Avery QEMU Co-Simulation

Volume 1: Reference Manual

Release 1.3

Contents

1. Getting Help

1.1. Support

If user needs assistance with any aspect of Avery Design Systems' products, user may contact Avery Design Systems using one of the following methods.

Web site		http://www.avery-design.com
Telephone	USA	(978) 851-3627
	Taiwan	+886 2 2327-8766
Email		support_emu@avery-design.com
Bugzilla		http://www.avery-design.com.tw/bugs
Sales		http://www.avery-design.com (Click Contact)

Table 1.1 Contact Information

1.2. Documentation Differences

Revision	**History**
1.0	Initial version of the manual
1.1	Add embedded platform information Add creating ELF and DTB files method
1.3	Adding Section 3.5 KVM Driver Patch to Support MMIO Atomic in QEMU

Table 1.2 List of Documentation Differences

2. Preparing and Running QEMU

Both statically and dynamically linked QEMU executable could be provided in the release, KVM acceleration functionality is recommended to optimize the execution speed of the Co-simulation. User is expected to compile the kernel version needed to suit their development requirements. On the QCOW disk image, Ubuntu Linux version 12.04.5 or above is recommended to be installed then updating to the corresponding Kernel version.

2.1. Installing 3rd_party Dependencies

If dynamically linked QEMU is provided, please install third-party libraries and dependencies required by QEMU.

```
#> yum install qemu-kvm virt-manager virt-viewer libvirt-bin glib2-devel.x86_64 \
pixman-devel.x86_64 chrpath socat xterm zlib-devel libgcrypt-devel libaio-devel \
libibumad-devel libzstd-devel
```

Figure 2.1 Installing third-party libraries

VNCViewer needs to be installed to enable users to interact with the guest OS once booted-up.

```
#> yum install tigervnc.x86_64
```

Figure 2.2 Installing VNCViewer

If Perl regression script is provided in the release, some extension libraries may be needed.

```
#> yum install perl-TermReadKey
```

Figure 2.3 Installing Perl TermReadKey library

2.2. Creating QCOW with OS Image

QCOW is a file format for disk image files used by QEMU. Use the “qemu-img” command to create a QCOW image.

```
#> qemu-img create -f qcow2 <image name> <size>
```

Figure 2.4 Creating a QCOW image

After creating the QCOW, users are able to install OS on the disk image. The option “-os_img” is provided to allow QEMU to boot from the CD-ROM with an OS installation image.

```
#> <QEMU run script> -os_img <location of OS img> -qc <location of QCOW image>
```

Figure 2.5 Boot with an OS installation image

Users may use VNCViewer to get graphic display of QEMU while installing their own OS:

```
#> vncviewer :5900
```

Figure 2.6 Use VNCViewer to get graphic display of QEMU

2.2.1. Creating QCOW Snapshot

The preparation of a QCOW image usually takes some time including installing the corresponding Guest OS, tools and libraries required to run users' application and firmware, a QCOW image takes more than 10GB of disk space and can only support one QEMU usage at a time.

To create a consistent environment and allow multiple users to run the QCOW image files with the same content, snapshots can be created from the base image file, multiple snapshots will refer to the same base image and record the differences.

```
#> qemu-img create -f qcow2 -b <QCOW image> <snapshot name>
```

Figure 2.7 Creating snapshot of a QCOW image

When snapshots are created, users are expected to keep the base image as a read only file, all QEMU should be booted using the created snapshot instead of the read only base image to avoid snapshots to be corrupted when the base image is modified.

2.3. OVMF Option

Please be noted that the BIOS installed on the provided QCOW image should be consistent with the BIOS used to boot QEMU machine, SeaBIOS is used by QEMU as default. However, newer BIOS version UEFI is downloaded from tianocore's git repository and used to boot Avery's QEMU co-simulation, the UEFI files are under $AVERY_QEMU/tools/3rd_party/edk2-ovmf.
Link here: https://github.com/tianocore/edk2.git

Users can specify UEFI used by invoking -ovmf option with their own UEFI built when launching the QEMU run script.

2.4. Creating source files for embedded platform

To execute embedded platform with qemu users need several source files like, ELF file and DTB file.

ELF file and DTB file are related. Users have to confirm that ELF file corresponds to DTB file.

2.4.1. Creating ELF file

To generate ELF file users will need to confirm which embedded architecture is in used.

Users should create minimum three files. One is linker script, main.c and makefile.

For example, if embedded architecture is microblaze then users need to use microblaze-xilinx-elf-gcc cross compiler to generate elf file. Please refer to Figure 2.8.

```
<aqxxx_sim-xxxx_xxxx>
├── aqemu
│   ├── example
│   │   └──
kc705
│   │       :
│   │       ├── src
│   │       │   :
│   │       │   └── elf
│   │       │       ├── Debug
│   │       │       │   ├── kc705.elf
│   │       │       │   ├── kc705.elf.size
│   │       │       │   ├── makefile
│   │       │       │   ├── objects.mk
│   │       │       │   ├── sources.mk
│   │       │       │   └── src
│   │       │       │       :
│   │       │       ├── lib
│   │       │       │   :
│   │       │       └── src
│   │       │           ├── lscript.ld
│   │       │           └── main.c
:   :       :
```

Figure 2.8 ELF example source files tree

2.4.2. Creating DTB file

To generate DTB file users will need to confirm which hardware is in used and remote-port setup and use tool dtc to generate dtb file. Please refer to Figure 2.9. If users not sure remote-port setup, please mail us. The DTS/DTSI example Please refer to Figure 2.10

```
#>dtc -I dts -O dtb -o <output dtb file name> <input dts file>
```

Figure 2.9 Creating DTB file

```
<aqxxx_sim-xxxx_xxxx>
├── aqemu
│   ├── example
│   │   └── kc705
│   │       ├── src
│   │       │   ├── dtc
│   │       │   │   ├── kc705-remoteport.dtsi
│   │       │   │   ├── pl.dtsi
│   │       │   │   └── system-top.dts
:   :       :   :
```

Figure 2.10 DTS/DTSI example source files tree

2.5. Checking for CPU KVM Support

The Intel VT extensions can be disabled in the BIOS. The virtualization extensions cannot be disabled in the BIOS for AMD-V.

Verify the virtualization extensions are enabled in BIOS. The BIOS settings for Intel® VT or AMD-V are usually in the Chipset or Processor menus. The menu names may vary from this guide, the virtualization extension settings may be found in Security Settings or other nonstandard menu names.

Procedure to enabling virtualization extensions in BIOS:

1. Reboot the computer and open the system's BIOS menu. This can usually be done by pressing the delete key, the F1 key or Alt and F4 keys depending on the system.
2. Select Restore Defaults or Restore Optimized Defaults, and then select Save & Exit.
3. Power off the machine and disconnect the power supply.
4. Enabling the virtualization extensions in BIOS.
5. Power on the machine and open the BIOS.
6. Open the Processor submenu, the processor settings menu may be hidden in the Chipset, Advanced CPU Configuration or Northbridge.
7. Enable Intel Virtualization Technology (also known as Intel VT) or AMD-V depending on the brand of the processor. The virtualization extensions may be labeled Virtualization Extensions, Vanderpool or various other names depending on the OEM and system BIOS.

8. Enable Intel VTd or AMD IOMMU, if the options are available. Intel VTd and AMD IOMMU are used for PCI passthrough.
9. Select Save & Exit.
10. Power off the machine and disconnect the power supply.
11. Run cat /proc/cpuinfo | grep vmx svm. If the command outputs, the virtualization extensions are now enabled. If there is no output user's system may not have the virtualization extensions or the correct BIOS setting enabled.

2.5.1. Checking for KVM Device Ownership

Device node KVM under /dev should be accessible to user accounts to allow users to accelerate QEMU simulation using the KVM hardware. To check ownership of the KVM device node:

```
#>ls -al /dev/kvm
crw-rw-rw-+ 1 root kvm 10, 232 Mar  5 15:00 /dev/kvm
```

Figure 2.9 Check ownership of KVM node

Users should have read write access to the KVM device not, if not, please contact system administrator of the current machine.

2.6. Xilinx fork of Quick EMUlator (QEMU)

Xilinx actively develops a QEMU tree forked off the latest QEMU git repo that models the hardware components of its MPSoC FPGAs and reference boards including the processor subsystem (PS) for Microblaze, Zynq, and Zynq UltraScale+. All the components of the FPGA and boards are modeled as QEMU devices; however, the Xilinx QEMU supports a SystemC co-simulation capability to allow users to develop SystemC models of their own programmable logic (PL) designs and run in the overall QEMU virtual machine environment. Official documentation can be found here:

https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842060/QEMU

The git repo is as follows:

https://github.com/Xilinx/qemu

SystemC co-simulation is one of the key functionality Xilinx enhanced from the original QEMU repository. Xilinx created a framework of virtual ports for their own customized devices to pass down their traffic for their SystemC model.

Avery has extended this framework to add co-simulation with Avery's PCIe Root Complex BFM VIP running in any SystemVerilog simulator which allows users to test their PCIe-based DUT using behaviors initiated by user applications, the BIOS and OS kernel (Linux, Windows®), and their DUT. Avery has added numerous customizations and enhancements to the Xilinx QEMU and therefore provides a pre-compiled version as a part of Avery product releases kits.

2.7. Using Customized Linux Kernel for QCOW Image

To use user's Linux kernel instead of the built-in kernel inside QCOW image, compiled kernel image and ramdisk file are required. User can either build them from source or download them online.

2.7.1. Building Kernel and ramdisk from source

To compile the kernel from source, please follow the commands below:

```
#>cd <linux-src-path>
#>make defconfig
#>sudo make bzImage -j`nproc`
```

Figure 2.10 Creating a compiled kernel

A ramdisk file is used as a temporary root file system to mount the root device while booting and loading the corresponding kernel. Hence each kernel image used as a pass-in file requires a

corresponding ramdisk file for QEMU to boot properly. Following the steps in figure 2.11 and figure 2.12 is one of the methods on how to generate a ramdisk file for user's kernel image.

```
#>cd <kernel source directory>
#>sudo make modules -j`nproc`
#>sudo make modules_install

sh ./arch/x86/boot/install.sh \
          5.14.0-rc6+ arch/x86/boot/bzImage \
          System.map "/boot"
dkms: running auto installation service for kernel 5.14.0-rc6+
```

Figure 2.11 Installing kernel modules

After the modules are installed, please check the kernel version number displayed in the output message. For example, the kernel version number is 5.14.0-rc6+ in Figure 2.11.

Please note that the <kernel version-number> used in the following commands should match with the kernel version number installed. Additionally, initrd-<kernel version number>.img in figure 2.12 is the generated initrd file.

```
#>cd <initrd  rd directory>
#>sudo mkinitrd initrd-<kernel version number>.img <kernel version number>
```

Figure 2.12 Making the initrd file

User can specify the path of the kernel image and the ramdisk file while running QEMU. Additionally, -ovmf option is not required.

```
#><QEMU run script> -kernel <kernel-file-path> -rd <ramdisk-file-path>
```

Figure 2.13 Runtime arguments for specifying kernel and ramdisk

Mounted root directory that used to drive OS boot process, can be either specified by user or /dev/map/ubuntu—vg—ubuntu-lv by default. Furthermore, user can check the file system disk space statistics by "df -h" command for finding the mounted root directory.

```
#><QEMU run script> -kernel <kernel file path> -rd <ramdisk file path> -mroot <mounted root directory>
```

Figure 2.14 Runtime arguments for specifying kernel, ramdisk and mounted root

2.7.2. Using Pre-built Linux Kernel for QCOW Image from Cloud

User can also use pre-built cloud images as the source of the compiled kernel image and ramdisk. The following steps show how to do so by using the officially released ubuntu 20.04 server cloud images.

To download the required files, please follow the commands below.

```
#>wget https://cloud-images.ubuntu.com/releases/focal/release-20210125/unpacked/ubuntu-20.04-server-cloudimg-amd64-vmlinuz-generic
#>wget https://cloud-images.ubuntu.com/releases/focal/release-20210125/unpacked/ubuntu-20.04-server-cloudimg-amd64-initrd-generic
```

Figure 2.15 Commands for downloading the compiled kernel and ramdisk file

After downloading the files, user can follow the commands in figure 2.13 and figure 2.14 to run QEMU.

2.8. Using Customized Linux Kernel for QCOW Image

In addition to using QCOW image for running QEMU, user can also use pre-built cloud image as the source of disk image, compiled kernel and ramdisk. The following steps show how to do so by using the officially released ubuntu 20.04 server cloud images.

To download the required files, please follow the commands below.

```
#>wget https://cloud-images.ubuntu.com/releases/focal/release-20210125/unpacked/ubuntu-20.04-server-cloudimg-amd64-vmlinuz-generic
#>wget https://cloud-images.ubuntu.com/releases/focal/release-20210125/unpacked/ubuntu-20.04-server-cloudimg-amd64-initrd-generic
#> wget https://cloud-images.ubuntu.com/releases/focal/release-20210125/unpacked/ubuntu-20.04-server-cloudimg-amd64-initrd-generic
```

Figure 2.16 Commands for downloading cloud images

To resize the image to 10G, please follow the command below.

```
#>  qemu-img resize ubuntu-20.04-server-cloudimg-amd64.img 10G
```

Figure 2.17 Commands for resizing cloud image

To create the disk image with user-data to be used for starting the cloud image, please follow the commands below. In this example, user can log into the image with account: ubuntu and password: pass

```
#>sudo yum install cloud-utils (or sudo apt-get install cloud-image utils)
#> cat >user-data <<EOF
# cloud-config
password: pass
chpasswd: {expire: False}
ssh_pwauth: True
EOF
#>cloud-localds user-data.img user-data
```

Figure 2.18 Commands for creating disk image

To run QEMU with pre-built images, user can follow the command below.

```
#><QEMU run script> -cimg  <cloud-image-path> -dimg <disk-image-path> -kernel <kernel-file-path> -rd <ramdisk-file-path>
```

Figure 2.19 Commands for creating disk

3. Debugging and Developing on QEMU

When developing on QEMU guest OS, any Linux built-in tools or custom application should behave the same as targeting a corresponding device node.

Built-in applications like devmem2 can be used to access PCIe BAR and memory address of AXI device, meanwhile lspci and setpci can be used to access the config space of the device. User can use any existing applications and debugger to aid custom application development.

3.1. Compiling and Installing Linux Kernel

Latest Kernel can be cloned from Linus' office git repository:

https://github.com/torvalds/linux

Copy the config setting of the current machine to the .config file under the cloned repository.

```
#>cp /boot/config-`uname -r` <LINUX_DIR>/.config
```

Figure 3.1 Command to copy current machine configurations

User can also modify the config variables by modifying the .config file directory or invoke the config menu, to invoke the config menu, use:

```
#>make menuconfig
```

Figure 3.2 Commands to invoke graphic configuration menu

After setting up the .config file, compile the Linux Kernel. The compilation could take roughly an hour and under the compiled directory, users should be able to see compiled debian packages appeared as linux-image-*kernel version*.deb, install the header and the debian package using the dpkg command and reboot.

```
#>make -j`nproc` deb-pkg
#>sudo dpkg -i linux-image-3.13.11-ckt39_3.13.11-ckt39-1_amd64.deb
#>sudp dpkg -i linux-headers-3.13.11-ckt39_3.13.11-ckt39-1_amd64.deb
```

Figure 3.3 Commands to reinstall image and header files

3.2. Creating Dynamic Kernel Module

To compile the loadable driver module, user must create their own Makefile under the driver directory.

```
# Modify the Makefile to point to Linux build tree.
DIST ?= $(shell uname -r)
KDIR:=/lib/modules/$(DIST)/build/
PWD:=$(shell pwd)
DRV_NAME:=<module name>
SOURCES := <module source*.c>

obj-m := <module name.o>
<module name>-objs= <module source* .o>

all:
	make -C $(KDIR) M=$(PWD) modules

clean:
	make -C $(KDIR) M=$(PWD) clean
```

Figure 3.4 An example of dynamic module makefile

Compile the kernel, remove the default driver, and insert the modified driver, command is shown using nvme as an example. Removing and inserting the driver might need root access.

```
#>rmmod nvme; insmod nvme.ko
```

Figure 3.5 Remove and insert kernel module

Using dmesg command to display the kernel debug messages. Piping out the log.

```
#>dmesg | tee /tmp/dmesg.txt
    cid 0 nsid 1 metadata 0 prp1 2720f6000 prp2 0 slba 21
```

Figure 3.6 Display debug and save log

The information displayed in dmesg log can be combined with Avery's VIP debug tracker files mentioned in VIP specific manual to trace the behaviors throughout the entire system.

3.3. Debugging Using GNU Debugger

GNU debugger is an application for finding out how C/C++ programs run or for analyzing the moment program crashes. Users can run, stop, set breakpoints under specific conditions, analyze the situations, make modifications, and test code changes using the GNU debugger. As a commonly used and powerful tool, users can run the GNU debugger inside the Guest OS to debug

their application, as well as using the Kernel GNU debugger to debug their own kernel drivers when running the co-simulation.

To invoke GDB debugger, in Guest OS, run GDB with applications compiled with debug symbols, while running GDB to debug Kernel will be slightly more complicated.

3.3.1. GDB Debugging Kernel Drivers

First, the Linux Kernel used must be compiled with debug symbols, modify the following variables under the .config file of the kernel source code:

CONFIG_DEBUG_KERNEL=y CONFIG_DEBUG_INFO=y CONFIG_CONSOLE_POLL=y CONFIG_KDB_CONTINUE_CATASTROPHIC=0 CONFIG_KDB_DEFAULT_ENABLE=0x1	CONFIG_KDB_KEYBOARD=y CONFIG_KGDB=y CONFIG_KGDB_KDB=y CONFIG_KGDB_LOW_LEVEL_TRAP=y CONFIG_KGDB_SERIAL_CONSOLE=y

Figure 3.7 Config variables to enable debug

Compile the Linux Kernel following 4.2. Compiling and Installing kernel. Restart aqcxl_qemu.pl with the -kgdb option, while boot up, at the GRUB menu, add kgdb options “kgdbwait kgdboc=ttyS0,115200 nokaslr” to the corresponding line, to allow OS to wait for kgdb while booting, press F10 to boot:

```
linux /boot/vmlinuz-<user's linux kernel version> root=UUID=be4127bb-0309-414b-8b11 c-
c2631b6fdcb5 ro  maybe-ubiquity kgdbwait kgdboc=ttyS0,115200 nokaslr
```

Figure 3.8 Commands to introduce kgdb wait

While OS boots using the modified GRUB, on Host OS, invoke gdb with the vmlinux symbol file, the connection between gdb and the QEMU Guest OS should be established automatically. Port 4321 is used as the default port for the Guest OS to communicate with the kgdb, specified by the aqcxl_qemu.pl script.

```
#>gdb vmlinux -ex 'target remote localhost:4321'
```

Figure 3.9 Commands to launch kgdb with local port

QEMU virtual machine should stop displaying “[0] kdb>” and wait for gdb to continue the boot process, users can set breakpoints to corresponding kernel function and key in “c” in gdb to continue the boot process to debug.

3.4. Debugging BIOS on QEMU

QEMU provides functionalities for BIOS debugging, here is a list of files to be prepared and sets to follow to obtain the BIOS debug messages.

A custom BIOS with debug message turned on, or default SeaBIOS QEMU uses. The official SeaBIOS github page is maintained by coreboot.

https://github.com/coreboot/seabios

Users can download the BIOS, modify and customize for their own usages.

```
#>git clone https://github.com/coreboot/seabios
#>cd seabios; make menuconfig
```

Figure 3.10 Cloning the SeaBIOS from git

A graphic config menu will appear, to enable debug message, <Select> -> Debugging-> <Select> -> Debug level -> change the value to value larger (3 for example) -> <Ok> -> <Save> -> <Exit> -> <Exist> -> <Yes>.

```
#>make
  Compiling checking out/src/misc.o
  ...
  …
  Creating out/bios.bin
```

Figure 3.11 Compiling the BIOS

Set the corresponding compiled BIOS location with the QEMU runtime option -bios, and run QEMU using -dbg_bios and -bios.

```
#><QEMU run script> -dbg_bios -bios <BIOS LOCATION>/bios.bin
```

Figure 3.12 Runtime arguments to display BIOS debug

A fifo file qemudebigpipe will be generated under the run directory, cat the fifo for the debug message printed. Please be noted that QEMU will pause from execution once the fifo is full, make sure to clean the fifo if -dbg_bios option is used.

```
  $ cat ./qemudebugpipe | tee qemu_bios.txt
```

Figure 3.13 Cat the fifo and tee the messages into a file

3.5. KVM Driver Patch to Support MMIO Atomic in QEMU

Avery noticed an atomic bug while using QEMU when KVM mode is enabled, this bug separates atomic functions like fetch_add() into a read followed by a write. Which causes commands to lose atomicity during MMIO traffic.

To address this issue, Avery proposed a patch to the KVM kernel driver that can be applied to any Linux kernel of user's choice, below demonstrates on how to apply such patch so user can directly apply it to their host machine.

To apply Avery's patch to your Linux Kernel (v6.2 is used as an example), follow these steps:

Clone Linux Kernel v6.2 from the official Linux git page:

```
#>git clone https://github.com/torvalds/linux/tree/v6.2
```

Figure 3. 11 Cloning the Linux Kernel tag v6.2 from git

This creates a local copy of the kernel source code you can modify.
Navigate to `linux/arch/x86/kvm/x86.c` and locates line 7603-7607. Apply the following modifications:

```
7603      vcpu->run->mmio.len = min(8u, vcpu->mmio_fragments[0].len);
7604      vcpu->run->mmio.is_write = vcpu->mmio_is_write = ops->write;
7605+      vcpu->run->mmio.lock_prefix = ctxt->lock_prefix;
7606      vcpu->run->exit_reason = KVM_EXIT_MMIO;
7607      vcpu->run->mmio.phys_addr = gpa;
```

Figure 3.12 Applying Avery MMIO KVM Patch

Navigate to `linux/include/uapi/linux/kvm.h` and locates line 335-341. Apply the following modifications:

```
335           __u8 data[8];
336           __u32 len;
337           __u8 is_write;
338+           __u8 lock_prefix;
339       } mmio;
340       /* KVM_EXIT_HYPERCALL */
341       struct {
```

Figure 3. 13 Applying Avery MMIO KVM Patch

Compile the modified Linux kernel according to Section 3.1 (Compiling and Installing Linux Kernel). Once users have installed the modified kernel on their host machine, users should be able to run QEMU with KVM enabled without experiencing the atomic bug. Note that users may need to consult their IT department before applying these changes.

To verify the effectiveness of the above modifications and observe their outcomes, it is recommended to use Avery's AQCXL_SIM co-simulation as the verification platform. To test CXL device memory access with atomic operation, users can either run an existing test from the CXL Performance Development Kit (CPDK) or create a custom scenario involving multiple threads participating in atomic behavior simultaneously to test their atomicity. The HelloWorld test provided by CPDK is ideal for testing this feature, as it involves multiple threads participating in fetch_add() behavior towards the same object stored inside the CXL device memory.

If the bug condition is met and fixed in QEMU provided by Avery's AQCXL_SIM co-simulation package, the following print should appear:

```
[QEMU] Atomic operation detected, PC= 0x557fb04577e7, hwaddr= 1930010c0, is_rw= 1
```

Figure 3. 14 QEMU Print Indicating Atomic Fix

This printout, as shown in Figure 3.14, confirms the effectiveness of the patch addressing the atomic instruction bug, displaying the program counter, hardware address, and transaction type.