf1110620,data=0x1,data-len=4 \
-device loader,addr=0xf1110624,data=0x0,data-len=4 \
-machine-path /tmp/qemu-shm
```

**ARM Machine**

```
qemu-system-aarch64 -M arm-generic-fdt \
-m 8G \
-nographic \
-serial null \
-serial null \
-serial mon:stdio \
-hw-dtb ${PROJ_DIR}/images/linux/versal-qemu-ps.dtb \
-dtb ${PROJ_DIR}/images/linux/system.dtb \
-drive file=qemu_qspi.bin,if=mtd,format=raw,index=0 \
-net nic -net nic \
-boot mode=1 \
-machine-path /tmp/qemu-shm
```

Dual Parallel Mode

To boot with dual parallel flash QSPI on Versal ACAP, specify:
```
-boot mode=1
-drive qspi_low,index=0
-drive qspi_high,index=3
```

or

```
-boot mode=2
-drive qspi_low,index=0
-drive qspi_high,index=3
```

For 24-bit or 32-bit QSPI respectively.

**MicroBlaze Machine**

```
qemu-system-aarch64 -M microblaze-fdt \
-nographic \
-serial mon:stdio \
-hw-dtb ${PROJ_DIR}/images/linux/versal-qemu-multiarch-pmc.dtb \
-device loader,file=${PROJ_DIR}/images/linux/pmc_cdo.bin,addr=0xf2000000 \
-device loader,file=${PROJ_DIR}/images/linux/plm.elf,cpu-num=1 \
-device loader,file=${PROJ_DIR}/images/linux/BOOT_bh.bin,addr=0xf201e000,force-raw \
-device loader,addr=0xf0000000,data=0xba020004,data-len=4 \
-device loader,addr=0xf0000004,data=0xb800fffc,data-len=4 \
-device loader,addr=0xf1110620,data=0x1,data-len=4 \
-device loader,addr=0xf1110624,data=0x0,data-len=4 \
-machine-path /tmp/qemu-shm
```

**ARM Machine**

```
qemu-system-aarch64 -M arm-generic-fdt \
-m 8G \
-nographic \
-serial null \
-serial null \
-serial mon:stdio \
-hw-dtb ${PROJ_DIR}/images/linux/versal-qemu-ps.dtb \
-dtb ${PROJ_DIR}/images/linux/system.dtb \
-drive file=qemu_qspi_high.bin,if=mtd,format=raw,index=0 \
-drive file=qemu_qspi_low.bin,if=mtd,format=raw,index=3 \
-net nic -net nic \
-boot mode=1 \
-machine-path /tmp/qemu-shm
```

U-Boot Commands

In this section we'll use the following U-Boot commands to copy our data from flash into RAM and boot from it:

| Command | Description | Example |
|---|---|---|
| `sf probe [bus[:cs]] [hz] [mode]` | Initializes a flash device on a given bus and chip select. | `sf probe 0 0 0`<br>Probes for a flash device on bus 0 |
| `sf read <memory addr> <flash addr> <len>` | Reads `len` bytes from flash at `flash addr` and stores it in memory at `memory addr`. | `sf read 0x1e00000 0x1e00000 0x20000`<br>Reads 0x20000 bytes from 0x1e00000 on the flash and stores it in memory at 0x1e00000 |
| `booti [addr [initrd[:size]] [fdt]]` | Boot ARM64 Linux image stored at addr.<br>initrd specifies the address of an initrd in memory. size allows you to specify the size of a raw initrd.<br>If booting a Linux image with a DTB but without an initrd, '-' must be used in place of the initrd argument. | `booti 0x1e40000 - 0x1e00000`<br>Boots using a Linux image stored at 0x1e40000 and a DTB stored at 0x1e00000, with no initrd. |

Use the commands below to copy the data from flash to RAM:

```
sf probe 0 0 0 # Probe the flash so we can access it
sf read 0x1e00000 0x1e00000 0x00020000 # Read system.dtb
sf read 0x1e40000 0x1e40000 0x1000000 # Read Image
sf read 0x3000000 0x3000000 0x2000000 # Read initrd
booti 0x1e40000 0x3000000 0x1e00000 # Boot
```

## 17.3 Using TFTP to Boot

### 17.3.1 TFTP Boot with Zynq UltraScale+ MPSoC

Run QEMU using the commands shown below:

**MicroBlaze Machine**

```
qemu-system-aarch64 -M microblaze-fdt \
-nographic \
-hw-dtb ${PROJ_DIR}/images/linux/zynqmp-qemu-multiarch-pmu.dtb \
-kernel ${PROJ_DIR}/images/linux/pmu_rom_qemu_sha3.elf \
-device loader,file=${PROJ_DIR}/images/linux/pmufw.elf \
-device loader,addr=0xfd1a0074,data=0x01011003,data-len=4 \
-device loader,addr=0xfd1a007c,data=0x01010f03,data-len=4 \
-machine-path /tmp/qemu-shm
```

**ARM Machine**

```
qemu-system-aarch64 -M arm-generic-fdt \
-serial mon:stdio \
-global xlnx,zynqmp-boot.cpu-num=0 \
-global xlnx,zynqmp-boot.use-pmufw=true \
-global xlnx,zynqmp-boot.load-pmufw-cfg=false \
-nographic \
-hw-dtb ${PROJ_DIR}/images/linux/zynqmp-qemu-arm.dtb \
-device loader,file=${PROJ_DIR}/images/linux/u-boot.elf \
-net nic -net nic -net nic -net nic \
-net user,id=eth0,tftp=${PROJ_DIR}/images/linux \
-m 4G \
-machine-path /tmp/qemu-shm
```

⚠ The TFTP folder should contain the boot image and system DTB.

In the U-Boot prompt, run:

```
tftpb 0x4000000 system.dtb
tftpb 0x80000 Image
booti 0x80000 - 0x4000000
```

## 17.3.2 TFTP Boot with Versal ACAP

Run QEMU using the commands shown below:

**MicroBlaze Machine**

```
qemu-system-aarch64 -M microblaze-fdt \
-nographic \
-serial mon:stdio \
-hw-dtb ${PROJ_DIR}/images/linux/versal-qemu-multiarch-pmc.dtb \
-device loader,file=${PROJ_DIR}/images/linux/pmc_cdo.bin,addr=0xf2000000 \
-device loader,file=${PROJ_DIR}/images/linux/plm.elf,cpu-num=1 \
-device loader,file=${PROJ_DIR}/images/linux/BOOT_bh.bin,addr=0xf201e000,force-raw \
-device loader,addr=0xf0000000,data=0xba020004,data-len=4 \
-device loader,addr=0xf0000004,data=0xb800fffc,data-len=4 \
-device loader,addr=0xf1110620,data=0x1,data-len=4 \
-device loader,addr=0xf1110624,data=0x0,data-len=4 \
-machine-path /tmp/qemu-shm
```

**ARM Machine**

```
qemu-system-aarch64 -M arm-generic-fdt \
-m 8G \
-nographic \
-serial null \
-serial null \
-serial mon:stdio \
-kernel ${PROJ_DIR}/images/linux/u-boot.elf \
-hw-dtb ${PROJ_DIR}/images/linux/versal-qemu-ps.dtb \
-net nic -net nic \
-net user,id=eth0,tftp=${PROJ_DIR}/images/linux \
-machine-path /tmp/qemu-shm
```

⚠ The TFTP folder should contain the boot image and system DTB.

In the U-Boot prompt, run:

```
tftpb 0x4000000 system.dtb
tftpb 0x80000 Image
tftpb 0x4002000 rootfs.cpio.gz.u-boot
booti 0x80000 0x4002000 0x4000000
```

## 17.4 SD Partitioning and Loading an Ubuntu-core File System

### 17.4.1 Creating a Dummy Container

To create a dummy container for QEMU, use `qemu-img`. `qemu-img` is built when QEMU is built and can be found in your build directory.

| Command | Description | Example |
|---|---|---|
| `qemu-img create <name> <size>` | Creates an SD card image with name *name* with size *size*. | `qemu-img create sd.img 2G` |

### 17.4.2 Creating the Network Backend

A network backend for QEMU can be created with `qemu-nbd`. `qemu-nbd` is built when QEMU is built and can be found in your build directory.

| Command | Description | Example |
|---|---|---|
| `qemu-nbd -c <nbd> <img>` | Connects image *img* to network block device *nbd*. | `qemu-nbd /dev/nbd0 sd.img` |

⚠ If `/dev/nbd0` is not found on the machine, the `nbd` driver is not running.
Install `nbd-client` and `nbd-server` (if they are not already installed) and do `modprobe nbd` to enable them.

### 17.4.3 Creating and Formatting Partitions

Partitions can be created using a terminal-based tool such as `fdisk` or a GUI-based tool like `gparted`.

| Command | Description | Example |
|---|---|---|
| `fdisk <img>` | Goes through partition creation on image *img*. | `fdisk /dev/nbd0` |
| `gparted <img>` | Goes through partition creation on image *img*. | `gparted /dev/nbd0` |

At least two primary partitions are required:

1. A bootable partition that contains `BOOT.BIN`, `Image`, and `system.dtb`. This should be large enough to fit these 3 items, generally around 300MB for Zynq UltraScale+ MPSoC.

   ⓘ The bootable flag can be toggled using `fdisk`.

2. A partition for `rootfs`.

For Example:

```
$ sudo fdisk /dev/nbd0

Welcome to fdisk (util-linux 2.27.1).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.

Device does not contain a recognized partition table.
Created a new DOS disklabel with disk identifier 0x358a6e79.

Command (m for help): m

Help:

  DOS (MBR)
   a   toggle a bootable flag
   b   edit nested BSD disklabel
   c   toggle the dos compatibility flag

  Generic
   d   delete a partition
   F   list free unpartitioned space
   l   list known partition types
   n   add a new partition
   p   print the partition table
   t   change a partition type
   v   verify the partition table
   i   print information about a partition

  Misc
   m   print this menu
   u   change display/entry units
   x   extra functionality (experts only)

  Script
   I   load disk layout from sfdisk script file
   O   dump disk layout to sfdisk script file

  Save & Exit
   w   write table to disk and exit
   q   quit without saving changes

  Create a new label
   g   create a new empty GPT partition table
   G   create a new empty SGI (IRIX) partition table
   o   create a new empty DOS partition table
   s   create a new empty Sun partition table

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 (2048-6291455, default 2048):
Last sector, +sectors or +size{K,M,G,T,P} (2048-6291455, default 6291455): +300M

Created a new partition 1 of type 'Linux' and of size 300 MiB.

Device      Boot Start    End Sectors  Size Id Type
/dev/nbd0p1       2048 616447  614400  300M 83 Linux

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 (616448-6291455, default 616448):
Last sector, +sectors or +size{K,M,G,T,P} (616448-6291455, default 6291455):

Created a new partition 2 of type 'Linux' and of size 2.7 GiB.

Command (m for help): a
Partition number (1,2, default 2): 1

The bootable flag on partition 1 is enabled now.

Command (m for help): p
Disk /dev/nbd0: 3 GiB, 3221225472 bytes, 6291456 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: dos
Disk identifier: 0x358a6e79

Device      Boot  Start     End Sectors  Size Id Type
/dev/nbd0p1 *      2048  616447  614400  300M 83 Linux
/dev/nbd0p2      616448 6291455 5675008  2.7G 83 Linux

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

Partitions can be created with `mkfs`. Bootable partitions must be formatted as `FAT` and other partitions can be `ext2` or `ext4`.

| Command | Description | Example |
| --- | --- | --- |
| `mkfs.vfat -F <fat-size> <partition>` | Creates a `FAT` filesystem on partition *partition*. | `mkfs.fvat -F 32 /dev/nbd0p1`<br>Creates a FAT32 filesystem on `p1` on `/dev/nbd0` |
| `mkfs.ext4 <partition>` | Creates an `ext4` filesystem on partition *partition*. | `mkfs.ext4 /dev/nbd0p2`<br>Creates an `ext4` filesystem on `p2` on `/dev/nbd0` |

### 17.4.4 Mounting Partitions and Copying Files

If using PetaLinux, the `Image` file must be loaded without `initramfs`.

To do this, do the following steps:

1. In your PetaLinux project, run petalinux-config
2. Go to Image Packaging Configuration
3. Go to Root filesystem type (INITRAMFS)
4. Select EXT
5. Exit petalinux-config

> This step is not required if performing `switch-root`.

`Ubuntu-core` can be downloaded from here. Download and extract the ARM64 image.

Mount the partitions and copy the necessary files.
Remember that the images required for booting, such as `BOOT.BIN`, `Image`, and `system.dtb`, must go into the bootable partition, and `ubuntu-core` must go into the second partition.

For Example:

```
$ sudo mount /dev/nbd0p1 /mnt
$ sudo cp -v Image system.dtb BOOT.BIN /mnt
'Image' -> '/mnt/Image'
'system.dtb' -> '/mnt/system.dtb'
'BOOT.BIN' -> '/mnt/BOOT.BIN'
$ sudo umount /dev/nbd0p1

$ sudo mount /dev/nbd0p2 /mnt
$ unxz ubuntu-core.img.xz
$ sudo cp ubuntu-core.img /mnt
$ sudo umount /dev/nbd0p2
```

Once the files are copied, un-mount the partitions and disconnect the nbd connection.

| Command | Description | Example |
|---|---|---|
| `qemu-nbd -d <nbd>` | Disconnects network block device *nbd*. | `qemu-nbd -d /dev/nbd0` |

### 17.4.5 Bootargs

Ensure that `bootargs` points to the correct filesystem for root. In this case, it is:

`root=/dev/mmcblk0p2 rw rootfstype=ext4`

See the bootparam manual page for more information.

# 18 QEMU Module Debug Printing

Module debug printing allows verbose logging for a QEMU virtual peripheral, such as a SPI controller.
This will allow you to see register reads/writes (if the module supports it), and any time a debug print statement is used in a module.



## 18.1 Module Debug Printing by Using the Command Line

QEMU provides a way to print debugging output by passing in the `-d` command-line parameter.

This output typically informs the user of things such as guest errors, unimplemented features in a module, and what a module is currently doing.
However, you could also use `-d` to see information on lower level things, such as CPU and MMU activity.

### 18.1.1 Passing in the Command-Line Argument

`-d` can take in 1 or more arguments separated by a comma (with no space between arguments), or `help` and `trace:help` to show a full list of what can be printed using `-d`.

See the QEMU Options and Commands page for syntax.

### 18.1.2 Example Output

For this example we'll be passing in

```
-d trace:m25p80_command_decoded
```

to see what commands our SPI flash receives.

The output shown below is ZCU102 Linux boot with the added `-d trace:m25p80_command_decoded` command-line parameter:

```
201160@1598298783.243037:m25p80_command_decoded [0x55555ada87d0] new
 command:0x9f
201160@1598298783.254801:m25p80_command_decoded [0x55555ada87d0] new
 command:0x5a
201160@1598298783.261425:m25p80_command_decoded [0x55555ada87d0] new
 command:0x6
201160@1598298783.262073:m25p80_command_decoded [0x55555ada87d0] new
 command:0x1
201160@1598298783.263738:m25p80_command_decoded [0x55555ada87d0] new
 command:0x5
201160@1598298783.269035:m25p80_command_decoded [0x55555ada87d0] new
 command:0x6
201160@1598298783.269038:m25p80_command_decoded [0x55555ada9d10] new
 command:0x6
201160@1598298783.269714:m25p80_command_decoded [0x55555ada87d0] new
 command:0xb7
201160@1598298783.269717:m25p80_command_decoded [0x55555ada9d10] new
 command:0xb7
201160@1598298783.270148:m25p80_command_decoded [0x55555ada9d10] new
 command:0x4
201160@1598298783.271772:m25p80_command_decoded [0x55555ada87d0] new
 command:0x6
201160@1598298783.271775:m25p80_command_decoded [0x55555ada9d10] new
 command:0x6
201160@1598298783.272155:m25p80_command_decoded [0x55555ada87d0] new
 command:0x1
201160@1598298783.272157:m25p80_command_decoded [0x55555ada9d10] new
 command:0x1
201160@1598298783.272779:m25p80_command_decoded [0x55555ada87d0] new
 command:0x5
201160@1598298783.272781:m25p80_command_decoded [0x55555ada9d10] new
 command:0x5
201160@1598298783.273502:m25p80_command_decoded [0x55555ada87d0] new
 command:0x70
201160@1598298783.273504:m25p80_command_decoded [0x55555ada9d10] new
 command:0x70
201160@1598298783.274585:m25p80_command_decoded [0x55555ada87d0] new
 command:0x6
201160@1598298783.274588:m25p80_command_decoded [0x55555ada9d10] new
 command:0x6
201160@1598298783.275370:m25p80_command_decoded [0x55555ada87d0] new
 command:0xb7
201160@1598298783.275373:m25p80_command_decoded [0x55555ada9d10] new
 command:0xb7
201160@1598298783.275769:m25p80_command_decoded [0x55555ada87d0] new
 command:0x4
201160@1598298783.275771:m25p80_command_decoded [0x55555ada9d10] new
 command:0x4
[   16.903020] m25p80 spi0.0: n25q512a (131072 Kbytes)
[   16.911027] 3 fixed-partitions partitions found on MTD device spi0.0
```

```
[   16.911781] Creating 3 MTD partitions on "spi0.0":
```

Note that in this case some commands are repeated, because the flash is in a dual parallel configuration (as shown by the different addresses QEMU prints in the `m25p80_command_decoded` lines).

## 18.2 Module Debug Printing by Modifying Source Code

Some modules have extra debug printing that can be enabled by modifying the source code of the module.

These print statements usually show register accesses and what the module is currently doing.
Additional print statements could also be added by the user, if desired.

### 18.2.1 Finding the Module

When using a device tree, there are many ways to ways to find the module files that you want to add debug printing to. This section will cover two of them.

#### Using info qtree

In the QEMU monitor, `info qtree` will show you the device tree model that QEMU is using.

A short snippet is shown below.

```
(qemu) info qtree
# ...
  dev: cdns.spi-r1p6, id "ps7-spi@0xFF050000"
    gpio-out "" 4
    gpio-out "sysbus-irq" 1
    num-busses = 1 (0x1)
    num-ss-bits = 4 (0x4)
    num-txrx-bytes = 1 (0x1)
    mmio ffffffffffffffff/0000000000000800
    bus: spi0
      type SSI
      dev: sst25wf080, id "spi1_flash3@0"
        gpio-in "ssi-gpio-cs" 1
        nonvolatile-cfg = 36863 (0x8fff)
        spansion-cr1nv = 0 (0x0)
        spansion-cr2nv = 8 (0x8)
        spansion-cr3nv = 2 (0x2)
        spansion-cr4nv = 16 (0x10)
        len-nv-cfg-large-stage = 0 (0x0)
        drive = ""
      dev: sst25wf080, id "spi1_flash2@0"
        gpio-in "ssi-gpio-cs" 1
        nonvolatile-cfg = 36863 (0x8fff)
        spansion-cr1nv = 0 (0x0)
        spansion-cr2nv = 8 (0x8)
        spansion-cr3nv = 2 (0x2)
        spansion-cr4nv = 16 (0x10)
        len-nv-cfg-large-stage = 0 (0x0)
        drive = ""
      dev: sst25wf080, id "spi1_flash1@0"
        gpio-in "ssi-gpio-cs" 1
        nonvolatile-cfg = 36863 (0x8fff)
        spansion-cr1nv = 0 (0x0)
        spansion-cr2nv = 8 (0x8)
        spansion-cr3nv = 2 (0x2)
        spansion-cr4nv = 16 (0x10)
        len-nv-cfg-large-stage = 0 (0x0)
        drive = ""
      dev: sst25wf080, id "spi1_flash0@0"
        gpio-in "ssi-gpio-cs" 1
        nonvolatile-cfg = 36863 (0x8fff)
        spansion-cr1nv = 0 (0x0)
        spansion-cr2nv = 8 (0x8)
        spansion-cr3nv = 2 (0x2)
        spansion-cr4nv = 16 (0x10)
        len-nv-cfg-large-stage = 0 (0x0)
        drive = ""
# ...
  dev: xlnx.zynqmp-csu-core, id "csu_core"
    gpio-out "sysbus-irq" 1
    version-platform = 3 (0x3)
```

```
    version-ps-version = 3 (0x3)
    idcode = 73400467 (0x4600093)
    mmio ffffffffffffffff/0000000000005038
# ...
```

On more complex systems, such as the Zynq UltraScale+ MPSoC or Versal ACAP, the output of `info qtree` can be very long.

For finding the module, the only part we care about is the line that says `dev:`.
The value to the right of `dev:`, e.g. `xlnx.zynqmp-csu-core`, is the object model name that QEMU uses.
We can use `grep` to find what file the object model name was defined in, which will most likely be the module file.

```
komlodi@machine:/scratch/proj/qemu/build$ grep -rin xlnx.zynqmp-csu-core --exclude-dir=build ..
../hw/misc/csu_core.c:42:#define TYPE_XLNX_CSU_CORE "xlnx.zynqmp-csu-core"
```

In this case, the Zynq UltraScale+ MPSoC CSU core source file is in `hw/misc/csu_core.c`.

We can verify this by looking at the top of the file and seeing a reference to debug printing for the CSU core.

```
#ifndef XLNX_CSU_CORE_ERR_DEBUG
#define XLNX_CSU_CORE_ERR_DEBUG 0
#endif
```

Now let's look for the SPI controller, `cdns.spi-r1p6`.

```
komlodi@machine:/scratch/proj/qemu/build$ grep -rin cdns.spi-r1p6 --exclude-dir=build ..
../hw/core/fdt_generic_devices_cadence.c:50:    {    .name = "cdns.spi-r1p6",            .parent = "xlnx.ps7-spi"        },
```

In `fdt_generic_devices_cadence.c` there is a reference to debug printing, but it isn't for a SPI controller.

```
#ifndef FDT_GENERIC_UTIL_ERR_DEBUG
#define FDT_GENERIC_UTIL_ERR_DEBUG 0
#endif
```

If we look deeper, we see a reference to a parent QEMU object model.

```
static const TypeInfo fdt_qom_aliases[] = {
    {   .name = "xlnx.ps7-ethernet",        .parent = "cadence_gem"    },
    {   .name = "cdns,gem",                 .parent = "cadence_gem"    },
    {   .name = "cdns,zynq-gem",            .parent = "cadence_gem"    },
    {   .name = "cdns,zynqmp-gem",          .parent = "cadence_gem"    },
    {   .name = "xlnx.ps7-ttc",             .parent = "cadence_ttc"    },
    {   .name = "cdns.ttc",                 .parent = "cadence_ttc"    },
    {   .name = "cdns.uart",                .parent = "cadence_uart"   },
    {   .name = "xlnx.ps7-uart",            .parent = "cadence_uart"   },
    {   .name = "cdns.spi-r1p6",            .parent = "xlnx.ps7-spi"         },  // <--
    {   .name = "xlnx.xuartps",             .parent = "cadence_uart"   },
};
```

This module name should be the one that contains the debug printing macro we want.

```
komlodi@machine:/scratch/proj/qemu/build$ grep -rin xlnx.ps7-spi --exclude-dir=build ..
../hw/core/fdt_generic_devices_cadence.c:50:    {   .name = "cdns.spi-r1p6",            .parent = "xlnx.ps7-spi"        },
../hw/arm/xilinx_zynq.c:124:    dev = qdev_create(NULL, is_qspi ? "xlnx.ps7-qspi" : "xlnx.ps7-spi");
../include/hw/ssi/xilinx_spips.h:144:#define TYPE_XILINX_SPIPS "xlnx.ps7-spi"
```

We found `#define TYPE_XILINX_SPIPS "xlnx.ps7-spi"`, but it's in a `.h` file.

We can assume that `include/hw/ssi/xilinx_spips.h` will be used in a `.c` file named `hw/ssi/xilinx_spips.c`, but that can be verified with `grep`.

```
komlodi@machine:/scratch/proj/qemu/build$ grep -rn xilinx_spips.h --exclude-dir=build ..
../hw/ssi/xilinx_spips.c:33:#include "hw/ssi/xilinx_spips.h"
Binary file ../.git/index matches
../MAINTAINERS:795:F: include/hw/ssi/xilinx_spips.h
../include/hw/arm/xlnx-zynqmp.h:27:#include "hw/ssi/xilinx_spips.h"
```

Verify that it has the debug printing macro we're looking for.

```
komlodi@machine:/scratch/proj/qemu/build$ vim ../hw/ssi/xilinx_spips.c
// ...
#ifndef XILINX_SPIPS_ERR_DEBUG
#define XILINX_SPIPS_ERR_DEBUG 0
#endif

#define DB_PRINT_L(level, ...) do { \
    if (XILINX_SPIPS_ERR_DEBUG > (level)) { \
        fprintf(stderr,  ": %s: ", __func__); \
        fprintf(stderr, ## __VA_ARGS__); \
    } \
} while (0)
```

### Using the DTS or DTB Files

Using the DTS or DTB files to find the module file is similar to using `info qtree`.

If using a DTB file, you first need to un-flatten it.

```
komlodi@machine:/path/to/xilinx-zcu102-2020.2/pre-built/linux/images$ dtc -I dtb -O dts system.dtb -o system.dts
komlodi@machine:/path/to/xilinx-zcu102-2020.2/pre-built/linux/images$ vim system.dts
```

In the DTS file, find the peripheral that you want to enable debug printing for.
For this example, we'll add debug printing to the ZynpMP's UART.

By looking at the register layout of the Zynq UltraScale+ MPSoC, we know UART0 is at address `0xFF000000`. We can use that to identify the node in the DTS file.

```
serial@ff000000 {
    u-boot,dm-pre-reloc;
    compatible = "cdns,uart-r1p12", "xlnx,xuartps";
    status = "okay";
    interrupt-parent = <0x4>;
    interrupts = <0x0 0x15 0x4>;
    reg = <0x0 0xff000000 0x0 0x1000>;
    clock-names = "uart_clk", "pclk";
    power-domains = <0xc 0x21>;
    clocks = <0x3 0x38 0x3 0x1f>;
    pinctrl-names = "default";
    pinctrl-0 = <0x1d>;
    cts-override;
    device_type = "serial";
    port-number = <0x0>;
};
```

The line we care about is `compatible = "cdns,uart-r1p12", "xlnx,xuartps";`.
QEMU will go left-to-right and look for modules that are compatible with those strings.

Usually you can `grep` the `compatible` string and find the module file that way.

```
komlodi@machine:/scratch/proj/qemu/build$ grep -rin cdns,uart ..
komlodi@machine:/scratch/proj/qemu/build$
```

However, that didn't work, so we need to look for something more generic.

```
komlodi@machine:/scratch/proj/qemu/build$ find .. -name "*uart*"
../hw/riscv/sifive_uart.c
../hw/char/lm32_juart.c
../hw/char/cmsdk-apb-uart.c
../hw/char/omap_uart.c
../hw/char/grlib_apbuart.c
../hw/char/xilinx_iomod_uart.c
../hw/char/exynos4210_uart.c
../hw/char/milkymist-uart.c
../hw/char/nrf51_uart.c
../hw/char/mcf_uart.c
../hw/char/xilinx_uartlite.c
../hw/char/lm32_uart.c
../hw/char/cadence_uart.c
../hw/char/digic-uart.c
```

`hw/char/cadence_uart.c` looks like the most likely file, verify it has a debug macro.

```
komlodi@machine:/scratch/proj/qemu/build$ vim ../hw/char/cadence_uart.c
// ...
#ifdef CADENCE_UART_ERR_DEBUG
#define DB_PRINT(...) do { \
    fprintf(stderr,  ": %s: ", __func__); \
    fprintf(stderr, ## __VA_ARGS__); \
    } while (0)
#else
    #define DB_PRINT(...)
#endif
```

This confirms `hw/char/cadence_uart.c` is the file we're looking for.
If we did not find it, we would search for the next string in the `compatible` string list and repeat the same steps above.

### 18.2.2 Enabling Debug Printing

Modules have different ways to enable debug printing, but the underlying concepts are the same.

#### Changing the Debug Print Level

With modules that have a debug macro similar to

```
komlodi@machine:/scratch/proj/qemu/build$ vim ../hw/ssi/xilinx_spips.c
// ...
#ifndef XILINX_SPIPS_ERR_DEBUG
#define XILINX_SPIPS_ERR_DEBUG 0
#endif

#define DB_PRINT_L(level, ...) do { \
    if (XILINX_SPIPS_ERR_DEBUG > (level)) { \
        fprintf(stderr,  ": %s: ", __func__); \
        fprintf(stderr, ## __VA_ARGS__); \
    } \
} while (0)
```

The `DB_PRINT_L` definition says that it will print if the level is above a threshold.
Therefore, to make it print, make `XILINX_SPIPS_ERR_DEBUG` greater than zero and rebuild QEMU. A higher debug level means more verbose printing.

An example QSPI debug output during ZCU102 Linux boot is shown below:

```
# ...
: xlnx_zynqmp_qspips_flush_fifo_g: Dummy GQSPI Delay Command Entry, Do nothing: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: f404
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 0
: xlnx_zynqmp_qspips_flush_fifo_g: Dummy GQSPI Delay Command Entry, Do nothing: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 1f501
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: fd00
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 1f501
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: fd00
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: c403
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 0
: xlnx_zynqmp_qspips_flush_fifo_g: Dummy GQSPI Delay Command Entry, Do nothing: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: f404
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 0
: xlnx_zynqmp_qspips_flush_fifo_g: Dummy GQSPI Delay Command Entry, Do nothing: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 1f501
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: fd00
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 6f502
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 0
: xlnx_zynqmp_qspips_flush_fifo_g: Dummy GQSPI Delay Command Entry, Do nothing: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: c403
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 0
: xlnx_zynqmp_qspips_flush_fifo_g: Dummy GQSPI Delay Command Entry, Do nothing: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: f404
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 0
: xlnx_zynqmp_qspips_flush_fifo_g: Dummy GQSPI Delay Command Entry, Do nothing: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 1f501
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: fd00
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 6f502
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 0
: xlnx_zynqmp_qspips_flush_fifo_g: Dummy GQSPI Delay Command Entry, Do nothing: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: c403
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 0
: xlnx_zynqmp_qspips_flush_fifo_g: Dummy GQSPI Delay Command Entry, Do nothing: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: f404
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 0
: xlnx_zynqmp_qspips_flush_fifo_g: Dummy GQSPI Delay Command Entry, Do nothing: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 1f501
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: fd00
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: c403
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 0
: xlnx_zynqmp_qspips_flush_fifo_g: Dummy GQSPI Delay Command Entry, Do nothing: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: f404
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 0
: xlnx_zynqmp_qspips_flush_fifo_g: Dummy GQSPI Delay Command Entry, Do nothing: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 1f501
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: fd00
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: c403
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 0
: xlnx_zynqmp_qspips_flush_fifo_g: Dummy GQSPI Delay Command Entry, Do nothing: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: f404
```

```
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 0
: xlnx_zynqmp_qspips_flush_fifo_g: Dummy GQSPI Delay Command Entry, Do
nothing: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 1f501
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: fd00
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: c403
: xlnx_zynqmp_qspips_flush_fifo_g: GQSPI command: 0
[   16.844785] m25p80 spi0.0: n25q512a (131072 Kbytes)
[   16.852754] 3 fixed-partitions partitions found on MTD device spi0.0
[   16.854076] Creating 3 MTD partitions on "spi0.0":
[   16.856450] 0x000000000000-0x000001e00000 : "boot"
[   16.882958] 0x000001e00000-0x000001e40000 : "bootenv"
[   16.896354] 0x000001e40000-0x000004240000 : "kernel"
```

### Adding a Debug Definition

With modules that don't mention print levels, such as

```
#ifdef CADENCE_UART_ERR_DEBUG
#define DB_PRINT(...) do { \
    fprintf(stderr,  ": %s: ", __func__); \
    fprintf(stderr, ## __VA_ARGS__); \
    } while (0)
#else
    #define DB_PRINT(...)
#endif
```

Add a definition for `CADENCE_UART_ERR_DEBUG` and debug printing will be enabled.

## 18.3 Caveats

In some situations, modules are accessed very frequently.
This means that enabling debug printing for that module will cause a lot of text to be printed.

For example, if enabling debug printing for `hw/char/cadence_uart.c` on a ZCU102 default PetaLinux image, we will see all of stdout and stderr redirected to the UART.

```
# ...
p: uart_read:  offset:0 data:00000010
: uart_read:  offset:2c data:0000000a
: uart_write:  offset:30 data:00000072
r: uart_read:  offset:0 data:00000010
: uart_read:  offset:2c data:0000000a
: uart_write:  offset:30 data:0000006f
o: uart_read:  offset:0 data:00000010
: uart_read:  offset:2c data:0000000a
: uart_write:  offset:30 data:00000063
c: uart_read:  offset:0 data:00000010
: uart_read:  offset:2c data:0000000a
: uart_write:  offset:30 data:00000065
e: uart_read:  offset:0 data:00000010
: uart_read:  offset:2c data:0000000a
: uart_write:  offset:30 data:00000073
s: uart_read:  offset:0 data:00000010
: uart_read:  offset:2c data:0000000a
: uart_write:  offset:30 data:00000073
s: uart_read:  offset:0 data:00000010
: uart_read:  offset:2c data:0000000a
: uart_write:  offset:30 data:00000069
i: uart_read:  offset:0 data:00000010
: uart_read:  offset:2c data:0000000a
: uart_write:  offset:30 data:0000006e
n: uart_read:  offset:0 data:00000010
: uart_read:  offset:2c data:0000000a
: uart_write:  offset:30 data:00000067
g: uart_read:  offset:0 data:00000010
: uart_read:  offset:2c data:0000000a
: uart_write:  offset:30 data:00000020
 : uart_read:  offset:0 data:00000010
: uart_read:  offset:2c data:0000000a
: uart_write:  offset:30 data:00000046
F: uart_read:  offset:0 data:00000010
: uart_read:  offset:2c data:0000000a
: uart_write:  offset:30 data:00000044
D: uart_read:  offset:0 data:00000010
: uart_read:  offset:2c data:0000000a
: uart_write:  offset:30 data:00000054
T: uart_read:  offset:0 data:00000010
: uart_read:  offset:2c data:0000000a
: uart_write:  offset:30 data:0000000d
: uart_read:  offset:0 data:00000010
: uart_read:  offset:2c data:0000000a
: uart_write:  offset:30 data:0000000a
# ...
```

# 19 Accessing Storage Media in QEMU

In this chapter, we are going to use mainly SATA disks but the techniques covered here are directly applicable to SD, NAND and QSPI media.



## 19.1 SATA Disks

QEMU can emulate a SATA disk attached to the machine via the AHCI SATA controller. The SATA controller is always present in the model, but without extra command-line arguments, the SATA slots are modeled as empty.

FIrst, create a blank file 2GB in size (all zeros):

```
dd if=/dev/zero of=sata0.img bs=1M count=2048
```

Launch QEMU in PetaLinux with the following extra arguments to attach a SATA disk:

```
petalinux-boot --qemu --prebuilt 3 --qemu-args "-drive
file=sata0.img,format=raw,id=sata-drive -device ide-drive,drive=sata-
drive,bus=ahci@0xFD0C0000.0"
```

Breaking it down, the `-drive` argument is creating a QEMU storage drive for use by a device.
The `file=sata0.bin` argument means the drive will use our new file as data storage.
The `format=raw` argument tells QEMU that the file we pass is a raw file and the file contents are to be the disk contents "as-is".
The `-device` argument creates the actual SATA disk. Note the `bus=ahci@0xFD0C0000.0` argument, which attaches the disk to the Zynq UltraScale+ MPSoC SATA controller (named by QEMU as `ahci@0xFD0C0000.0`).
Both arguments specify the same ID, which causes the SATA disk to use the drive specified in the first argument (in turn backing onto `sata0.bin`) for disk data.

The Kernel boot log should contain something like this:

```
[    6.815251] ata2: SATA link down (SStatus 0 SControl 300)
[    6.818587] ata1: SATA link up 1.5 Gbps (SStatus 113 SControl 300)
[    6.823298] ata1.00: ATA-7: QEMU HARDDISK, 2.5+, max UDMA/100
[    6.823960] ata1.00: 4194304 sectors, multi 16: LBA48 NCQ (depth 32)
[    6.825414] ata1.00: applying bridge limits
[    6.827816] ata1.00: configured for UDMA/100
[    6.842715] scsi 0:0:0:0: Direct-Access     ATA      QEMU HARDDISK    2.5+ PQ: 0
ANSI: 5
[    6.855480] sd 0:0:0:0: [sda] 4194304 512-byte logical blocks: (2.15 GB/2.00 GiB)
[    6.858989] sd 0:0:0:0: [sda] Write Protect is off
[    6.870276] sd 0:0:0:0: [sda] Write cache: enabled, read cache: enabled, doesn't
support DPO or FUA
```

Note the third last line, which indicates we have found a 2GB SATA disk and enumerated as `sda`. This will be available for use in the logged in system as `/dev/sda`.
For those with experience using `dev` nodes, be aware that `/dev/sda` is not a partition, it is the raw disk image as full contents.
Later, when we come to work with partitions, we will have extra `/dev` notes for the partitions. We have no partitions or file-systems yet (as we simply have no meaningful data at all!).

The size of the `sata0.bin` file has been used to size the SATA disk itself. If we created our blank file as a different size, this size would change accordingly.
QEMU does not model any particular manufacturer of the disk, rather it models a generic SATA drive and hence it self-identifies to the system as "`QEMU HARDDISK`" (4th last line).

Log in to the system with *root* as username and password.

## 19.2 Low-Level Data Read and Write

In the QEMU system, inspect the first 4kB of the SATA disk contents using the `hexdump` utility. We could inspect more data by increasing the `-n` argument, but we will only operate of the first 4k for brevity in this exercise.

```
root@xilinx-zcu102-2020_2:~# hexdump -C /dev/sda -n 4096 | more
```

It should be blank, this is the expected output:

```
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001000
```

We will use the `dd` utility to write some randomized data to the disk. Randomized `dd`'ing is a good low-level test of whether storage media works (without the complication of files and filesystems).
Write random data to the first 3kB of `/dev/sda` using this command.

```
root@xilinx-zcu102-2020_2:~# dd if=/dev/urandom of=/dev/sda bs=1k count=3
```

The `dd` command is a low-level data copying utility that is usable for copying data from one file to another. It allows the copying of a subsection of a source file to a subsection of the destination file.
The `dev` node `/dev/sda` is not a regular file, but behaves like a file and backs onto the SATA disk (via the kernel

drivers and SATA hardware).
So dd'ing to `/dev/sda` will copy data to the SATA disk.

> Don't try this at home! - if you do this on your Linux host you will destroy your OS installation (although you will usually have to use `sudo` to execute the command).

The arguments are as follows (check the man page by doing `man dd` on the host machine, outside of QEMU, for more explanation of each):

| Argument | Meaning |
|---|---|
| `if=/dev/urandom` | Use the special file `urandom` for source data. This file when read just generates random data from the kernel's built-in random number generator |
| `of=dev/sda` | Use the SATA disk as the destination |
| `bs=1k` | Copy data in 1kB chunks |
| `count=3` | Copy 3 chunks of data total |

The expected output should be something like:

```
3+0 records in
3+0 records out
```

`hexdump` the data again, to see our random data:

```
root@xilinx-zcu102-2020_2:~# hexdump -C /dev/sda -n 4096 | more
```

The output should be something like this. Note that your actual data will be different from this one, as it is randomized:

```
00000000  5d a3 a1 75 74 08 b8 01  c3 70 0d 4c 0f d7 5c a0  |]..ut....p.L..\.|
...
...
00000be0  88 e9 04 34 80 6d 81 7e  c4 ba 5e 68 03 9b a5 60  |...4.m.~..^h...`|
00000bf0  36 ec 9d 95 f3 b9 7b 49  5f ea da 76 7d bf db c3  |6.....{I_..v}...|
00000c00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001000
```

Copy-paste the data (or just the start of it) to notepad or a similar utility for future reference. Exit QEMU.

Back on the host machine (not in QEMU) `hexdump` the `sata0.bin` file:

```
hexdump -C sata0.bin -n 4096 | more
```

You should observe identical contents, as the SATA disk file is just a raw file containing the disk data:

```
00000000  5d a3 a1 75 74 08 b8 01  c3 70 0d 4c 0f d7 5c a0  |]..ut....p.L..\.|
00000010  13 b7 cf 74 6c 7c e0 51  1d 6f c9 a0 01 b1 15 03  |...tl|.Q.o......|
00000020  ed e1 f4 27 b9 fd 1f d8  48 44 0e aa 27 81 5f 07  |...'....HD..'._.|
00000030  5e 5d 08 4e 3c 7a 73 70  39 ed 1c f8 ef e8 79 18  |^].N<zsp9.....y.|
00000040  28 6b e7 a7 c3 f5 27 7b  e9 75 bf 3e 70 ec 16 19  |(k....'{.u.>p...|
00000050  f9 af 1b 72 a6 0e 25 e5  1f 0c f9 c9 b3 14 ae fa  |...r..%.........|
...
00000be0  88 e9 04 34 80 6d 81 7e  c4 ba 5e 68 03 9b a5 60  |...4.m.~..^h...`|
00000bf0  36 ec 9d 95 f3 b9 7b 49  5f ea da 76 7d bf db c3  |6.....{I_..v}...|
00000c00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001000
```

Boot QEMU again and log in, and repeat the same `hexdump` command (first one, dumping `/dev/sda`).
You should observe the identical output. The same file contents have been populated from the first run.

## 19.3 Compressed Disk Images

2GB isn't very big and doesn't really test if our software can handle big disks. Creating large disks with the above method will use a lot of space. To resolve this problem, QEMU provides a compressed disk image format called QCOW2.
To create a QCOW2 image, run:

```
# qemu-img is located at build directory of QEMU.
qemu-img create -f qcow sata0.qcow 8T
```

The qemu-img utility is also included as part of the PetaLinux tool suite. This is going to create an 8TB (don't panic) compressed image using the `qcow2` image format. QCOW is an image format designed for use as virtual machine disk images. `sata0.qcow` is going to be the file holding the data. The expected output should be something like:

```
Formatting 'sata0.qcow', fmt=qcow size=8796093022208
```

Use `ls` to inspect the size of the newly created file, the expected output should look like:

```
ls -l sata0.qcow
-rw-r--r-- 1 fnuv icdes 33554480 Feb 13 13:51 sata0.qcow
```

It only consumes 32MB so far, even though it is an 8TB image. This file size will grow as the disk accumulates meaningful data.

Start QEMU using this new disk image (instead of `sata0.bin` from before):

```
petalinux-boot --qemu --prebuilt 3 --qemu-args "-drive file=sata0.qcow,id=sata-drive
-device ide-drive,drive=sata-drive,bus=ahci@0xFD0C0000.0"
```

⚠ The `format=raw` argument to the drive is dropped now. QEMU will auto-detect that we are using a QCOW image from the file header in `sata0.qcow`.

Check the boot log (same as before about halfway down) and see something like this:

```
[ 6.866559] ata1.00: configured for UDMA/100
[ 6.882732] scsi 0:0:0:0: Direct-Access ATA QEMU HARDDISK 2.5+ PQ: 0 ANSI: 5
[ 6.893114] sd 0:0:0:0: [sda] 17179869184 512-byte logical blocks: (8.80 TB/8.00 TiB)
[ 6.896786] sd 0:0:0:0: [sda] Write Protect is off
[ 6.899581] sd 0:0:0:0: [sda] Write cache: enabled, read cache: enabled, doesn't
support DPO or FUA
```

We can see the Linux kernel has found an 8TB SATA disk (third last line). Log in to the system using usual *root*/*root* credentials.

Like in the previous example, inspect the first 4kB of the SATA disk contents using the `hexdump` utility.

```
root@xilinx-zcu102-2020_2:~# hexdump -C /dev/sda -n 4096 | more
# Again, it should be blank, this is the expected output:
00000000  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00001000
```

`dd` data, as before, but this time `dd` 1MB:

```
root@xilinx-zcu102-2020_2:~# dd if=/dev/urandom of=/dev/sda bs=1k count=1024
```

`hexdump` the first 4k (you could `hexdump` more, but 4k keeps the output brief):

```
root@xilinx-zcu102-2020_2:~# hexdump -C /dev/sda -n 4096 | more
```

The output should be something like this (your random data will differ):

```
00000f80  0d d8 8a 6f eb e7 e7 cf  87 d8 3e c1 6d 43 ff 2a  |...o......>.mC.*|
00000f90  42 7a 77 85 d6 13 a3 10  4f 09 4a 0e e4 31 05 2d  |Bzw.....O.J..1.-|
00000fa0  df 50 9d 98 26 5a 97 22  36 83 9d 20 ed 06 29 df  |.P..&Z."6.. ..).|
00000fb0  1b 42 2c 5c ec 41 1a 2f  54 5c 27 f6 ab 71 1e 6f  |.B,\.A./T\'..q.o|
00000fc0  0e 4b fd fc f9 d2 6d b9  f2 72 8f 56 5b ff ca f9  |.K....m..r.V[...|
00000fd0  a8 32 31 13 05 24 ad 5e  d5 21 60 1a c0 81 e6 34  |.21..$.^.!`....4|
00000fe0  46 b2 b8 3b ac 75 5c 39  d7 50 7f 53 62 e2 fa 97  |F..;.u\9.P.Sb...|
00000ff0  73 de 07 93 08 d3 b1 ac  e7 a1 94 cf ca 99 56 d7  |s.............V.|
00001000
```

Copy the output (or just a little bit of it) to notepad or similar for future reference. Exit QEMU. Inspect the size of the backing file:

```
ls -l sata0.qcow
# The output should be something like:
-rw-r--r-- 1 fnuv icdes 34611200 Feb 13 14:00 sata0.qcow
```

It's grown! - by about 1MB. This makes sense, as our DD command created 1MB of actual data that needs storage.
Restart QEMU and log in.
Repeat the same `hexdump` command and compare it to the copied data from before. It should match. Exit QEMU.

We can use `qemu-img` to covert the compressed file back to a raw file, as 8TB is far too big to create as a raw file.

## 19.4 Multiple Disks

The SATA controller in QEMU has two ports. We can attach a data disk to each.
Let's create one more disk, called `sata1.qcow`. Note that this is just an arbitrary filename that is named to reduce confusion.

```
cp sata0.qcow sata1.qcow
```

Double up the command line arguments, with an extra `-drive` and `-device` for SATA disk 1:

```
petalinux-boot --qemu --prebuilt 3 --qemu-args "\
    -drive file=sata0.bin,format=raw,id=sata-drive0 -device ide-drive,drive=sata-
drive0,bus=ahci@0xFD0C0000.0 \
    -drive file=sata1.qcow,id=sata-drive1 -device ide-drive,drive=sata-
drive1,bus=ahci@0xFD0C0000.1"
```

Note that the `id=` arguments for each pair have been made unique. The second `-device` argument attaches to `bus=ahci@0xFD0C0000.1` instead of `.0`, which will connect this SATA disk to the second port.

```
[    6.921009] scsi 0:0:0:0: Direct-Access     ATA      QEMU HARDDISK    2.5+ PQ: 0
ANSI: 5
[    6.931400] sd 0:0:0:0: [sda] 17179869184 512-byte logical blocks: (8.80 TB/8.00
TiB)
[    6.934314] sd 0:0:0:0: [sda] Write Protect is off
[    6.935904] sd 0:0:0:0: [sda] Write cache: enabled, read cache: enabled, doesn't
support DPO or FUA
[    6.953829] scsi 1:0:0:0: Direct-Access     ATA      QEMU HARDDISK    2.5+ PQ: 0
ANSI: 5
[    6.960608] dwc3 fe200000.dwc3: Failed to get clk 'ref': -2
[    6.962348] sd 1:0:0:0: [sdb] 17179869184 512-byte logical blocks: (8.80 TB/8.00
TiB)
[    6.962873] sd 1:0:0:0: [sdb] Write Protect is off
[    6.962968] dwc3 fe200000.dwc3: Configuration mismatch. dr_mode forced to gadget
[    6.964927] sd 1:0:0:0: [sdb] Write cache: enabled, read cache: enabled, doesn't
support DPO or FUA
```

In the kernel boot log, we should see two SATA disks detected. The 2GB raw image is used for disk 0, enumerated as `/dev/sda`, and the 8TB QCOW image is used for disk 1, enumerated as `/dev/sdb`.
Log in to the system and `hexdump /dev/sdb` to see the data on the second SATA disk.

```
root@xilinx-zcu102-2020_2:~# hexdump -C /dev/sdb -n 4096 | more
```

Data should match the data from the QCOW disk image `dd` test (check your notepad pastes).

## 19.5 File Systems

Reboot the QEMU system and log in. Just boot with disk0 using the `sata0.bin` raw image:

```
petalinux-boot --qemu --prebuilt 3 --qemu-args "-drive
file=sata0.bin,format=raw,id=sata-drive -device ide-drive,drive=sata-
drive,bus=ahci@0xFD0C0000.0"
```

Log in as normal. From within the booted system, we are going to format the SATA disk with a partition table and single FAT partition. We use the `fdisk` utility on the `/dev/sda` node:

```
root@xilinx-zcu102-2020_2:~# fdisk /dev/sda
Device contains neither a valid DOS partition table, nor Sun, SGI, OSF or GPT
disklabel
Building a new DOS disklabel. Changes will remain in memory only,
until you decide to write them. After that the previous content
won't be recoverable.
```

It created a new partition table for us. Print the partition table:

```
Command (m for help): p

Disk /dev/sda: 2147 MB, 2147483648 bytes
255 heads, 63 sectors/track, 261 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks  Id System
```

The partition table contents are empty. Create a new partition, using the 'n' command. Select 'p' for primary, partition number 1, and use the defaults for the offset and sizes:

```
Command (m for help): n
Command action
   e   extended
   p   primary partition (1-4)
p
Partition number (1-4): 1
First cylinder (1-261, default 1): 1
Last cylinder or +size or +sizeM or +sizeK (1-261, default 261): Using default value
261
```

Print the partition table again:

```
Command (m for help): p

Disk /dev/sda: 2147 MB, 2147483648 bytes
255 heads, 63 sectors/track, 261 cylinders
Units = cylinders of 16065 * 512 = 8225280 bytes

   Device Boot      Start         End      Blocks  Id System
/dev/sda1               1         261     2096451  83 Linux
```

It now has a partition. Use the 't' command to set the partition type to FAT32. Use 'L' if you want to look at the options for partition types. Select partition #1 and set the Hex code to 'b'.

```
Command (m for help): t
Selected partition 1
Hex code (type L to list codes): L

 0 Empty                    1b Hidden Win95 FAT32     9f BSD/OS
 1 FAT12                    1c Hidden W95 FAT32 (LBA) a0 Thinkpad hibernation
 4 FAT16 <32M               1e Hidden W95 FAT16 (LBA) a5 FreeBSD
 5 Extended                 3c Part.Magic recovery    a6 OpenBSD
 6 FAT16                    41 PPC PReP Boot          a8 Darwin UFS
 7 HPFS/NTFS                42 SFS                    a9 NetBSD
 a OS/2 Boot Manager        63 GNU HURD or SysV       ab Darwin boot
 b Win95 FAT32              80 Old Minix              b7 BSDI fs
...

Hex code (type L to list codes): b
Changed system type of partition 1 to b (Win95 FAT32)
```

Write the changes to disk using the *w* command:

```
Command (m for help): w
The partition table has been altered.
Calling ioctl() to re-read partition table
[  490.729264]  sda: sda1
```

This will exit the `fdisk` program. We should now have a `dev` node for the partition:

```
root@xilinx-zcu102-2020_2:~# ls /dev/sda1
/dev/sda1
```

Make a FAT filesystem on our new partition.

```
root@xilinx-zcu102-2020_2:~# mkfs.vfat /dev/sda1
```

Mount the partition.

```
root@xilinx-zcu102-2020_2:~# mount /dev/sda1 /mnt
```

We can now copy files to and from `/mnt` directory. This will copy files to the emulated disk. Create a file in `/mnt`.

```
echo "Hello QEMU training world" >> /mnt/file
```

Sync filesystems:

```
root@xilinx-zcu102-2020_2:~# sync
```

Exit QEMU. `hexdump` the `sata0.bin` file with the following query (which will search for "FAT" string and some context around it). We should see the binary data for the partition table and the new FAT partition.

```
$ hexdump -C sata0.bin | grep FAT -A 100 -B 10 | more
```

Check the ASCII in the rightmost columns to see human-readable strings. We can see the "FAT32" magic string used to identify FAT partitions.

```
00000bd0  f4 62 02 18 94 aa 98 81  86 32 e6 84 74 f8 cb 31  |.b.......2..t..1|
00000be0  88 e9 04 34 80 6d 81 7e  c4 ba 5e 68 03 9b a5 60  |...4.m.~..^h...`|
00000bf0  36 ec 9d 95 f3 b9 7b 49  5f ea da 76 7d bf db c3  |6.....{I_..v}...|
00000c00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00007e00  eb 58 90 6d 6b 64 6f 73  66 73 00 00 02 08 06 00  |.X.mkdosfs......|
00007e10  02 00 00 00 00 f8 00 00  3f 00 ff 00 00 00 00 00  |........?.......|
00007e20  86 fa 3f 00 f7 0f 00 00  00 00 00 00 02 00 00 00  |..?.............|
00007e30  01 00 03 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00007e40  00 01 29 c2 05 00 00 00  00 00 00 00 00 00 00 00  |..).............|
00007e50  00 00 46 41 54 33 32 20  20 20 0e 1f be 77 7c ac  |..FAT32   ...w|.|
00007e60  22 c0 74 0b 56 b4 0e bb  07 00 cd 10 5e eb f0 32  |".t.V.......^..2|
00007e70  e4 cd 16 cd 19 eb fe 54  68 69 73 20 69 73 20 6e  |.......This is n|
00007e80  6f 74 20 61 20 62 6f 6f  74 61 62 6c 65 20 64 69  |ot a bootable di|
00007e90  73 6b 0d 0a 00 00 00 00  00 00 00 00 00 00 00 00  |sk..............|
00007ea0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00406600  41 66 00 69 00 6c 00 65  00 00 00 0f 00 bc ff ff  |Af.i.l.e........|
00406610  ff ff ff ff ff ff ff ff  ff ff 00 00 ff ff ff ff  |................|
00406620  46 49 4c 45 20 20 20 20  20 20 20 20 00 00 00 00  |FILE        ....|
00406630  21 00 21 00 00 00 00 00  21 00 03 00 1a 00 00 00  |!.!.....!.......|
00406640  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00407600  48 65 6c 6c 6f 20 51 45  4d 55 20 74 72 61 69 6e  |Hello QEMU train|
00407610  69 6e 67 20 77 6f 72 6c  64 0a 00 00 00 00 00 00  |ing world.......|
00407620  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
80000000
```

At the bottom here, we can even see the file data from our `echo` command as readable. Check the "Hello QEMU training world" string around address `407600` (your addresses may differ).

Restart QEMU and log in. Mount the `/dev/sda1` partition again and `cat` the contents of the file we created in the previous boot to confirm our filesystem data was preserved across boots.

```
root@Xilinx-ZynqMP-EAApr2015:~# mount /dev/sda1 /mnt
[   43.396186] FAT-fs (sda1): Volume was not properly unmounted. Some data may be
corrupt. Please run fsck.
root@Xilinx-ZynqMP-EAApr2015:~# cat /mnt/file
Hello QEMU training world
```

You may get a warning about the volume not being unmounted from before.

# 20 QEMU Device Model Development



## 20.1 Writing your own device model

Specification:

In this example, we will be creating a new basic QEMU device model. This device will be attached to the memory-mapped bus (AXI bus) in QEMU. This example device is going to have two 32-bit registers that are both read and write. The device is also going to have one interrupt.
In QEMU, every register is mapped as per real hardware. Which means it will usually have a reset value, readable/writable bits. The two registers are:

### 20.1.1 Xdata Register (Offset 0x0)

*Reset value: 0x0*

Each time the Xdata register is written, the current value is bitwise XORed with the previously written value. Example:

If the Xdata registers hold the current value of *0xFFFF0105* and software writes *0xFF00030A*, the new value will be

```
      0xFFFF0105
(+)   0xFF00030A

=     0x00FF020F
```

The Xdata register value can be read back by software as normal (without side effects).

### 20.1.2 Match Register (Offset 0x4)

*Reset value: 0xFFFFFFFF*

The match register is a 32bit register that can be read and written by software. If the Xdata register value exactly matches the match register at any time, the interrupt pin is asserted. The interrupt can be cleared by writing any value to the match register.

## 20.2 Creating the Device Model

Go to your QEMU source tree.

```
cd /path_to_QEMU/qemu
```

### 20.2.1 Create a file and add necessary #includes

Create a file *xlnx-xor-test.c* in *hw/misc* subdirectory. Open this File for editing. Paste the below `#includes` at the top of your *xlnx-xor-test.c* file:

**hw/misc/xlnx-xor-test.c**

```
#include "qemu/osdep.h"
#include "hw/sysbus.h"
#include "hw/register.h"
#include "qemu/bitops.h"
#include "qemu/log.h"
#include "qapi/error.h"
#include "hw/irq.h"
```

These above includes will give access to the various APIs we will interact with to construct our device model.

### 20.2.2 Define the model name and Err flags

**hw/misc/xlnx-xor-test.c**

```
#ifndef XOR_TEST_ERR_DEBUG
#define XOR_TEST_ERR_DEBUG 1
#endif

#define TYPE_XOR_TEST "xlnx.xor-test"
#define XOR_TEST(obj) \
    OBJECT_CHECK(XorTestState, (obj), TYPE_XOR_TEST)
```

In the above section, the `ERR_DEBUG` logic defines a symbol for debugging but defines it to 0 to disable it by default. This is useful for adding debug-only code that should be conditionally compiled in only by developers. In this example, we will keep the debug on for checking what is going on when a user reads/writes to these registers.

`TYPE_XOR_TEST` is the string name of our device. Note the value of this string, it will be used by FDT generic to match the device model to a device tree node (via the DTS `compatible` property) - more on this later.

`XOR_TEST` is what's called a QOM cast macro. It allows object casts to our new device model type.

### 20.2.3 Define registers

**hw/misc/xlnx-xor-test.c**

```
REG32(XDATA, 0x0)
REG32(MATCHER, 0x4)

#define R_MAX (R_MATCHER + 1)
```

This defines some constant symbols for our two registers. Note the offsets match our spec. Check *include/hw/register.h* for the definition of `REG32` macro to see exactly what it defines, but it will define both register index offset as well as bus address offset for each. The `R_MAX` definition is used to define the `MATCHER` register as the last register.

### 20.2.4 Define the device state struct

**hw/misc/xlnx-xor-test.c**

```
typedef struct XorTestState {
    SysBusDevice parent_obj;

    MemoryRegion iomem;
    qemu_irq irq;

    uint32_t regs[R_MAX];
    RegisterInfo regs_info[R_MAX];
} XorTestState;
```

This `XorTestState` is the device state. The physical state of the device at any given time is captured in this struct. The `parent_obj` field is used by QOM to implement the object-oriented inheritance. We won't use this at all - only core code uses this feature.
`iomem` and `irq` are our two external interfaces, for the register interface and interrupt pin respectively. `regs` is the raw state of our two registers. We can index into this array directly to get either the `xdata` or the `matcher` register.

### 20.2.5 Define irq function

**hw/misc/xlnx-xor-test.c**

```
static void xor_test_update_irq(XorTestState *s)
{
    if (s->regs[R_XDATA] == s->regs[R_MATCHER]) {
        qemu_irq_raise(s->irq);
    }
}
```

This is a private function that our code logic can call to update the IRQ. Remember from the spec that if `XDATA` matches `MATCHER`, the interrupt will assert. This inspects the device state (`s->regs`) and causes this interrupt raise should they match.

### 20.2.6 Define the *post write* function for Matcher register

**hw/misc/xlnx-xor-test.c**

```
static void xor_test_matcher_post_write(RegisterInfo *reg, uint64_t val64)
{
    XorTestState *s = XOR_TEST(reg->opaque);

    qemu_irq_lower(s->irq);
    xor_test_update_irq(s);
}
```

This function is going to be called after software writes to the `MATCHER` register. It implements the needed side effects. That is, as per the spec the interrupt is lowered for any write to the `MATCHER` register. We also call `xor_test_update_irq`, as a change in the `matcher` value could now cause the `matcher` and `xdata` to match. So we need to check for this condition. Note the update of `s->regs[R_MATCHER]` is not done here. This will be done by the core register code for us.

## 20.2.7 Define the *pre-write* function for Xdata register

**hw/misc/xlnx-xor-test.c**

```
static uint64_t xor_test_xdata_pre_write(RegisterInfo *reg, uint64_t
val64)
{
    XorTestState *s = XOR_TEST(reg->opaque);

    s->regs[R_XDATA] = s->regs[R_XDATA] ^ val64;
    xor_test_update_irq(s);

    return s->regs[R_XDATA];
}
```

This function is going to be called before software or a user writes to the `Xdata` register. It allows the insertion of logic involving the old value of the register as needed by the spec. The argument `val64` is the value as written by software.

In this code, we manually update `Xdata` as the XOR of the old value and new (as required by the spec). We check `xor_test_update_irq` as this could cause the interrupt condition to go true. We return the value written to the register as this is needed by the core code.

## 20.2.8 Define the register block

**hw/misc/xlnx-xor-test.c**

```
static RegisterAccessInfo xor_test_regs_info[] = {
    {   .name = "XDATA", .addr = A_XDATA,
        .pre_write = xor_test_xdata_pre_write,
    },{ .name = "MATCHER", .addr = A_MATCHER,
        .reset = 0xffffffff,
        .post_write = xor_test_matcher_post_write,
    },
};
```

This is the register block definition. It creates the register definitions for our two registers. The two register specific functions we just defined are defined as the pre/post write ops for our two registers as needed. The non-zero reset value (0xFFFFFFFF) of the `MATCHER` register is defined here.

### 20.2.9 Define the reset function

**hw/misc/xlnx-xor-test.c**

```
static void xor_test_reset(DeviceState *dev)
{
    XorTestState *s = XOR_TEST(dev);
    unsigned int i;

    for (i = 0; i < ARRAY_SIZE(s->regs_info); ++i) {
        register_reset(&s->regs_info[i]);
    }
    qemu_irq_lower(s->irq);
}
```

This is our `reset` function, called when the device is reset (and at least once on machine creation). The `for` loop instructs core code (`register_reset`) to reset all our register based on their defined reset values. We also lower the interrupt as this makes sense on a reset.

### 20.2.10 Define read/write handler

**hw/misc/xlnx-xor-test.c**

```
static const MemoryRegionOps xor_test_ops = {
    .read = register_read_memory,
    .write = register_write_memory,
    .endianness = DEVICE_LITTLE_ENDIAN,
    .valid = {
        .min_access_size = 4,
        .max_access_size = 4,
    },
};
```

These are the MMIO (AXI) main read and write handlers. They use the `register_read` and `register_write` functions to instruct core code to perform the read and write operations based on `xor_test_regs_info`. This is standard stuff and can be copy-pasted as-is into most Xilinx device models.

### 20.2.11 Define the *init* function

**hw/misc/xlnx-xor-test.c**

```
static void xor_test_init(Object *obj)
{
    XorTestState *s = XOR_TEST(obj);
    SysBusDevice *sbd = SYS_BUS_DEVICE(obj);

    RegisterInfoArray *reg_array;

    memory_region_init(&s->iomem, obj, TYPE_XOR_TEST,
                       R_MAX * 4);
    reg_array = register_init_block32(DEVICE(obj), xor_test_regs_info,
                              ARRAY_SIZE(xor_test_regs_info),
                              s->regs_info, s->regs,
                              &xor_test_ops,
                              XOR_TEST_ERR_DEBUG,
                              R_MAX * 4);

    memory_region_add_subregion(&s->iomem, 0x00, ®_array->mem);
    sysbus_init_mmio(sbd, &s->iomem);
    sysbus_init_irq(SYS_BUS_DEVICE(obj), &s->irq);
}
```

This is the device init function. It initiates the device state when the device is created. It sets up the dynamic device registers with the static config defined in `xor_test_regs_info`. This is standard initialization and can be copy-pasted to all Xilinx devices with little changes. It defines our registers interface and IRQ for use in the wider system (entity definition if you think in RTL).

> ⓘ Sometimes function arguments or some lines in code are split over two lines, this is to keep each line less than 80 characters long. This is required by the QEMU coding specifications.

### 20.2.12 Define *class_init*

**hw/misc/xlnx-xor-test.c**

```
static void xor_test_class_init(ObjectClass *klass, void *data)
{
    DeviceClass *dc = DEVICE_CLASS(klass);

    dc->reset = xor_test_reset;
}
```

The class init function defines our reset handler.

### 20.2.13 Define this model in an object form

**hw/misc/xlnx-xor-test.c**

```
static const TypeInfo xor_test_info = {
 .name = TYPE_XOR_TEST,
 .parent = TYPE_SYS_BUS_DEVICE,
 .instance_size = sizeof(XorTestState),
 .class_init = xor_test_class_init,
 .instance_init = xor_test_init,
};
```

This is the type of info defining this object in the inheritance hierarchy. The interesting line is `.parent`, which defines this device as being a child class of the `sysbus` device abstraction.

### 20.2.14 Register the model with QEMU core

**hw/misc/xlnx-xor-test.c**

```
static void xor_test_register_types(void)
{
    type_register_static(&xor_test_info);
}

type_init(xor_test_register_types)
```

This final logic registers the device model with the QEMU core. System-level code can now lookup this device and instantiate it as an object.
And we are done! Save xlnx-*xor-test.c*.

### 20.2.15 Add the model for compile.

Edit *hw/misc/Makefile.objs* and add the below line:

**hw/misc/Makefile.objs**

```
obj-$(CONFIG_XLNX_VERSAL)+=xlnx-xor-test.o
```

Reconfigure and rebuild QEMU using `make -j4`.

## 20.3 Adding the device to the Device Tree

Go the device tree source repo. Open the file *versal-ps-iou.dtsi*. Check the file out. You should see device tree nodes for many of the Versal ACAP peripherals. Find the UART1 controller (any peripheral will do really).
Add below lines after `serial@MM_UART1` definitions:

**versal-ps-iou.dtsi**

```
xor_test: xor_test@0xA0001000 {
        compatible = "xlnx,xor-test";
        reg = <0x0 0xA0001000 0x0 0x1000 0x0>;
};
```

Note the `compatible` string which must exactly match the string defined by `TYPE_XOR_TEST` in the *xlnx-xor-test.c* source code. The `reg` property defines the base address of the peripheral.

Save the file. Rebuild the DTB using `make`. Your device model should be ready for use.

> In the above device tree node, we assigned the module to use the address from 0xA0001000 till 0xA0001000 + 0x1000. You may provide different memory addresses but make sure that the address is not used by any other module. Check this for what may go wrong if you enter any random address: WordsofCaution

See the Device Trees page for more information on how to use device trees.

## 20.4 Adding the device for Zynq UltraScale+ MPSoC

This device can also be compiled for other Xilinx devices. Like for Zynq UltraScale+ MPSoC, add below line in same *Makefile.objs* under *hw/misc/* directory:

**hw/misc/Makefile.objs**

```
obj-$(CONFIG_XLNX_ZYNQMP)+=xlnx-xor-test.o
```

Also, add the above nodes in the Zynq UltraScale+ MPSoC device tree. For example, add the device tree node in *zynqmp-iou.dtsi* file.

## 20.5 Testing the device model:

Let's test the device model. We will test this in three ways:

### 20.5.1 Write a simple Baremetal application

Go to the *qemu-user-guide-example* repository. Under this repository find *BareMetal_examples\baremetal_new_model* directory and check the file *new_model.c*. Compile this using `make`. If having difficulties compiling this, please check baremetal compilation steps.
In this user application, we will write to the `XDATA` and `MATCHER` registers and read back the value from `XDATA` register after doing the first XOR. Check for prints in QEMU console. Launch QEMU using below commands:

```
/Path_to_your_rebuilt_qemu/qemu-system-aarch64 -nographic -M arm-generic-fdt \
    -hw-dtb Path_to_your_dts/dts/LATEST/SINGLE_ARCH/board-versal-ps-virt.dtb \
    -device loader,file= /Path_to_example_directory/new_model.elf,cpu-num=0 \
    -device loader,addr=0xFD1A0300,data=0x8000000e,data-len=4
```

It will print out all register read or written by the user application.

### 20.5.2 R/W register using GDB

Information on how to use GDB with QEMU is available in chapter 3. Launch QEMU using the below commands:

```
/scratch/devops/qemu_docs/qemu/build/aarch64-softmmu/qemu-system-aarch64 \
-M arm-generic-fdt  -serial null -serial null -serial mon:stdio -display none -s \
-hw-dtb /scratch/devops/qemu_docs/dts/LATEST/SINGLE_ARCH/board-versal-ps-virt.dtb \
-m 4G -device loader,addr=0xFD1A0300,data=0x8000000e,data-len=4
```

Now, in another terminal type the below command:

```
gdb-multiarch
```

Connect it to QEMU running in the previous windows using:

```
target remote localhost:1234
```

Once connected use below commands to read and write to register:

```
# Read command format x /x(Reading format) (0xA0001000)register address
x/x 0xA0001000
# Write command format set *(Write format) Register address =  value;
set *((int *) 0xA0001000) = 0xFFFFFF
# Try writing to MATCHER register by replacing address in above instructions.
```

Observe the QEMU window and you can see that it prints all read and write operations done to the registers.

### 20.5.3 R/W register from QEMU monitor

Launch QEMU with using below commands:

```
/scratch/devops/qemu_docs/qemu/build/aarch64-softmmu/qemu-system-aarch64 \
-M arm-generic-fdt  -serial null -serial null -serial mon:stdio -display
none -S \
-hw-dtb /scratch/devops/qemu_docs/dts/LATEST/SINGLE_ARCH/board-versal-ps-
virt.dtb \
-m 4G -device loader,addr=0xFD1A0300,data=0x8000000e,data-len=4
```

`-S` option in the above command will freeze the CPUs at startup. Press *Ctrl+a* followed by *c* to go to QEMU monitor. After type the below command:

```
x /x 0xA0001000
```

It should print something like this:

```
(qemu) x /x 0xA0001000
xlnx.xor-test:XDATA: read of value 0
00000000a0001000: 0x00000000
```

In the above example, we tried to read data from 0xA0001000 address i.e. `XDATA` register. Given that we enabled `DEBUG` flag in our model, it printed `xlnx-xor-test`(model-name)`: XDATA`(register)`: read`(operation) `of value 0`.
Let us read the `MATCHER` register. Write "*x /x 0xA0001004*" to QEMU monitor. This should print that `MATCHER` register was read with the value of 0xffffffff.

`MATCHER` register has a value of 0xffffffff even though we didn't write any value to it. Check the section "register block" and you can see we define a reset value of 0xffffffff for this register.

# 21 Using USB With QEMU

Xilinx QEMU supports USB XHCI in the host-only mode. For Versal ACAP, we have a USB2.0 controller. For Zynq UltraScale+ MPSoC, we have two USB3.0 controllers. You can plug virtual USB devices or real host USB devices. We will talk about how to use the USB host mode in the Versal ACAP and ZCU102 device.

## 21.1 USB on Versal

```
# create a dummy usb image of size 16 MB on the host machine using fallocate or qemu-
img(availble in the qemu build directory if QEMU was built from source):
fallocate -l 16M versal_usb.img
# qemu-img create versal_usb.img 16M
mkfs -t ext4 versal_usb.img

# Boot QEMU using petalinux commands:
petalinux-boot --qemu --prebuilt 3

# Login to prompt using:
username: root
password: root

# Enter QEMU monitor by using:
Ctrl+A + c.
# In monitor add following command to attach the driver:
drive_add 0 if=none,file=versal_usb.img,id=stick

# You should 'OK' being printed on screen for sucessfull drive add.

# To attach the USB device:
device_add usb-storage,bus=usb2@USB2_0_XHCI.0,port=1,id=usb_dev1,drive=stick

# Optional: Enter below command to see usb attached with QEMU or not:
info usb
# It should print this: Device 0.0, Port 1, Speed 5000 Mb/s, Product QEMU USB MSD,
ID: usb_dev1

# to find the mount path for usb, enter command:
df -h

# Above command should print something like this:
#/dev/sda                 14.5M     140.0K     13.2M    1% /run/media/sda

# Add a text file to mounted path using:
echo "Hello from QEMU" >/run/media/sda/test.txt

# Sync using:
sync

# Exit the QEMU using:
Ctrl + A + x

# On your host machine, create a directory:
mkdir test_mount

# On your host machine, mount the usb.img to this path using:
```

```
sudo mount versal_usb.img test_mount/

# On your host machine, check the text we added in this image:
cat test_mount/test.txt
```

## 21.2 USB on ZynqMP

```
# create a dummy usb image of size 16 M on the host machine using fallocate or qemu-
img(availble in the qemu build directory if QEMU was built from source):
fallocate -l 16M zynqmp_usb.img
# qemu-img create zynqmp_usb.img 16M
mkfs -t ext4 zynqmp_usb.img

# Boot QEMU using petalinux or any other way. Example for Petalinux:
petalinux-boot --qemu --prebuilt 3

# Login to prompt using:
username: root
password: root

# Enter QEMU monitor by using:
Ctrl+A + c.
# In monitor add following command to attach the driver. Please add correct path for
zynqmp_usb.img:
drive_add 0 if=none,file=zynqmp_usb.img,id=stick

# You should 'OK' being printed on screen for sucessfull drive add.

# To attach the USB device:
device_add usb-storage,bus=usb3@0xFE200000.0,port=1,id=usb_dev1,drive=stick

# Optional: Enter below command to see usb attached with QEMU or not:
info usb
# Above should print: Device 0.1, Port 1, Speed 5000 Mb/s, Product QEMU USB MSD, ID:
usb_dev1

#Exit monitor mode by using:
Ctrl + A + c

# to find the mount path for usb, enter command:
df -h

# Above command should print something like this:
# /dev/sda                 14.5M     140.0K      13.2M    1% /run/media/sda

# Add a text file to mounted path using:
echo "Hello from QEMU" >/run/media/sda/test.txt

# Sync using:
sync

# Exit the QEMU using:
Ctrl + A + x
```

```
# On your host machine, create a directory:
mkdir test_mount

# On your host machine, mount the usb.img to this path using:
sudo mount zynqmp_usb.img test_mount/

# On your host machine, check the text we added in this image:
cat test_mount/test.txt

# It should print the following:
Hello from QEMU
```

For more use cases for USB on QEMU, please check the USB emulation link.

# 22 Fault Injection in QEMU

QEMU provides a Python framework that allows users to read and write guest memory and set GPIO lines and QEMU Object Model (QOM) properties while the guest is executing.
This framework can be used on any platform in QEMU.

For this page, we will use examples directly in a Python interpreter, however they could just as easily be used in a Python program file.

Full documentation for the framework can be found here, however some information from the link will be repeated here for convenience.



## 22.1 Getting Started

### 22.1.1 QEMU arguments

In order to connect the fault injector to QEMU, QEMU must be launched with following additional command-line arguments:

| Argument | Description | Example |
|---|---|---|
| `-S` | Pause QEMU guest execution at the first instruction until resumed by the user. | `-S` |
| `-qmp <proto>:<path>,server` | Create a QEMU Machine Protocol (QMP) socket using protocol *proto* in path *path*. | `-qmp tcp:localhost:7777,server`<br>`-qmp unix:/path/to/qmp-sock,server` |

Or for a full list of example boot parameters, in this case for a single-arch Versal ACAP A72 QEMU machine:

```
qemu-system-aarch64 \
-M arm-generic-fdt \
-serial null -serial null -serial mon:stdio \
-device loader,file=<baremetal_for_versal_a72.elf>,cpu-num=0 \
-device loader,addr=0xFD1A0300,data=0x8000000e,data-len=4 \
-hw-dtb <device tree binary for Versal> \
-m <DDR memory size> \
-display none \
-qmp tcp:localhost:7777,server \
-S
```

For a full list of example boot parameters for other machines, see the Boot Examples section for multi-arch environments, and the Running Bare Metal Applications section for single-arch environments.

### 22.1.2 Connecting with Python

Once QEMU is launched, the QMP socket can be connected to.

In the `qemu/script/qmp` directory, launch your Python interpreter and import the fault injection framework and connect to QEMU:

```
$ python
>>> import fault_injection
>>> inj=fault_injection.FaultInjectionFramework("localhost:7777", 0)
Connected to QEMU 5.0.50
>>>
```

Once this is done, you are able to inject faults to the guest through the Python interpreter.

## 22.2 Commands

The framework provides the commands shown below.

The examples assume you have done the steps shown above.

> If a command needs a QOM path and property and you're not sure how to find them, the QOM tree can be printed by doing `info qtree` in the QEMU monitor. The QOM path is the ID that corresponds to the device you want to access.
>
> To find the property name, do `qom-list <QOM path>` in the QEMU monitor.

| Command | Description | Example |
|---|---|---|
| `cont()` | Resume execution when the guest is paused. | `inj.cont()`<br>Resume guest execution. |

<table>
<tr><th>Command</th><th>Description</th><th>Example</th></tr>
<tr><td><code>FaultInjectionFramework(sock, debug)</code></td><td>Constructor for the framework. Connects to socket sock and prints debug information depending on what debug is set to.
<table>
<tr><th>Debug Value</th><th>Verbosity</th></tr>
<tr><td>0</td><td>Prints command responses (if any)</td></tr>
<tr><td>1</td><td>Prints debug 0 information and timed traces, such as a read or write.</td></tr>
<tr><td>2</td><td>Prints debug 1 and 0 information, and QMP commands as well.</td></tr>
</table>
Returns the fault injection framework object.</td><td><code>inj=fault_injection.FaultInjectionFramework("localhost:7777", 0)</code><br>Creates a fault injection framework object connected to <code>localhost:7777</code> with no verbose debug printing.</td></tr>
<tr><td><code>get_qom_property(path, property)</code></td><td>Get a value from a QEMU Object Model (QOM) Property.<br>Returns the value of property property from QOM path.</td><td><code>inj.get_qom_property("wdt@0xFD4D0000", "pclk")</code><br>Returns the value of <code>pclk</code> from <code>wdt@0xFD4D0000</code>.</td></tr>
<tr><td><code>help()</code></td><td>Prints a list of commands and their descriptions.</td><td><code>inj.help()</code><br>Prints a list of commands and their descriptions.</td></tr>
<tr><td><code>notify(time_ns, cb)</code></td><td>Notify the callback cb in guest time time_ns.</td><td><code>inj.framework.notify(1000000000, write_mem_callback)</code><br>Calls the function <code>write_mem_callback</code> after 1000000000ns of guest execution time.</td></tr>
<tr><td><code>read(address, size, cpu)</code></td><td>Reads size bytes at address address accessed by CPU cpu.<br>cpu can be either a QOM path or the CPU ID.<br>Returns the value read from the address.</td><td><code>inj.read(0xFF000000, 4, 0)</code><br>Reads 4 bytes from 0xFF000000 (<code>UART0:UARTDR</code> on Zynq UltraScale+ MPSoC) accessed by CPU 0 (APU 0).</td></tr>
</table>

| Command | Description | Example |
|---|---|---|
| `run()` | Starts the guest. | `inj.run()`<br>Starts guest execution. |
| `set_gpio(path, gpio, num, value)` | Sets the GPIO *gpio* number *num* in QOM path *path* to the value *value*, where *value* is a boolean.<br>ⓘ GPIOs will be labeled as `<irq>` if using `qom-list` on a QOM path. | `inj.set_gpio("wdt@0xFD4D0000", "pwr_cntrl", 0, 1)`<br>Sets the `pwr_cntrl` GPIO in `wdt@0xFD4D0000` to 1. |
| `set_qom_property(path, property, value)` | Sets the property *property* in path *path* to value *value*. | `inj.set_qom_property("wdt@0xFD4D0000", "pclk", 1000)`<br>Sets `pclk` in `wdt@0xFD4D0000` to 1000. |
| `write(address, value, size, cpu)` | Writes value *value* of *size* bytes from CPU *cpu* to address *address*.<br>*cpu* can be either a QOM path or the CPU ID. | `inj.write(0xFF000000, 0xFF, 1, 0)`<br>Writes 0xFF to 0xFF000000 (`UART0:UARTDR` on Zynq UltraScale+ MPSoC) accessed by CPU 0 (APU 0). |

# 23 Troubleshooting

This page contains problems that may be encountered when using QEMU, and their solutions.

If you encounter problems you don't see solutions for on this page, navigate to the Xilinx Developer Forums for additional help.



## 23.1 QEMU CPU stall messages

When running Linux on top of QEMU, Linux can warn about CPU stalls. These stalls can be caused by:

- vCPU in QEMU has hung.
  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:
  This may be caused by multiple reasons related to vCPUs not getting scheduled to run. For example, due to an overloaded host, or low priority scheduling assignments to QEMU.

### 23.1.1 Solution

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

## 23.2 QEMU failed to connect socket

When running QEMU for a multi-arch environment (for example Zynq UltraScale+ MPSoC or Versal ACAP), two instances of QEMU are needed and memory needs to be shared between them.
This message can be caused by:

- QEMU being unable to create the socket
  In this case, QEMU will fail to boot.
- QEMU waiting for connection on the socket
  In this case, there should be another message that said "QEMU waiting for connection on: /tmp/socket/path".

### 23.2.1 Solution

Make sure QEMU has access to the path specified by the `-machine-path` parameter.

## 23.3 When running a Versal ACAP machine, QEMU says the machine cannot be found

This typically happens when building QEMU from source and occurs shortly after the guest starts.

In this case, it's usually because QEMU was compiled to use a different encryption library than `libgcrypt`.

### 23.3.1 Solution

Reconfigure QEMU with the `--enable-gcrypt` parameter passed in.

If `libgcrypt` is not installed, install it and then reconfigure QEMU.

## 23.4 When booting with U-Boot, it boots using an image used in a previous emulation.

When running QEMU for a multi-arch environment, the `-machine-path` directory keeps a cache of certain things. In this instance, it caches U-Boot data and it will boot with the previous U-Boot image.

### 23.4.1 Solution

If using U-Boot, clear the contents in the `-machine-path` directory between each emulation.

## 23.5 When booting QEMU, the command fails and says "-hw-dtb: invalid option"

This happens when trying to run Xilinx QEMU commands (for example, commands copied when using QEMU with PetaLinux tools) on a different version of QEMU, such as mainline QEMU.
Some options, such as `-hw-dtb`, are only supported in Xilinx QEMU.

### 23.5.1 Solution

Use Xilinx QEMU when developing for Xilinx hardware.

# 24 Known Issues

This page contains known issues and bugs with QEMU, and their solutions.

- First stage bootloader (FSBL) hangs on QEMU
- Unable to see ARM-R5 CPUs on Zynq UltraScale+ MPSoC and Versal ACAP platforms with XSDB on 2020.1 QEMU
- TFTP Put Fails on QEMU
- When using XSDB, my watchpoint was hit, but XSDB doesn't say so and my program is stopped
- When using XSDB, my program is stopped in QEMU, but XSDB says my CPUs are running
- When using a GDB remote connection to debug my program on QEMU, my program segfaults and GDB does not catch it
- Incorrect dummy cycle count with Quad IO Read (0xEB) command with Micron Flashes
- Incorrect dummy cycles sent when using GQSPI dual or quad mode byte transfer instead of CS hold time
- Versal ACAP LPD XPPU Is not controlling APU accesses to TCM

## 24.1 First stage bootloader (FSBL) hangs on QEMU

There are a few situations where the FSBL can hang in QEMU.

One is when initializing the DDR controller in `xfsbl_initialization.c`.
The DDR controller is not fully modeled in QEMU, so the FSBL will hang when the DDR controller does not behave as expected.

Another situation is because the FSBL uses `psu_init.c`, which is dynamically generated code that is changed 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.

For more information on building and customizing the FSBL, visit the Zynq UltraScale+ FSBL page.

### 24.1.1 Solution

Build a customized FSBL by commenting out the functions that cause the hangs in `psu_init.c` or `xfsbl_initialization.c`.

For PetaLinux 2018.3:

**psu_init.c**

```
unsigned long psu_ddr_phybringup_data(void)
{


    unsigned int regval = 0;

    unsigned int pll_retry = 10;

    unsigned int pll_locked = 0;


    while ((pll_retry > 0) && (!pll_locked)) {

        Xil_Out32(0xFD080004, 0x00040010);/*PIR*/
        Xil_Out32(0xFD080004, 0x00040011);/*PIR*/

    while ((Xil_In32(0xFD080030) & 0x1) != 1) {
    /*****TODO*****/

    /*TIMEOUT poll mechanism need to be inserted in this block*/

    }


        pll_locked = (Xil_In32(0xFD080030) & 0x80000000)
        >> 31;/*PGSR0*/
        //pll_locked &= (Xil_In32(0xFD0807E0) & 0x10000)
        //>> 16;/*DX0GSR0*/
        //pll_locked &= (Xil_In32(0xFD0809E0) & 0x10000)
        //>> 16;/*DX2GSR0*/
        //pll_locked &= (Xil_In32(0xFD080BE0) & 0x10000)
        //>> 16;/*DX4GSR0*/
        //pll_locked &= (Xil_In32(0xFD080DE0) & 0x10000)
        //>> 16;/*DX6GSR0*/
        pll_retry--;
    }
```

For PetaLinux 2019.1 and later:

**xfsbl_initialization.c**

```
#ifdef XFSBL_PS_DDR
#ifdef XPAR_DYNAMIC_DDR_ENABLED
    /*
     * This function is used for all the ZynqMP boards.
     * This function initialize the DDR by fetching the SPD data from
     * EEPROM. This function will determine the type of the DDR and decode
     * the SPD structure accordingly. The SPD data is used to calculate the
     * register values of DDR controller and DDR PHY.
     */
//  Status = XFsbl_DdrInit();
//  if (XFSBL_SUCCESS != Status) {
//      XFsbl_Printf(DEBUG_GENERAL,"XFSBL_DDR_INIT_FAILED\n\r");
//      goto END;
//  }
#endif
#endif
```

## 24.2 Unable to see ARM-R5 CPUs on Zynq UltraScale+ MPSoC and Versal ACAP platforms with XSDB on 2020.1 QEMU

2020.1 QEMU does not give processor information to XSDB, so XSDB does not know that these platforms have R5s on them.

### 24.2.1 Solution

Use the directions and patches found on this page to patch QEMU and XSDB.

## 24.3 TFTP Put Fails on QEMU

The TFTP put command is not supported in mainline or Xilinx QEMU for security reasons.

### 24.3.1 Solution

Use SCP or another protocol.

AR link: https://xilinx.sharepoint.com/sites/XKB/SitePages/Articleviewer.aspx?ArticleNumber=75613

## 24.4 When using XSDB, my watchpoint was hit, but XSDB doesn't say so and my program is stopped

### 24.4.1 Solution

Delete or disable the watchpoint that was hit, and then unlock the CPUs by using the `con` command.
If you're not sure which watchpoint was hit, delete or disable all of them.

AR link: https://xilinx.sharepoint.com/sites/XKB/SitePages/Articleviewer.aspx?ArticleNumber=75621

## 24.5 When using XSDB, my program is stopped in QEMU, but XSDB says my CPUs are running

### 24.5.1 Solution

This only happens when using watchpoints. If this does happen, Exit QEMU by doing `CTRL+A X`, and then restart it.
To avoid this from happening, avoid using watchpoints when debugging with XSDB.

AR link: https://xilinx.sharepoint.com/sites/XKB/SitePages/Articleviewer.aspx?ArticleNumber=75614

## 24.6 When using a GDB remote connection to debug my program on QEMU, my program segfaults and GDB does not catch it

QEMU's GDB server does not support catching the SIGSEGV signal at this time.
The GDB server can only catch SIGINT and SIGTRAP.

### 24.6.1 Solution

If possible, run GDB on the QEMU guest and debug your application using GDB on the guest.

AR link: https://xilinx.sharepoint.com/sites/XKB/SitePages/Articleviewer.aspx?ArticleNumber=75615

## 24.7 Incorrect dummy cycle count with Quad IO Read (0xEB) command with Micron Flashes

On hardware, Micron flashes expect 10 dummy cycles for a Quad IO Read (QIOR) command, but QEMU only expects 8.

### 24.7.1 Solution

Use 8 dummy cycles for a QIOR command on Micron flashes instead of 10.

AR link: https://xilinx.sharepoint.com/sites/XKB/SitePages/Articleviewer.aspx?ArticleNumber=75600

## 24.8 Incorrect dummy cycles sent when using GQSPI dual or quad mode byte transfer instead of CS hold time

When using GQSPI, it is possible to use a byte transfer to send dummy cycles instead of using CS hold time.

QEMU does not emulate link state for GQSPI commands, so if sending 1 byte using quad or dual mode, QEMU will send 1 cycle to flash instead of 2 or 4 respectively.

### 24.8.1 Solution

Use CS hold time for the amount of dummy cycles you need for your command instead of transferring bytes.

AR link: https://xilinx.sharepoint.com/sites/XKB/SitePages/Articleviewer.aspx?ArticleNumber=75599

## 24.9 Versal ACAP LPD XPPU Is not controlling APU accesses to TCM

On hardware, LPD XPPU can control accesses to TCM, however this behavior is not implemented in QEMU. This is due to how the XPPU is implemented in QEMU, and the possibility of LPD XPPU blocking APU and RPU accesses to TCM.

### 24.9.1 Solution

There is no known workaround for this at the moment.

AR link: https://xilinx.sharepoint.com/sites/XKB/SitePages/Articleviewer.aspx?ArticleNumber=75684

# 25 Acronyms

## 25.1 Quick Jump

### 25.1.1 Acronyms#A

Acronyms#B
Acronyms#C
Acronyms#D
Acronyms#E
Acronyms#F
Acronyms#G
Acronyms#H
Acronyms#I
Acronyms#J
Acronyms#K
Acronyms#L
Acronyms#M
Acronyms#N
Acronyms#O
Acronyms#P
Acronyms#Q
Acronyms#R
Acronyms#S
Acronyms#T
Acronyms#U
Acronyms#V
Acronyms#W
Acronyms#X
Acronyms#Y
Acronyms#Z

## 25.2 Acronym Table

| Acronym | Meaning |
|---|---|
| A | |
| ACAP | Adaptive Compute Acceleration Platform |

| | |
|---|---|
| ACE | AXI Coherency Extension |
| APM | AXI Performance Monitor |
| APU | Application Processing Unit (ARM A series) |
| ATF | ARM Trusted Firmware |
| ATM | AXI Trace Monitor |
| AXI | Advanced eXtensible Interface |
| B | |
| BBRAM | Battery-Backed RAM |
| BSP | Board Support Package |
| C | |
| CAN | Controller Area Network (bus) |
| CAN FD | Controller Area Network Flexible Data-Rate (bus) |
| CCI | Cache-Coherent Interconnect |
| CHI | Coherent Hub Interface |
| CPM | Coherent PCIe Module |
| D | |
| DHCP | Dynamic Host Configuration Protocol |
| DNS | Domain Name System |
| DT | Device Tree |
| DTB | Device Tree Binary |
| DTC | Device Tree Compiler |
| DTS | Device Tree Source |

| | |
|---|---|
| DUT | Device Under Test |
| E | |
| EEPROM | Electrically Erasable Programmable Read-Only Memory |
| eMMC | embedded MultiMediaCard |
| F | |
| FDT | Flattened Device Tree |
| FPD | Full Power Domain |
| FPDDMA | Full Power Domain Direct Memory Access |
| FPGA | Field Programmable Gate Array |
| FSBL | First Stage BootLoader |
| G | |
| GEM | Gigabit Ethernet Module |
| GICvX | Generic Interrupt Controller version X |
| GQSPI | Generic QSPI |
| GTK | GNOME ToolKit |
| H | |
| HIL | Hardware In Loop |

| | |
|---|---|
| I | |
| J | |
| K | |
| L | |
| LAN | Local area network |
| LPD | Low Power Domain |
| LPDDMA | Low Power Domain Direct Memory Access |
| LQSPI | Legacy QSPI |
| M | |
| MTTCG | Multi-Threaded Tiny Code Generator |
| N | |
| NFS | Network File System |
| NOC | Network On a Chip |
| O | |
| OCM | On-Chip Memory |
| OSPI | Octal SPI |

| | |
|---|---|
| **P** | |
| PCIe | Peripheral Component Interconnect Express |
| PL | Programmable Logic |
| PMU | Platform Management Unit Controller |
| POSH | Posh Open Source Hardware |
| PPU | Platform Management Controller Processing Unit |
| PS | Processing System |
| PSM | Processing System Manager |
| **Q** | |
| QEMU | Quick EMUlator |
| QOM | QEMU Object Model |
| QMP | QEMU Machine Protocol |
| QSPI | Quad SPI |
| **R** | |
| RCU | ROM Code Unit |
| RPU | Realtime Processing Unit (ARM R series like R5) |
| **S** | |
| SCP | Secure Copy Protocl |
| SD | Secure Digital (card) |
| SDL | Simple DirectMedia Layer |

| | |
|---|---|
| SLCR | System-Level Control Register |
| SMMU | System Memory Management Unit |
| SPI | Serial Peripheral Interface |
| SSH | Secure Shell |
| SWDT | System WatchDog Timer |
| SYSMON | SYStem MONitor |
| T | |
| TAP | Terminal Access Point |
| TBU | SMMU Translation Buffer Unit |
| TFTP | Trivial File Transfer Protocol |
| TLM | Transaction Level Modeling (SystemC) |
| TTC | Triple Timer Counter |
| U | |
| V | |
| VFIO | Virtual Function I/O |
| W | |
| WDT | WatchDog Timer |
| WWDT | Window WatchDog Timer |

| X | |
|---|---|
| XDMA | Xilinx DMA IP |
| XMPU | Xilinx Memory Protection Unit |
| XPPU | Xilinx Peripheral Protection Unit |
| XRAM | Accelerator RAM |
| XSCT | Xilinx Software Command-line Tool |
| XSDB | Xilinx System Debugger |
| Y | |
| Z | |

# 26 Additional Resources

This section contains additional Xilinx resources that are useful for development



## 26.1 Xilinx Resources

For support resources such as answers, documentation, downloads, and forums, see Xilinx Support.

Xilinx also provides a video that gives a brief overview of QEMU:

Sorry, the widget is not supported in this export.
But you can reach it using the following URL:

https://www.youtube.com/watch?v=ifpmJ5y5rfs

## 26.2 Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle.
Topics include design assistance, advisories, and Vivado Design Suite Documentation.

## 26.3 Documentation Navigator and Design Hubs

Xilinx Documentation Navigator provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information.
Any of the following methods can be used to open the Xilinx Documentation Navigator (DocNav):

- In Vitis, select Help → Xilinx OS and Libraries Help
- 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 the Xilinx Documentation Navigator, click the Design Hubs View tab.
- On the Xilinx website, see the Design Hubs page.

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

## 26.4 Mainline QEMU Resources

The mainline QEMU landing page can be found here.

For documentation on mainline QEMU, visit this page.

# 27 Document References

- Xilinx Documentation
- Other References

## 27.1 Xilinx Documentation

- Zynq UltraScale+ MPSoC Software Developers Guide (UG1137)
- Zynq UltraScale+ MPSoC Technical Reference Manual (UG1085)
- Zynq UltraScale+ Registers User Guide (UG1087)
- UltraScale Architecture and Product Overview (DS890)
- Xilinx Software Developer Kit Help (Includes XSDB) (UG782)
- OS and Libraries Document Collection (UG643)
- Xilinx Third-Party Licensing Guide (UG763)
- Versal ACAP System Software Developers Guide (UG1304)
- Versal ACAP Technical Reference Manual (AM011)
- Versal ACAP Design Guide (UG1273)
- PetaLinux Tools
- Vivado Design Suite Documentation
- Vitis Documentation
- UltraScale Architecture

## 27.2 Other References

- Zynq MPSoC XEN Wiki
- ARM Information Center
- Using Git
- Mainline QEMU GitHub
- Xilinx QEMU Github
- Upstream QEMU user guide
- OpenAMP Wiki
- libsystemctlm-soc and TLM-2.0 Co-simulation Demo Repository
- Device Tree Repository