# Xen Zynq Distribution

## *User's Manual*

**Xilinx-XenZynq-DOC-0001 v0.14 January 6, 2017**

DORNERWORKS

## Revision History

The following table shows the revision history for this document.

| Date MM/DD/YYYY | Version | Changes |
|---|---|---|
| 03/30/2015 | 0.1 | Converted Alpha Release Document using the Xilinx Template |
| 04/22/2015 | 0.2 | Updated steps and release to work with the beta version of PetaLinux |
| 06/22/2015 | 0.3 | Fixed the numbering scheme and added a section on non-Linux guests |
| 09/24/2015 | 0.4 | Added pass-through and alternative guest file system sections |
| 02/16/2015 | 0.5 | Using v2015.4 of Xilinx Tools and added support for the Xilinx ZCU102 board. |
| 03/30/2016 | 0.6 | Minor wording changes for clarity. |
| 06/30/2016 | 0.7 | Updated for v2016.1 of Xilinx Tools; added sections covering SD card boot, xentop, and shared memory.<br>XZD_20161231 |
| 07/14/2016 | 0.8 | Updated for v2016.2 of Xilinx tools. |
| 07/21/2016 | 0.9 | Revised SD card boot section (4.2.2.1) for clarity. |
| 08/05/2016 | 0.10 | Removed unused option from xen,dom0-bootargs. |
| 09/26/2016 | 0.11 | Revamped Chapter1, added Paravirtualization section to Chapter 8, and made misc proofread edits. |
| 09/27/2016 | 0.12 | Added Bare Metal Container steps to Chapter 7. |
| 09/30/2016 | 0.13 | Added Chapter 8 to reference other guests, specifically FreeRTOS.<br>Minor corrections in section 4.2.2.1 and Chapter 7. |
| 01/06/2017 | 0.14 | Minor corrections.<br>Updated for v2016.3.<br>Added Ubuntu 16.04 as a recommended host. |

# Table of Contents

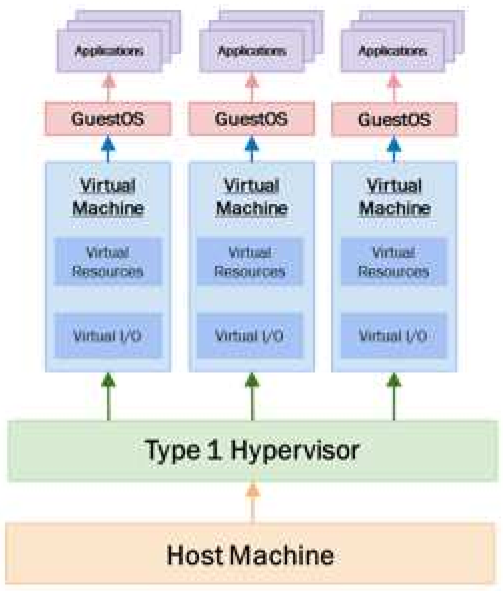
## 1.1. Introduction to Xen Zynq Distribution

The Xen Zynq Distribution (XZD), is the port of the Xen open source hypervisor to the Xilinx Zynq UltraScale+ MPSoC. All components of the XZD except for the Bare Metal Container (BMC) are released under the GNU General Public License version 2 (GPLv2); the BMC is released under the FreeBSD license. You are free to make modifications and derivative products for private use without having to make your source code changes available to others. If you decide to release any modified, or derivative, components other than the BMC, then the GPLv2 license requires that you make the source code, including your modifications and additions, available to the users of the modified software. Note that the software running in the virtualized environments that Xen creates are not a derivative XZD product, and so therefor are not subject to the GPLv2 licensing requirements. However, such software, or constituent components thereof such as the Operating System (OS), may be subject to their own licensing restrictions.

### *1.1.1. Introduction to Xen and Virtualization*

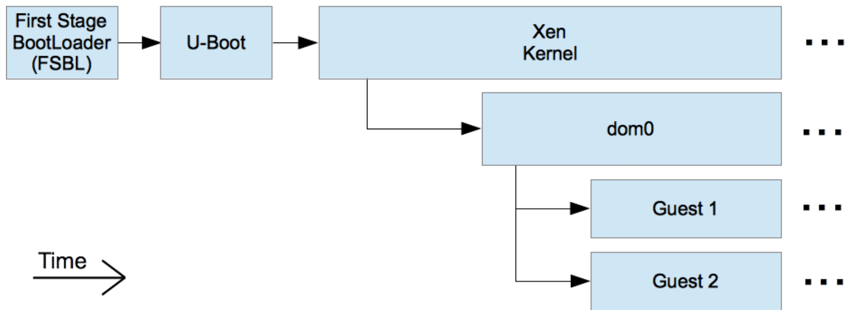
Xen is a type-1 hypervisor. Xen can run on Intel and ARM platforms with hypervisor hardware extensions, which include ARMv8 and some ARMv7 processors. Xen on Intel platforms has an extensive history providing cloud services. Amazon Web Services alone runs ½ million virtualized Xen instances [1]. Work on adding ARM support for Xen was started in 2008, and made it to the mainline in 2013. The release of multicore SoC using the 64-bit ARMv8 architectures in 2015 and 2016 marked an inflection in the embedded market similar to one experienced by the server market in the 1990's, where system designers begin to struggle with being able to fully utilize the processing power of the silicon, making this market a ripe opportunity for using virtualization to fully load those chips.

A hypervisor, or Virtual Machine Monitor, is the software layer responsible for creating and managing Virtual Machines (VMs), which are execution environments that to software running in them are, ideally, indistinguishable from the real thing. Each of these VMs can be used to run a different guest software stack, from Operating Systems (OS) to applications. A type-1 hypervisor is the lowest layer of software and runs directly on the hardware and has full control of the platform's resources. In contrasts, a type-2 hypervisor runs as an application on top of a host OS and can only control the resources the host OS allows it to.



Xen provides the ability to host multiple operating systems on the same computing platform. Xen can do this by running each guest in its own VM. The guest OS interacts with the VM just as it would with real hardware. Xen creates memory partitions so each VM has its own allocation of RAM that the other VMs cannot access, unless special permissions are given. Xen also controls when, for how long, and how many CPU cores a VM uses at any given time, preventing a guest from consuming all of the processing resources. Xen also has control over incoming interrupts, determining when and to which guests those interrupts are ultimately delivered.

As the lowest layer of software in the system, Xen is the component that bootloaders like U-Boot would boot into. After initializing Xen immediately starts up the first VM, known as *dom0*. *Dom0* is Xen's proxy and special agent in the system, it is required to be there and to start up. Through *dom0*, system developers can control VM, launching, pausing, or killing other guests. By default, *dom0* has access to all system resources. Guests other than *dom0* are typically referred to as *domU*.



Xen provides a few different ways to interact with peripheral I/O devices, to best meet the requirements of your project. See **Chapter 8 Other Guests**

**DornerWorks is constantly** working on getting new operating systems to run as guests on Xen.

In addition to Linux and bare metal guests, the XZD also provides libraries for making guests using FreeRTOS run on Xen. The latest documentation can be found at http://xzdforums.dornerworks.com/showthread.php?tid=620. A copy is also provided in the distribution at $(RELEASE_DIR)/docs/XZD FreeRTOS Guest Guide.pdf.

**Interacting with I/O Devices** for more details.

### 1.1.2. Benefits of Hypervisors and Virtualization

There are three main categories of reason for using virtualization technologies like Xen on your embedded project: to reduce cost/schedule, to enable new or improved features, and to reduce project and product risk. These are potent benefits that can be enjoyed by your project without a recurring license fee. You can even try it out with no up-front costs to see how Xen-provided virtualization can meet your needs. Best of all, DornerWorks is ready and able to provide expert advice and support if you decide you need customizations, additional features, or Xen consultation.

**Reduce Cost and Schedule**

The primary benefit of using virtualization is that it can reduce the production cost of your product, both in nonrecurring engineering cost and unit cost, while also helping to reduce schedule. The main way virtualization lets you accomplish that is by allowing you to combine and consolidate different software components while still maintaining isolation between them. This feature of virtualization enables maybe use cases.

One such use case is reducing of size, weight, power, and cost (SWaP-C) by reducing part count. Thanks to Moore's law, modern multi-core processors like the ZUS+ are processing powerhouses, often providing more computation power than needed for a single function. The ability to consolidate while maintaining isolation allows you to combine software that otherwise might have been deployed on multiple hardware systems or processors onto a single MPSoC chip. A single hardware platform is also easier to manage than a multi-platform system.

Consolidation with isolation can also be used to enforce greater decoupling of the software components. Coupling between software components leads to all kinds of problems with development, integration, maintenance, and future migrations. This is because coupling leads to dependencies, sometimes implicit or unknown, between the components such that a change or addition to one software unit often has a wide-reaching and unexpected ripple effect. Running different software functions in their own VMs leads to very strong decoupling where any dependencies between software functions are made explicit through configuration or I/O calls, making it easier to understand and eliminate unintended or unexpected interactions. Strong decoupling also allows greater freedom to develop the software functions in parallel with a higher confidence that the different pieces won't interfere with one another during integration.

As an aside, this level of decoupling is critical in applications needing security or safety certification, as it is a requirement to show certification authorities that there are no unintended interactions. By restricting and reducing the amount of intended interaction with strict design decoupling and VM isolation, you can also reduce re-certification costs by being able to show how changes and additions are bounded to the context of a particular VM.

Even outside the realm of safety and security considerations, the ability to replace a software function with a compatible one without having to worry about side effects can result in significant savings. Likewise, you can rest easy known that adding software in a new VM won't causing existing software functions to fail. It is also easier to re-use software components developed for one project on another, simply take the VM in which it runs and deploy it to run as a guest in a different system, allowing a mix'n'match approach with your existing software IP.

**Enable New and Improved Features**

The capabilities provided by Xen virtualization can also be used to enable new features and improve old ones. The isolation capability allows for enhanced security and safety, as it becomes possible to run functions in isolation, i.e. sandbox them, so that a breach or failure in one VM is limited to that VM alone. Not even security vulnerabilities in the VM's OS would result in compromise of functions in another VM, providing defense in depth.

The capability to consolidate disparate software functions enables the implementation of a centralized monitoring and management function that operates externally to the software functions being monitored. This MM function could be used to detect and dynamically respond to breaches and faults, for example, restarting faulted VMs or terminating compromised VMs before the hacker could exploit it. A centralized monitoring function could also prove useful in embedded applications which have a greater emphasis on up-time. The monitoring function could detect or predict when a VM is faulting, or about to fault, and ready a backup VM to take over with minimal loss of service.

There are other use cases that are common in the server world, where VMs are managed algorithmically by other programs, being created, copied, migrated, or destroyed in response to predefined stimulus. Virtualization enables guest migration, where the entire software stack, or part of it, could be moved from one VM to another, potentially on another platform entirely. This could be an important enabler for self-healing systems. Migration can help with live system upgrades, where the system operator could patch the OS or service critical library in a backup copy of the VM then test the patched VM to validate correct operation before migrating the actively running application to the patched VM, again with a minimal loss of service. Another use case seen in the server market is the ability to perform load balancing, either by dynamically controlling the number of VMs running to meet the current demands, or by migrating VMs to a computing resource closer to where the processing is actually needed, reducing traffic on the network.

**Reduce Program and Product Risk**

Virtualization can be used to reduce program risk by providing means to reconcile contradictory requirements. The most obvious example being the case where two pre-existing applications are needed for a product, but where each were developed to run on a different RTOS. In this case the contradictory requirements are regarding the OS to use. Other examples including different safety or security levels, where isolation allows you to avoid having to develop all of your software to the highest level, or using software functions with different license agreements.

Long lived programs can also benefit from the ability to add new VMs to the system at a later date, creating a path for future upgrades. Likewise, in a system using VMs, it becomes easier to migrate to newer hardware, especially if the hardware supports backward compatibility, like the ARMv8 does for the ARMv7. Even if it isn't, thanks to Moore's law, newer processors will have even greater processing capabilities, and emulation can be used in an VM to provide the environment necessary to run legacy software.

Virtualization can also be used to reduce risk of system failure during runtime. Previously mentioned was dynamic load balancing, which can also be considered one way to reduce the risk of failure, but virtualization can also be used to easily provide redundancy to key functionality by running a second copy of the same VM. With the centralized monitoring also previously mentioned, the redundant VM can even be kept in a standby state, and only brought to an active state if data indicates a critical function is experiencing issues or otherwise about to fail.

## 1.2. Introduction to the Xilinx Zynq UltraScale+ Multiprocessor System-on-Chip (MPSoC) Platform

The Xilinx Zynq UltraScale+ Multiprocessor System-on-Chip (MPSoC) is a fully featured processing platform. Xilinx provides an overview of the Zynq UltraScale+ MPSoC here, zynq-ultrascale-mpsoc. The Zynq UltraScale+ MPSoC has many advanced features including a quad core ARM Cortex-A53, a dual core ARM Cortex–R5, the Mali-400MP2 GPU, and a highly configurable FPGA fabric. For more information on the ARM Cortex-A53, please visit cortex-a53-processor. The Zynq UltraScale+ MPSoC is capable of running the open source hypervisor Xen. Details on the Xen hypervisor are located at this web site, xenproject. These features make the new Zynq UltraScale+ MPSoC a strong choice for embedded applications, including aerospace & defense, medical, industrial, telecom, and many other application spaces.

[1] http://www.zdnet.com/article/amazon-ec2-cloud-is-made-up-of-almost-half-a-million-linux-servers/

# *Chapter 2 Board Setup*

## 2.1. ZCU102

This chapter will contain instructions on how to setup the ZCU102 board. See the ZCU102 Evaluation Board Overview document from Xilinx for a block diagram of the board to see where all the ports are.

1. *Configure the boot mode DIP switch (SW6) for JTAG boot. This requires setting SW6 to 0000. SW6 seems to be upside down, so if you can't connect to JTAG correctly, set it to 1111 instead.*
2. *Connect the power cable.*
3. *Connect the Ethernet port to your network.*
4. *Connect the USB UART and USB JTAG to your host machine.*
5. *Power up the board.*
6. *Make sure you have an SD Card that is at least 8 GB.*

# Chapter 3 Host Setup

## 3.1. Release Image

The release image is a compressed TAR archive. The archive contains a prepackaged image that will allow the engineer to run the Xen hypervisor with a set of prepackaged domains. The figure below illustrates the contents of the release image with the included components needed to run the development system.

Once the archive is unpackaged, the directory structure will contain the following subfolders and files:

- *XZD_20161231*
  - *dist*
    - *hw-description*
    - *images*
      - *linux*
    - *subsystems*
  - *docs*
  - *dts*
  - *misc*
    - *examples*
      - *baremetal*
        - …
    - *xzd_bmc*
  - *XZD_20161231.bsp*

## 3.2. Setting up Host OS

These instructions are intended to be run on an x86_64 Ubuntu 14.04 or 16.04 host. Ensure that the host has at least 40GB free space.

### 3.2.1. Required Tools

You will need `git`, several tools in Ubuntu's build-essential package, and others to complete this build process. To install the Ubuntu tools, use the command below:

```
$ sudo apt-get install -y build-essential git mercurial dos2unix gawk tftpd-hpa flex bison
unzip screen
```

You can optionally install Vivado if you need to make modifications to the FPGA or other board devices. The installation and use of Vivado is outside the scope of this manual and it is left to the user to understand its requirements and the interactions caused by their changes.

The Xilinx XSDK is required to be installed if you plan to boot the ZCU102 from JTAG. The rest of this guide assumes the XSDK is installed to the default location at /opt/Xilinx/. Once the XSDK is correctly installed, make sure to install the Xilinx device drivers:

```
$ cd /opt/Xilinx/SDK/2016.3/data/xicom/cable_drivers/lin64/install_script/install_drivers
$ sudo ./install_drivers
```

### 3.2.2. Required Libraries

Install these additional libraries prior to PetaLinux installation.

| **Library** | **Ubuntu Package Name** |
|---|---|
| ncurses terminal library | ncurses-dev |
| 64-bit Openssl library | libssl-dev |
| 64-bit zlib compression library | zlib1g-dev |
| 32-bit zlib compression library | lib32z1 |
| 32-bit GCC support library | lib32gcc1 |
| 32-bit ncurses | lib32ncurses5 |
| 32-bit Standard C++ library | lib32stdc++6 |
| 32-bit selinux library | libselinux1:i386 |

```
$ sudo apt-get install -y ncurses-dev lib32z1 lib32gcc1 lib32ncurses5 lib32stdc++6
libselinux1:i386 zlib1g-dev libssl-dev
```

## 3.3. Installing the Image

There should be at least 10GB of free disk space available to decompress the archive and run the included scripts. The following setup was tested on an x84_64 PC running native Ubuntu 14.04 or 16.04 with 6 Gb DDR2 Memory, and a Core 2 Quad Q6600 processor.

If a developer wants to install/run the distribution:

1. *Copy the release image to an appropriate location on the host system.*
2. *Open a terminal on the host system.*

3. *The PetaLinux build environment requires that you link /bin/sh to /bin/bash.*

```
$ cd /bin
$ sudo rm sh
$ sudo ln -s bash sh
```

4. *Close, and reopen your terminal and navigate to the image's location.*
5. *Extract the image into the directory with the following command:*

```
$ tar –xvzf XZD_20161231.tgz
```

6. *Create an environment variable, $RELEASE_DIR*

```
$ export RELEASE_DIR=`pwd`/XZD_20161231
```

7. *Download petalinux-v2016.3-final-installer.run from the Xilinx Tools website to the $RELEASE_DIR directory.*
8. *Once downloaded, the PetaLinux installer must be made executable.*

```
$ cd $RELEASE_DIR
$ chmod u+x petalinux-v2016.3-final-installer.run
```

9. *Install PetaLinux in your release directory by running the following command (This will require accepting a license agreement).*

```
$ ./petalinux-v2016.3-final-installer.run
```

10. *Once PetaLinux is installed, source the PetaLinux script using the following command.*

```
$ source petalinux-v2016.3-final/settings.sh
```

11. *Create tftpboot folder and install prebuilt binaries using the following commands*

```
$ sudo mkdir -p /tftpboot
$ sudo chmod 777 /tftpboot
$ cp $RELEASE_DIR/dist/images/linux/* /tftpboot/
```

## 3.4. Setup TFTP Server

A TFTP server is needed to load images to any target boards.

1. *Configure the TFTP server by changing the value of TFTP_DIRECTORY to "/tftpboot" in /etc/default/tftpd-hpa*
2. *Restart the TFTP server*

```
$ sudo service tftpd-hpa restart
```

3. *Take note of your IP configuration. Save the values for inet addr, Bcast, and Mask. These will be used in U-Boot for serverip, gatewayip, and netmask respectively.*

```
$ ifconfig
```

# Chapter 4 Booting and Running XZD

## 4.1. Booting the Emulated System

### 4.1.1. *QEMU Introduction*

QEMU is used to emulate and virtualize hardware on a host computer. This allows a user to create and test software against the UltraScale+ MPSoC architecture. In many cases, access to actual hardware may be limited and using a virtualized system provides a significant cost savings while enabling the embedded engineer to develop applications specific for the hardware platform. QEMU is capable of emulating hardware ranging from x86 to PowerPC and, and in this case, ARM processors. One can download the QEMU source code and build for any number of supported hardware architectures.

The Xen Zynq Distribution environment consists of a version of QEMU built specifically for the Zynq UltraScale+ MPSoC processor. When the development system setup described in section 4 is started, a QEMU instance boots and is passed arguments that tell QEMU to emulate the quad core ARM Cortex-A53 hardware for the Zynq UltraScale+ MPSoC.

### 4.1.2. *Booting and Running QEMU*

If you want to boot on actual hardware, skip to the next section.

1. *Install the Device Tree blob for the emulated system*

```
$ cp $RELEASE_DIR/dist/images/linux/xen-qemu.dtb /tftpboot/xen.dtb
```

2. *Once the images have been installed correctly, they can be booted using QEMU. Boot QEMU using the following two commands (the second command is all on a single line):*

```
$ cd $RELEASE_DIR/dist
$ qemu-system-aarch64 -L $RELEASE_DIR/petalinux-v2016.3-final/etc/qemu -M arm-generic-fdt -
device loader,addr=0xfd1a0104,data=0x8000000e,data-len=4 -serial mon:stdio -serial
/dev/null -display none -device loader,file=$RELEASE_DIR/dist/images/linux/bl31.elf,cpu=0 -
device loader,file=$RELEASE_DIR/dist/images/linux/u-boot.elf -gdb tcp::9000 -tftp /tftpboot
-drive file=$RELEASE_DIR/dist/images/dom0.img,format=raw,if=sd -redir tcp:2222::22 -net
nic,vlan=0 -net user,vlan=0 -net nic,vlan=0 -net nic,vlan=0 -net nic,vlan=0 -hw-dtb
$RELEASE_DIR/dist/images/linux/zynqmp-qemu-arm.dtb -pflash
$RELEASE_DIR/dist/images/linux/nand0.qcow2
```

Running the command above will start the prepackaged images on the target machine. In this case the target is QEMU, running on the host computer. While the script is running, there will be kernel messages written to the stdout. They can be ignored.

Executing the command will boot a QEMU system which emulates the Zynq UltraScale+ MPSoC, and necessary hardware. Once the emulated system is initialized, it will load the first stage boot loader (FSBL), which bootstraps the second stage boot loader, U-Boot, in QEMU's memory (RAM). The script will then load the dom0 file system as a virtual hard drive.

At this point, QEMU will start booting the emulated Zynq UltraScale+ MPSoC. Once the FSBL has completed, U-Boot then takes over and loads the Xen kernel followed by the dom0 kernel. To accomplish this, U-Boot transfers the Xen kernel, the Xen device tree blob (DTB), and the dom0 kernel into QEMU's RAM. Once the images have been placed in memory and Xen has booted, boot up responsibilities transfer to Xen whose main priority is booting the dom0 kernel. The script passes in the dom0.img file to QEMU as a SATA device, and then QEMU uses that to emulate the hard drive.

Once all of the above has been completed you will be presented with the dom0 login prompt.

3. *Log into the system using root/root as the username and password.*

   Once these steps have been completed, Xen and a virtual machine, specifically a Linux dom0 will be running on an emulated Xilinx UltraScale+ MPSoC.

   The image in Figure 2 visually shows the QEMU setup with Xen and the two domains included in the setup that we are walking through here.

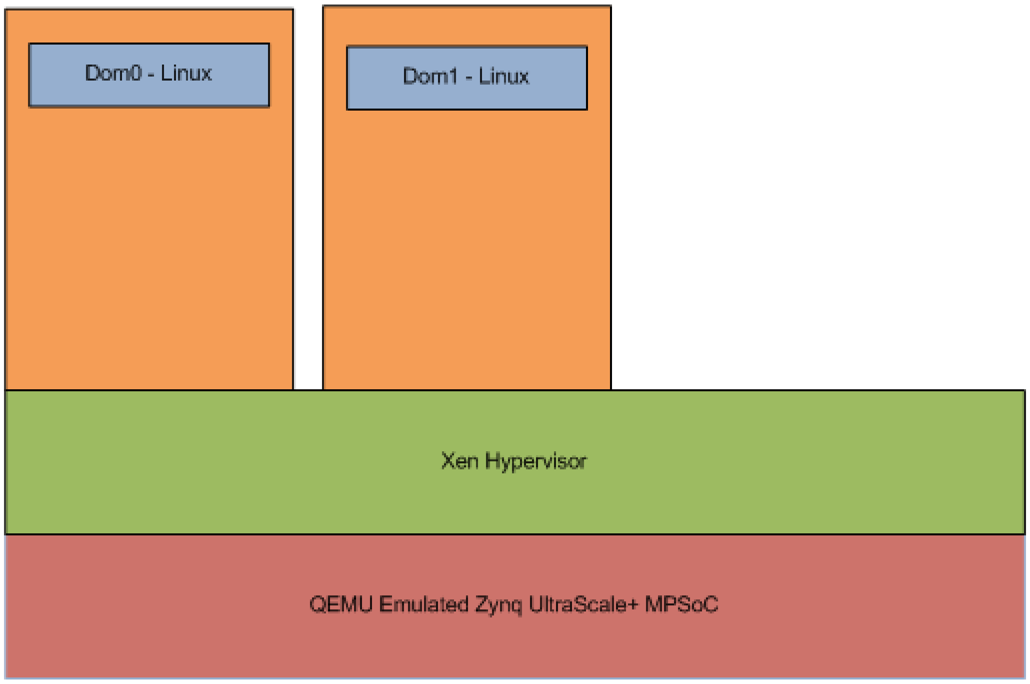


**Figure 2. System Architecture**

Now that QEMU has booted the emulated hardware, we need to make sure that Xen is running on the emulated hardware. To test that Xen is running and display diagnostic information, use the 'xl info' command. The expected output is shown on the following page.

```
[root@xilinx-dom0 ~]# xl info
host                   : xilinx-dom0
release                : 4.4.0
version                : #3 SMP Wed Jun 15 12:01:36 EDT 2016
machine                : aarch64
nr_cpus                : 4
max_cpu_id             : 127
nr_nodes               : 1
cores_per_socket       : 1
threads_per_core       : 1
cpu_mhz                : 50
hw_caps                :
00000000:00000000:00000000:00000000:00000000:00000000:00000000:00000000
virt_caps              :
total_memory           : 4096
free_memory            : 3038
sharing_freed_memory   : 0
sharing_used_memory    : 0
outstanding_claims     : 0
free_cpus              : 0
xen_major              : 4
xen_minor              : 7
xen_extra              : .0-rc
xen_version            : 4.7.0-rc
xen_caps               : xen-3.0-aarch64 xen-3.0-armv7l
xen_scheduler          : credit
xen_pagesize           : 4096
platform_params        : virt_start=0x200000
xen_changeset          : Tue Feb 2 14:24:02 2016 -0500 git:398c245
xen_commandline        : console=dtuart dtuart=serial0 dom0_mem=512M bootscrub=0
dom0_vcpus_pin maxcpus=3 timer_slop=0
cc_compiler            : aarch64-linux-gnu-gcc (crosstool-NG linaro-1.13.1-4.9-2014.09 -
cc_compile_by          : robertvanvossen
cc_compile_domain      :
cc_compile_date        : Wed Jun 15 12:00:58 EDT 2016
build_id               : ccd4c633e7094b77493d12feda5aff138fe5b677
xend_config_format     : 4
```

If Xen is not properly installed or running, you will receive an error similar to:

```
xl: command not found
```

If you receive this error, then your terminal context is most likely not in the QEMU emulated environment. Make sure QEMU's terminal is in focus. If you continue to receive error, then you have booted a QEMU emulated system without Xen somehow. Please re-extract the release image, and try again.

To view the list of running domains, use the 'xl list' command

```
# xl list

Output:
Name                                        ID   Mem VCPUs      State   Time(s)
dom0                                         0   512     1     r-----     133.4
```

You are now free to test and experiment on the QEMU emulated system!

## 4.2. Booting the ZCU102

### 4.2.1. Booting via JTAG

If you already booted in QEMU, you can skip this section.

1. *Populate your SD Card. This only needs to be done when a change is made to the file system.*
   a. *Insert your SD Card into your host machine.*
   b. *Figure out which device the SD Card shows up as. It should be the last device that shows up.*

```
$ dmesg
```

   c. *Copy the Dom0 file system to SD Card. (Our device is /dev/sdb)*

```
$ sudo dd if=$RELEASE_DIR/dist/images/dom0.img of=/dev/sdb bs=1M
```

   d. *Unmount SD Card and place it back in the ZCU102.*

2. *Install the Device tree blob for the ZCU102*

```
$ cp $RELEASE_DIR/dist/images/linux/xen-zcu102.dtb /tftpboot/xen.dtb
```

3. *In a new terminal, connect to the ZCU102 UART, assuming the device is mounted to /dev/ttyUSB2*

```
$ sudo screen /dev/ttyUSB2 115200
```

4. *Restart the ZCU102 using the power switch*
5. *In the other terminal, connect to the jtag, load boot images, and run them*

```
$ cd $RELEASE_DIR/dist
$ sudo /opt/Xilinx/SDK/2016.3/bin/xsdb dist_zcu102_boot.tcl
```

6. *In the screen terminal, stop the U-Boot autoboot and set the following environment variables from the network values from earlier (Use an open IP address on your network for ipaddr):*

```
U-Boot-PetaLinux> setenv serverip  xxx.xx.xxx.xxx
U-Boot-PetaLinux> setenv gatewayip xxx.xx.xxx.xxx
U-Boot-PetaLinux> setenv netmask   xxx.xx.xxx.xxx
U-Boot-PetaLinux> setenv ipaddr    xxx.xx.xxx.xxx
U-Boot-PetaLinux> run xen
```

7. *Log into the system using root/root as the username and password.*

### 4.2.2. Booting via SD card

#### 4.2.2.1. Setting up dual partition SD card

As an alternative to booting the ZCU102 using U-Boot and FSBL from the host PC, the board can be booted with all the boot systems on a second partition of the SD card. Section 4.2.2 is an alternative to 4.2.1.

The following commands will create a 256M fat partition on a new SD card image for the boot files and a second ~7G ext4 partition for the dom0 root file system.

1. Create a new clean two partition image.

```
$ dd if=/dev/zero of=$RELEASE_DIR/dist/images/sdcard.img bs=1G count=7
$ echo -e "n\np\n1\n\n+256M\nt\nc\nn\np\n2\n\n\nt\n2\n83\nw\n" | fdisk
$RELEASE_DIR/dist/images/sdcard.img
```

2. Install kpartx if it is not already.

```
$ sudo apt-get install kpartx
```

3. Mount both partitions of the image using kpartx.

```
sudo kpartx -av $RELEASE_DIR/dist/images/sdcard.img
sudo mkfs.vfat /dev/mapper/loop1p1
sudo mkfs.ext4 /dev/mapper/loop1p2
sudo fatlabel /dev/mapper/loop1p1 BOOT
sudo e2label /dev/mapper/loop1p2 Dom0FS
mkdir $RELEASE_DIR/tmpboot
mkdir $RELEASE_DIR/tmpext4
sudo mount /dev/mapper/loop1p1 $RELEASE_DIR/tmpboot/
sudo mount /dev/mapper/loop1p2 $RELEASE_DIR/tmpext4/
```

4. Mount the dom0 root file system image using kpartx to copy the files onto the root file system partition.

```
sudo kpartx -av $RELEASE_DIR/dist/images/dom0.img
mkdir $RELEASE_DIR/tmpxilfs
sudo mount /dev/mapper/loop2p1 $RELEASE_DIR/tmpxilfs/
sudo cp -r $RELEASE_DIR/tmpxilfs/* $RELEASE_DIR/tmpext4/
```

5. Clean up the root file system workspaces.

```
sudo umount $RELEASE_DIR/tmpxilfs/
sudo kpartx -d $RELEASE_DIR/dist/images/dom0.img
sudo umount $RELEASE_DIR/tmpext4/
rmdir $RELEASE_DIR/tmpxilfs/ $RELEASE_DIR/tmpext4/
```

6. Make DTS bootargs modification to tell dom0 to use mmcblk0p2 as the rootfs.

   a. Create a copy of the DTS file.

```
cp dts/xen-zcu102.dts dts/xen-zcu102_sd.dts
```

   b. Make the following modification to dts/xen-zcu102_sd.dts:

```
diff -u xen-zcu102.dts xen-zcu102_sd.dts
--- xen-zcu102.dts
+++ xen-zcu102_sd.dts
@@ -1603,7 +1603,7 @@
             #address-cells = <0x2>;
             #size-cells = <0x1>;
             xen,xen-bootargs = "console=dtuart dtuart=serial0 dom0_mem=512M bootscrub=0
dom0_vcpus_pin dom0_max_vcpus=1 maxcpus=4 timer_slop=0";
-            xen,dom0-bootargs = "console=hvc0 earlycon=xen earlyprintk=xen rootdelay=1
root=/dev/mmcblk0p1 devtmpfs.mount=1 maxcpus=1";
+            xen,dom0-bootargs = "console=hvc0 earlycon=xen earlyprintk=xen rootdelay=1
root=/dev/mmcblk0p2 devtmpfs.mount=1 maxcpus=1";

             dom0 {
                    compatible = "xen,linux-zimage", "xen,multiboot-module";
```

c. Re-generate the dtb under a new name.

```
$ dtc -I dts -O dtb -o $RELEASE_DIR/dts/xen_sd.dtb $RELEASE_DIR/dts/xen-zcu102_sd.dts
```

d. Save the new dtb to the FAT partition of the SD card.

```
$ sudo cp $RELEASE_DIR/dts/xen_sd.dtb $RELEASE_DIR/tmpboot/xen.dtb
```

7. Optional: Generate your own boot files. Skip this step and go to the next if you are going to use the pre-made boot files.

a. Create a .bif file with the following contents:

```
$ > $RELEASE_DIR/zcu102_sd_boot.bif
the_ROM_image:
{
      [fsbl_config] a53_x64
      [bootloader] dist/images/linux/zynqmp_fsbl.elf
      [destination_cpu=a53-0] dist/images/linux/bl31.elf
      [destination_cpu=a53-0] dist/images/linux/u-boot.elf
}
```

b. Generate the boot.bin file using the bootgen utility:

```
$ /opt/Xilinx/SDK/2016.3/2016.3bin/bootgen -image zcu102_sd_boot.bif -arch zynqmp -w -o i
$RELEASE_DIR/dist/images/linux/boot.bin
```

8. Copy the 'boot.bin' files, and the 'xen.ub' and 'Image' files to the fat partition of the SD card. Note that the 'xen.dtb' file was previously copied over in step 6.

```
sudo cp $RELEASE_DIR/dist/images/linux/boot.bin $RELEASE_DIR/tmpboot/
sudo cp $RELEASE_DIR/dist/images/linux/xen.ub $RELEASE_DIR/tmpboot/
sudo cp $RELEASE_DIR/dist/images/linux/Image $RELEASE_DIR/tmpboot/
```

9. Unmount and clean up.

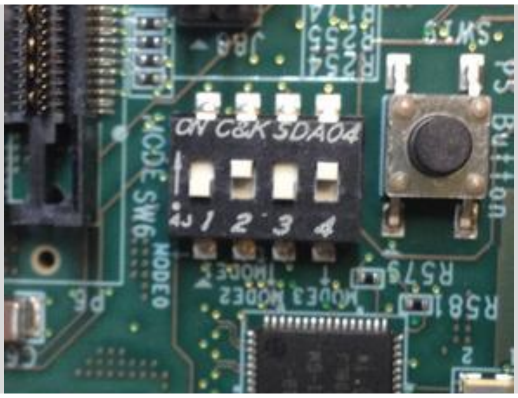
```
sudo umount $RELEASE_DIR/tmpboot/
sudo kpartx -d $RELEASE_DIR/dist/images/sdcard.img
rmdir $RELEASE_DIR/tmpboot/
```

10. Write the SD card image to an appropriate SD card, ensuring that you have the correct /dev/sdX for your SD card. Be sure to umount the SD card from the file system before running dd.

```
dmesg | tail
sudo dd if=$RELEASE_DIR/dist/images/sdcard.img of=/dev/sd? bs=4M
sudo sync
```

#### 4.2.2.2. Booting the board

1. Set the DIP switch SW6 boot mode pins to 0b0101.



2. Insert the SD card, connect to UART, and power on the board.

   The ZCU102 board has a Cygnal/Silabs CP2108 Quad USB to UART adapter, and Linux creates four `/dev/ttyUSB*` devices in an unpredictable order. Here's a bash script that will automatically grab the correct device:

```
#!/bin/bash
# Looks up the Cygnal/Silabs usb-to-uart converter by VID/PID, grabs the first of the four
ports (:1.0) and echoes the device name
echo /dev/$(echo $(grep '10c4/ea71' /sys/bus/usb-serial/devices/ttyUSB*/../uevent | tail -n
1 | sed 's/uevent\:.*$//g')../*\:1.0/ttyUSB* | gawk -F "/" '{print $NF}')
```

   Once the correct device is known, the following command will open a serial port connection in the terminal:

```
sudo screen /dev/ttyUSB# 115200
```

   Substituting `/dev/ttyUSB#` with the correct device found using the script above. For example:

```
sudo screen /dev/ttyUSB0 115200
```

3. If the U-Boot boot command was not modified, you will need to interrupt the startup sequence and enter the boot commands manually:

```
U-Boot-PetaLinux> fatload mmc 0:1 $fdt_addr xen.dtb
U-Boot-PetaLinux> fdt addr $fdt_addr
U-Boot-PetaLinux> fdt resize
U-Boot-PetaLinux> fatload mmc 0:1 0x80000 Image
U-Boot-PetaLinux> fdt set /chosen/dom0 reg <0 0x80000 0x$filesize>
U-Boot-PetaLinux> fatload mmc 0:1 6000000 xen.ub
U-Boot-PetaLinux> bootm 6000000 - $fdt_addr
```

## 4.3. Running XZD

The following sections explain how to do some very basic domain management on Xen. This works whether you are running on QEMU or on actual hardware.

### 4.3.1. Booting a Guest

The necessary components for your first guest have already been created.  They are stored in dom0's file system:

```
Component                      Location in dom0's file system
Guest Linux Kernel             /root/Dom1-Kernel
Guest File System Image        /root/Dom1.img
Guest Domain Configuration     /etc/xen/dom1.cfg
```

Staying in the terminal, we will prepare to run a guest domain. The additional domain is already included in the archive we have been working with and will now prepare to run it on the Xen hypervisor. To do that type the following commands in dom0's console:

1. *Mount the guest's file system to a loop device in domain 0.*

```
# losetup /dev/loop0 /root/Dom1.img
```

   You should receive no output if the command succeeds.

2. *Boot another domain (dom1), and connect to its console*

```
# xl create -c /etc/xen/dom1.cfg
```

   The -c flag will automatically attach dom1's console. That is, once this command is executed, you will be logging into dom1. In order to return to dom0's console while leaving the guest (dom1) running, you can press CTRL-] to get back to dom0.

3. *Login using 'root' and 'root' for the username and password.*

   Since your guest is not a privileged domain, typing 'xl info' will output less detailed information, and 'xl list' will generate an error as it can only be run in dom0.

```
[root@xilinx-dom1 ~]# xl info
host                    : xilinx-dom1
release                 : 4.4.0
version                 : #3 SMP Wed Jun 15 12:01:36 EDT 2016
machine                 : aarch64
libxl: error: libxl.c:5183:libxl_get_physinfo: getting physinfo: Operation not permitted
libxl_physinfo failed.
libxl: error: libxl.c:5752:libxl_get_scheduler: getting current scheduler id: Operation not
permitted
get_scheduler sysctl failed.
xend_config_format      : 4
```

```
[root@xilinx-dom1 ~]# xl list

Output:
```

```
libxl: error: libxl.c:670:libxl_list_domain: getting domain info list: Operation not
permitted
libxl_list_domain failed.
```

The system is now running Xen and two domains or virtual machines: dom0 and dom1.

If you want to return to dom0's console while leaving the guest running, you may press CTRL-]. This will close the internal console connection, and bring dom0 back into focus within QEMU's terminal.

To reconnect to a guest terminal, use the "xl console" command

```
[root@xilinx-dom0 ~]# xl console dom1
```

It is important that you are aware which guest you are issuing commands to. Pay careful attention to the hostname listed in the terminal:

```
[root@xilinx-dom0 ~]#        dom0
[root@xilinx-dom1 ~]#        dom1
```

### 4.3.2. Copying a Guest

Both dom0 (control) and dom1 (guest) are included in the archive. You can easily boot a second guest domain, for a total of three domains including dom0, by making copies of dom1's components.

1. *Make sure that domain 1 is powered down before we copy its kernel and file system.*

```
[root@xilinx-dom0 ~]# xl console dom1
[root@xilinx-dom1 ~]# poweroff
```

The guest will shutdown, and the system should return to domain0's console automatically.

Make sure that the guest is completely shutdown by using the 'xl list' command. Dom1 should **NOT** have an entry. If you do see an entry for dom1, then this means that dom1 is still shutting down. Wait for approximately 15 or 20 seconds and try the command again. The results should appear similar to the below output.

```
[root@xilinx-dom0 ~]# xl list

Output:
Name                                        ID   Mem VCPUs State  Time(s)
dom0                                         0   512     1     r-----     259.7
```

2. *Copy the dom1 FS image.*

```
[root@xilinx-dom0 ~]# cp /root/Dom1.img /root/Dom2.img
```

This file is 1Gb, and will take a while to copy on the emulated SATA device.

3. *Copy the dom1 kernel.*

```
[root@xilinx-dom0 ~]# cp /root/Dom1-Kernel /root/Dom2-Kernel
```

4. *Copy the dom1 configuration file.*

```
[root@xilinx-dom0 ~]# cp /etc/xen/dom1.cfg /etc/xen/dom2.cfg
```

5. *Edit the new dom2 configuration file.*

You will need to:
a. Rename the guest to be named dom2.

b. Configure the guest to boot domain 2's kernel.
c. Change the targeted loop device to allow two domains to run simultaneously.

You can accomplish this change using vi or sed expressions.

```
[root@xilinx-dom0 ~]# vi /etc/xen/dom2.cfg
```

or

```
[root@xilinx-dom0 ~]# sed -i 's/om1/om2/' /etc/xen/dom2.cfg
[root@xilinx-dom0 ~]# sed -i 's/loop0/loop1/' /etc/xen/dom2.cfg
```

6. Verify that the changes have been made to the appropriate files.

```
[root@xilinx-dom0 ~]# cat /etc/xen/dom2.cfg

# =====================================================================
# Example PV Linux guest configuration
# =====================================================================

...

# Guest name
name = "dom2"

...

# Kernel image to boot
kernel = "/root/Dom2-Kernel"

...

# Disk Devices
# A list of `diskspec' entries as described in
# docs/misc/xl-disk-configuration.txt
disk = [ 'phy:/dev/loop1,xvda,w' ]
```

7. Mount the guest file systems to their respective loop devices in domain 0. The losetup command creates a device on which we can mount the file systems created.

```
# If you have already mounted /root/Dom1.img to /dev/loop0,
# there is no need to mount it again
[root@xilinx-dom0 ~]# losetup /dev/loop0 /root/Dom1.img
# The dom2 file system hasn't been mounted yet:
[root@xilinx-dom0 ~]# losetup /dev/loop1 /root/Dom2.img
```

8. *Start the domains (or virtual machines) with the following commands.*

```
[root@xilinx-dom0 ~]# xl create /etc/xen/dom1.cfg
```

When you leave the -c flag off of the domain creation command, you will receive the guests initial boot messages to dom0's standard out, while also keeping dom0's console in focus.

This chatter does not affect your standard input however it does make it a bit hard to type the next command. You might want to wait before issuing the next command. The last message should look similar to the below output (the '3' in the vif3.0 output is variable).

```
...
xenbr0: port 2(vif3.0) entered forwarding state
xenbr0: port 2(vif3.0) entered forwarding state
xenbr0: port 2(vif3.0) entered forwarding state
```

Hit enter to bring up a new line in the console, indicating your hostname. Make sure you are still in dom0's console, and that you didn't attach to the guest.

```
[root@xilinx-dom0 ~]#
[root@xilinx-dom0 ~]# xl create /etc/xen/dom2.cfg
```

You should see similar console chatter from dom2 booting as it reports to standard out.

The 'xl list' command will now show all three domains running. The ID values are sequential and will increase each time a domain is created. The ID numbers here might look different in your output.

```
[root@xilinx-dom0 ~]# xl list

Name                                        ID   Mem VCPUs      State   Time(s)
dom0                                         0   512     1     r-----     378.0
dom1                                         1   128     1     -b----      67.8
dom2                                         2   128     1     -b----      46.5
```

Guest domains should be shutdown carefully, as their file systems are easily corrupted if they are reset improperly or shutdown in the middle of an IO operation. There are two methods to shut down a guest:

From domain0, you can request the Xen Kernel to send a shutdown signal to a guest:

```
[root@xilinx-dom0 ~]# xl shutdown dom1
```

You could also attach to dom1's console by executing the command 'xl console dom1' and send the poweroff command:

```
# To attach to dom1's console
[root@xilinx-dom0 ~]# xl console dom1
# While in dom1
[root@xilinx-dom1 ~]# poweroff
```

The poweroff command will function as expected within dom0's console as well, but you should make sure that all domains are properly shutdown before doing so.

```
[root@xilinx-dom0 ~]# poweroff
```

Once all the domains have been shut down, you may terminate the instance of QEMU using the keyboard shortcut CTRL-A X. (Control-A, release, then X)

View other QEMU shortcuts by typing CTRL-A H (Control-A, release, then H, while QEMU is running).

### 4.3.3. Booting Guests with Alternate File Systems

Two alternate guest file systems and matching Xen configuration files have also been provided, these are the Ubuntu Core file system and the Linaro flavored OpenEmbedded file system. Their images can be mounted and the guests booted using commands similar to those found in 4.3.1:

For the Ubuntu Core FS:

1. *Mount the guest's file system to a loop device in domain 0.*

```
# losetup /dev/loop1 /root/ubuntu-core-fs.img
```

2. *Boot another domain, and connect to its console*

```
# xl create -c /etc/xen/ubuntu-core-fs.cfg
```

For the Linaro OpenEmbedded FS:

1. *Mount the guest's file system to a loop device in domain 0.*

```
# losetup /dev/loop2 /root/linaro-openembedded-fs.img
```

2. *Boot another domain, and connect to its console*

```
# xl create -c /etc/xen/linaro-openembedded-fs.cfg
```

You can perform all of the same operations with these guests as described in 4.3.1 and 4.3.2.

# Chapter 5 Building from Source

## 5.1. Environment Setup and Build Process

If you have not executed the steps in section 3.3, in a terminal, set RELEASE_DIR to the directory path where the archive was decompressed. This variable will be used in several instructions so that you may copy-paste them.

```
$ export RELEASE_DIR=`pwd`/XZD_20161231
```

## 5.2. Build Dom0 Linux Kernel, Xen, U-Boot, & FSBL

The following instructions assume that you are using the provided PetaLinux and QEMU binaries.

***TIP:*** *If you have not yet setup your host, please follow the steps in Chapter 3.*

1. Once PetaLinux is installed, source the PetaLinux script using the following command.

```
$ cd $RELEASE_DIR
$ source petalinux-v2016.3-final/settings.sh
```

2. Create a project directory named 'XenZynqDist' using the BSP provided:

```
$ petalinux-create -t project -s XZD_20161231.bsp -n XenZynqDist
```

   This project directory should have everything you need to build the Xen Zynq distribution.

3. Enter the project directory

```
$ cd $RELEASE_DIR/XenZynqDist
```

4. The default configuration will be sufficient to run as the dom0 kernel for this exercise.

```
$ petalinux-config -- oldconfig
```

   The command below can be used to select and additional features that you might want to add to the dom0 kernel configuration. When finished select exit to write the configuration file.

   Added kernel configurations are beyond the scope of this document and are the responsibility of the user to understand their interactions and consequences if used.

```
$ petalinux-config -c kernel
```

5. Use PetaLinux to build the linux kernel, Xen, U-Boot, and FSBL images.

```
$ petalinux-build
```

***TIP:** If you want more information during the build, you can use the verbose flag '–v' with the petalinux-build command.*

6. View your images in the 'images/linux/' directory

```
$ ls images/linux/
bl31.bin  Image      image.ub    System.map.linux  u-boot.elf    u-boot-s.elf  u-boot-
s.srec  xen.ub
bl31.elf  image.elf  system.dtb  u-boot.bin        u-boot-s.bin  u-boot.srec   vmlinux
zynqmp_fsbl.elf
```

## 5.3. Build the Dom0 File System

These instructions build the file system (FS) for the system domain, Dom0.

1. Clone the buildroot source and create the default configuration file for the Dom0 FS.

```
$ cd $RELEASE_DIR
$ git clone https://github.com/dornerworks/buildroot.git -b xen-guest-fs
$ cd buildroot
$ make zynq_ultra_mpsoc_dom0_defconfig
```

2. Run the menuconfig tool if there is specific functionality that needs to be built into the FS.

```
$ make menuconfig
```

3. Once your configuration is complete, build the FS.

```
$ make
```

The make may run for up to one hour depending on the host system specifications.

The make command generates a FS tarball called rootfs.tar in the following location:

```
$ ls $RELEASE_DIR/buildroot/output/images
rootfs.cpio  rootfs.cpio.gz  rootfs.tar
```

4. Copy package configuration within buildroot:

```
$ cp $RELEASE_DIR/buildroot/output/host/usr/bin/pkg-config
$RELEASE_DIR/buildroot/output/host/usr/bin/aarch64-buildroot-linux-gnu-pkg-config
```

5. Add buildroot compiler to your path:

```
$ export PATH=$PATH:$RELEASE_DIR/buildroot/output/host/usr/bin/
```

6. Configure and build the Xen tools:

```
$ cd $RELEASE_DIR/XenZynqDist/components/apps/xen/xen-src
$ ./configure --host=aarch64-buildroot-linux-gnu --enable-tools
$ make dist-tools CROSS_COMPILE=aarch64-buildroot-linux-gnu- XEN_TARGET_ARCH=arm64
CONFIG_EARLY_PRINTK=ronaldo
```

7. Add the Xen tools to buildroot:

```
$ cd $RELEASE_DIR
$ cp $RELEASE_DIR/buildroot/output/images/rootfs.tar dom0.rootfs.tar
$ fakeroot tar -rvf dom0.rootfs.tar --transform='s,usr/lib64,usr/lib,S' --
transform='s,var/log,tmp,S' --show-transformed-names -
C$RELEASE_DIR/XenZynqDist/components/apps/xen/xen-src/dist/install ./etc/ -
C$RELEASE_DIR/XenZynqDist/components/apps/xen/xen-src/dist/install ./usr/ -
C$RELEASE_DIR/XenZynqDist/components/apps/xen/xen-src/dist/install ./var/
```

8. Create and partition the dom0 image file:

```
$ dd if=/dev/zero of=$RELEASE_DIR/XenZynqDist/images/dom0.img bs=1G count=7
$ echo -e "n\np\n1\n\n\nt\n83\nw\n" | fdisk $RELEASE_DIR/XenZynqDist/images/dom0.img
```

9. Mount and format the Dom0 image file:

```
$ sudo losetup -o 1048576 /dev/loop0 $RELEASE_DIR/XenZynqDist/images/dom0.img
$ sudo mkfs.ext2 /dev/loop0
$ mkdir -p $RELEASE_DIR/tmp/dom0_fs
$ sudo mount /dev/loop0 $RELEASE_DIR/tmp/dom0_fs
```

10. Install the Dom0 FS onto the image file:

```
$ sudo tar -xvf dom0.rootfs.tar -C $RELEASE_DIR/tmp/dom0_fs
```

11. Unmount the Dom0 image file:

```
$ sudo umount /dev/loop0
$ sudo losetup -d /dev/loop0
```

## 5.4. Build Device Tree Blob

1. Create the device tree blob for your target by executing one of the following commands:
   a. For the emulated system, create the xen-qemu device tree blob (DTB)

```
$ dtc -I dts -O dtb -o /tftpboot/xen.dtb $RELEASE_DIR/dts/xen-qemu.dts
```

   b. Or, for the zcu102, create the xen-zcu102 DTB

```
$ dtc -I dts -O dtb -o /tftpboot/xen.dtb $RELEASE_DIR/dts/xen-zcu102.dts
```

2. Create the DTB for QEMU to use

```
$ dtc -I dts -O dtb -o $RELEASE_DIR/XenZynqDist/images/linux/zynqmp-qemu-arm.dtb
$RELEASE_DIR/XenZynqDist/subsystems/linux/configs/device-tree/zynqmp-qemu-arm.dts
```

## 5.5. Building and Installing the Guest

### *5.5.1. Build the Guest Domain Kernel*

If the guest kernel does not need to be configured differently from the dom0 kernel, then one can skip to step 5 in this section.

1. Change directory to the Xen Zynq Distribution

```
$ cd $RELEASE_DIR/XenZynqDist
```

2. Configure the guest kernel

```
$ petalinux-config -c kernel
```

3. Build a new kernel

```
$ petalinux-build -c kernel
```

4. Assemble the kernel image

```
# Assemble the image:
$ petalinux-build -x package
```

5. Copy the guest linux kernel image

```
$ cp $RELEASE_DIR/XenZynqDist/images/linux/Image $RELEASE_DIR/Dom1-Kernel
```

### *5.5.2. Build the Guest Domain File System Using BuildRoot*

1. Change to the $RELEASE_DIR/buildroot directory. Create the default configuration file for a guest FS.

```
$ cd $RELEASE_DIR/buildroot
$ make zynq_ultra_mpsoc_guest_defconfig
```

2. Run the menuconfig tool if there is specific functionality that needs to be built into the FS.

```
$ make menuconfig
```

3. Once your configuration is complete, build the FS.

```
$ make
```

   The make command generates a FS tarball called rootfs.tar in the following location:

```
$ ls $RELEASE_DIR/buildroot/output/images
rootfs.cpio  rootfs.cpio.gz  rootfs.tar
```

4. If the Xen tools are not built, then follow the steps to build the Xen-Tools in the step *Configure and build the Xen tools* to create the tools. Following this, then execute the command below.

```
$ cd $RELEASE_DIR
$ cp $RELEASE_DIR/buildroot/output/images/rootfs.tar dom1.rootfs.tar
$ fakeroot tar -rvf dom1.rootfs.tar --transform='s,usr/lib64,usr/lib,S' --
transform='s,var/log,tmp,S' --show-transformed-names -
C$RELEASE_DIR/XenZynqDist/components/apps/xen/xen-src/dist/install ./etc/ -
C$RELEASE_DIR/XenZynqDist/components/apps/xen/xen-src/dist/install ./usr/ -
C$RELEASE_DIR/XenZynqDist/components/apps/xen/xen-src/dist/install ./var/
```

### *5.5.3. Install the BuildRoot Guest Image into the Dom0 File System*

To construct the configuration files for a new domain and then insert those files into the dom0 File system, follow the steps  below:

1. Attach dom0's FS to a loop device, and mount it to a temporary directory.

```
$ cd $RELEASE_DIR
$ sudo losetup -o 1048576 /dev/loop0 $RELEASE_DIR/XenZynqDist/images/dom0.img
$ mkdir -p $RELEASE_DIR/tmp/dom0_fs
$ sudo mount /dev/loop0 $RELEASE_DIR/tmp/dom0_fs
```

2. Move the Dom1-Kernel into dom0's file system.

```
$ sudo mv $RELEASE_DIR/Dom1-Kernel $RELEASE_DIR/tmp/dom0_fs/root/Dom1-Kernel
```

3. Create a configuration file for your DomU. Below is the configuration file for Dom1. It contains additional options that will not be used in the rest of the demonstration.

   Move to the following directory and insert the following file in your desired manner:

```
$ cd $RELEASE_DIR/tmp/dom0_fs/etc/xen/
```

```
# =====================================================================
# Example PV Linux guest configuration
# =====================================================================
#
# This is a fairly minimal example of what is required for a
# Paravirtualised Linux guest. For a more complete guide see xl.cfg(5)

# Guest name
name = "dom1"

# 128-bit UUID for the domain as a hexadecimal number.
# Use "uuidgen" to generate one if required.
# The default behavior is to generate a new UUID each time the guest is started.
#uuid = "XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX"

# Kernel image to boot
kernel = "/root/Dom1-Kernel"

# Kernel command line options
extra = "console=hvc0 earlyprintk=xenboot root=/dev/xvda1 rw"

# Initial memory allocation (MB)
memory = 128

# Maximum memory (MB)
# If this is greater than `memory' then the slack will start ballooned
# (this assumes guest kernel support for ballooning)
#maxmem = 512

# Number of VCPUS
vcpus = 1

# Network devices
# A list of 'vifspec' entries as described in
# docs/misc/xl-network-configuration.markdown
vif = [ 'bridge=xenbr0' ]

# Disk Devices
# A list of `diskspec' entries as described in
# docs/misc/xl-disk-configuration.txt
disk = [ 'phy:/dev/loop0,xvda,w' ]
```

***TIP: Each** domain requires a configuration file that Xen can use to boot the domain. The configuration files are generally located in /etc/xen, however they can be located anywhere one chooses.*

4. Create a blank 1Gb file to be used as the Guest's (Dom1's) FS Image

   Take note of the Output File name (of=<Name>) you are choosing in the following command:

```
$ sudo dd if=/dev/zero of=$RELEASE_DIR/tmp/dom0_fs/root/Dom1.img bs=1M count=1024
$ cd $RELEASE_DIR
```

5. Create a partition table on the image, and use the entire image file as a Linux partition

```
$ sudo sh -c 'echo -e "n\np\n1\n\n\nt\n83\nw\n" | fdisk tmp/dom0_fs/root/Dom1.img'
```

6. Make a temporary mounting directory for the Guest FS image.

```
$ mkdir -p $RELEASE_DIR/tmp/dom1_fs
```

7. Attach the Guest FS image to a loop device.

```
$ sudo losetup -o 1048576 /dev/loop1 $RELEASE_DIR/tmp/dom0_fs/root/Dom1.img
```

8. Format the Guest FS image using 'mkfs.ext2'

```
$ sudo mkfs.ext2 /dev/loop1
```

9. Mount the Guest FS image to the temporary mounting directory.

```
$ sudo mount /dev/loop1 $RELEASE_DIR/tmp/dom1_fs
```

10. Un-pack the files for the FS.

```
$ sudo tar -xvf dom1.rootfs.tar -C $RELEASE_DIR/tmp/dom1_fs
```

11. Unmount and detach the Guest and dom0 file systems.

```
$ sudo umount /dev/loop1
$ sudo losetup -d /dev/loop1

$ sudo umount /dev/loop0
$ sudo losetup -d /dev/loop0
```

***CAUTION!** Due to how PetaLinux currently works, anytime a new guest domain is built that is configured differently from the dom0, a new dom0 must be built as the original one will be over-written*

### 5.5.4. Alternate 1: Use Ubuntu Core FS

The following are instructions for using the Ubuntu Core file system as an *alternative* to the steps described in 5.5.2 and 5.5.3 for the BuildRoot file system. If you have completed 5.5.2 and 5.5.3 or 5.5.5 please continue to 5.6 to finish the process of booting the guest.

1. Obtain the file system contents from the following link:

2. http://cdimage.ubuntu.com/ubuntu-base/releases/14.04/release/ubuntu-base-14.04-core-arm64.tar.gz
3. Attach dom0's FS to a loop device, and mount it to a temporary directory.

```
$ cd $RELEASE_DIR
$ sudo losetup -o 1048576 /dev/loop0 $RELEASE_DIR/dist/images/dom0.img
$ mkdir -p $RELEASE_DIR/tmp/dom0_fs
$ sudo mount /dev/loop0 $RELEASE_DIR/tmp/dom0_fs
```

4. Use the following commands to create a file system image for holding the Ubuntu Core contents: (For more information see Section 5.5.3 steps 4-10)

```
$ sudo dd if=/dev/zero of=$RELEASE_DIR/tmp/dom0_fs/root/ubuntu-core-fs.img bs=1M count=1024
$ cd $RELEASE_DIR
$ sudo sh -c 'echo -e "n\np\n1\n\n\nt\n83\nw\n" | fdisk tmp/dom0_fs/root/ubuntu-core-
fs.img'
$ mkdir -p $RELEASE_DIR/tmp/ubuntu-core-fs
$ sudo losetup /dev/loop1 $RELEASE_DIR/tmp/dom0_fs/root/ubuntu-core-fs.img
$ sudo mkfs.ext4 /dev/loop1
$ sudo mount /dev/loop1 $RELEASE_DIR/tmp/ubuntu-core-fs
```

5. Untar the Ubuntu Core FS files into the into the newly created image file in the dom0 FS

```
$ sudo tar -xvpzf $DOWNLOAD_LOCATION/ubuntu-base-14.04-core-arm64.tar.gz -C $RELEASE_DIR/tmp
/ubuntu-core-fs
```

6. Configure the Ubuntu Core image to work with Xen

   First, using a text editor, add the file `$RELEASE_DIR/tmp/ubuntu-core-fs/etc/init/hvc0.conf` with the following contents to enable login:

```
# hvc0 - getty
#
# This service maintains a getty on tty1 from the point the system is
# started until it is shut down again.

start on stopped rc RUNLEVEL=[2345] and (
            not-container or
            container CONTAINER=lxc or
            container CONTAINER=lxc-libvirt)

stop on runlevel [!2345]

respawn
exec /sbin/getty -8 38400 hvc0
```

   Second, edit the `root` entry in the `$RELEASE_DIR/tmp/ubuntu-core-fs/etc/passwd` file to look as follows. This will set the root user password to empty and allow you to login in as the root user:

```
root::0:0:root:/root:/bin/bash
```

7. Add a configuration file for your domU in the `$RELEASE_DIR/tmp/dom0_fs/etc/xen` folder as detailed in 5.5.3 step 3, name the file `ubuntu-core.cfg` and make the following edits:

   - Set the domU to have a different name:

```
    name = “ubuntu-core”
```

   - Set the extra line to read:

```
    extra = "console=hvc0 earlyprintk=xenboot root=/dev/xvda rw"
```

- Set the disk line to read:

```
    disk = [ 'phy:/dev/loop1,xvda,w' ]
```

8. Unmount and detach the Ubuntu Core and dom0 file systems.

```
$ sudo umount /dev/loop1
$ sudo losetup -d /dev/loop1

$ sudo umount /dev/loop0
$ sudo losetup -d /dev/loop0
```

**Usage Note:** On first boot of the Ubuntu Core FS guest, log in with user name "root" with no password. Once logged in, set the root password using the passwd command, it should look something like this:

```
Ubuntu 14.04.1 LTS localhost.localdomain hvc0

localhost login: root
Welcome to Ubuntu 14.04.1 LTS (GNU/Linux 3.18.0 aarch64)

 * Documentation:  https://help.ubuntu.com/

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

root@localhost:~# passwd
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
```

### 5.5.5. Alternate 2: Use Linaro OpenEmbedded FS

The following are instructions for using the Linaro OpenEmbedded file system as an *alternative* to the steps described in 5.5.2 and 5.5.3 for the BuildRoot file system. If you have completed 5.5.2 and 5.5.3 or 0 please continue to 5.6 to finish the process of booting the guest.

1. Obtain the image from the following link:

   https://releases.linaro.org/15.06/openembedded/juno-lsk/lt-vexpress64-openembedded_minimal-armv8-gcc-4.9_20150620-722.img.gz

2. Attach dom0's FS to a loop device, and mount it to a temporary directory.

```
$ cd $RELEASE_DIR
$ sudo losetup -o 1048576 /dev/loop0 $RELEASE_DIR/XenZynqDist/images/dom0.img
$ mkdir -p $RELEASE_DIR/tmp/dom0_fs
$ sudo mount /dev/loop0 $RELEASE_DIR/tmp/dom0_fs
```

3. Unzip the new FS image to the dom0 FS

```
$ gunzip -c lt-vexpress64-openembedded_minimal-armv8-gcc-4.9_20150620-722.img.gz >
$RELEASE_DIR/tmp/dom0_fs/root/linaro-openembedded-fs.img
```

4. Add a configuration file for your domU as detailed in 5.5.3 step 3, name the file `linaro-openembedded.cfg` and make the following edits:

   - Set the domU to have a different name:

   ```
   name = “linaro-openembedded”
   ```

   - Set the “disk” line to read *(take note of xvdb!)*:

   ```
   disk = [ 'phy:/dev/loop2,xvdb,w' ]
   ```

   - Set the “extra” line to read as follows (*take note of xvdb2)*:

   ```
   extra = "console=hvc0 earlyprintk=xenboot root=/dev/xvdb2 rw”
   ```

5. Unmount and detach the dom0 file system.

```
$ sudo umount /dev/loop0
$ sudo losetup -d /dev/loop0
```

**Usage Note:** On first boot of the Linaro OpenEmbedded FS, use ‘root’ for both the username and password

## 5.6 Building Emulated NAND files

QEMU requires a file to use for space for the NAND devices. Use these commands to generate these files.

```
$ qemu-img create -f qcow2 $RELEASE_DIR/XenZynqDist/images/linux/nand0.qcow2 193K
$ qemu-img create -f qcow2 $RELEASE_DIR/XenZynqDist/images/linux/nand1.qcow2 193K
```

## 5.7 Running the System on QEMU

1. Boot QEMU by running the following command:

```
$ cd $RELEASE_DIR/XenZynqDist
$ qemu-system-aarch64 -L $RELEASE_DIR/petalinux-v2016.3-final/etc/qemu -M arm-generic-fdt -
device loader,addr=0xfd1a0104,data=0x8000000e,data-len=4 -serial mon:stdio -serial
/dev/null -display none -device
loader,file=$RELEASE_DIR/XenZynqDist/images/linux/bl31.elf,cpu=0 -device
loader,file=$RELEASE_DIR/XenZynqDist/images/linux/u-boot.elf -gdb tcp::9000 -tftp /tftpboot
-drive file=$RELEASE_DIR/XenZynqDist/images/dom0.img,format=raw,if=sd -redir tcp:2222::22 -
net nic,vlan=0 -net user,vlan=0 -net nic,vlan=0 -net nic,vlan=0 -net nic,vlan=0 -hw-dtb
```

```
$RELEASE_DIR/XenZynqDist/images/linux/zynqmp-qemu-arm.dtb -pflash
$RELEASE_DIR/XenZynqDist/images/linux/nand0.qcow2
```

2. Start your customized domain.

   For the standard file system created using BuildRoot:

```
[root@xilinx-dom0 ~]# losetup /dev/loop0 /root/Dom1.img
[root@xilinx-dom0 ~]# xl create /etc/xen/dom1.cfg -c
```

   If you're using the Ubunt Core FS or Linaro OpenEmbedded FS they can be started similarly:

```
[root@xilinx-dom0 ~]# losetup /dev/loop1 /root/ubuntu-core-fs.img
[root@xilinx-dom0 ~]# xl create /etc/xen/ubuntu-core.cfg -c
```

   And

```
[root@xilinx-dom0 ~]# losetup /dev/loop2 /root/linaro-openembedded-fs.img
[root@xilinx-dom0 ~]# xl create /etc/xen/linaro-openembedded-fs.cfg -c
```

***TIP:*** *There are times when the domain is created in a paused state. To correct this problem, enter the following commands below:*

```
[root@xilinx-dom0 ~]# xl unpause dom1
[root@xilinx-dom0 ~]# xl console dom1
```

3. Verify that the new domain is running.

```
[root@xilinx-dom0 ~]# xl list
Name                                        ID   Mem VCPUs      State   Time(s)
dom0                                         0   512     1     r-----      36.7
dom1                                         1   128     1     -b----      10.3
```

## 5.8 Running the System on the ZCU102

1. Populate your SD Card (This only needs to be done when a change is made to the file system)

1.1 Insert your SD Card into your host machine.

1.2 Figure out which device the SD Card shows up as. It should be the last device that shows up.

```
$ dmesg
```

1.3 Copy the Dom0 file system to SD Card (Our device is /dev/sdb):

```
$ sudo dd if=$RELEASE_DIR/XenZynqDist/images/dom0.img of=/dev/sdb bs=1M
```

1.4 Unmount SD Card and place it back in the ZCU102.

2. Copy the prebuilt ZCU102 boot script to the build directory:

```
$ cp $RELEASE_DIR/dist/dist_zcu102_boot.tcl $RELEASE_DIR/XenZynqDist/zcu102_boot.tcl
```

3. In a new terminal, connect to the ZCU102 UART, assuming the device is mounted to /dev/ttyUSB2:

```
$ sudo screen /dev/ttyUSB2 115200
```

4. Restart the ZCU102 using the power switch
5. In the other terminal, connect to the jtag, load boot images, and run them:

```
$ cd $RELEASE_DIR/XenZynqDist
$ sudo /opt/Xilinx/SDK/2016.3/bin/xsdb zcu102_boot.tcl
```

6. In the screen terminal, stop the U-Boot autoboot and set the following environment variables from the nework values from earlier (Use an open IP address on your network for ipaddr):

```
U-Boot-PetaLinux> setenv serverip  xxx.xx.xxx.xxx
U-Boot-PetaLinux> setenv gatewayip xxx.xx.xxx.xxx
U-Boot-PetaLinux> setenv netmask   xxx.xx.xxx.xxx
U-Boot-PetaLinux> setenv ipaddr    xxx.xx.xxx.xxx
U-Boot-PetaLinux> run xen
```

7. Log into the system using root/root as the username and password.

8. Start your customized domain.

   For the standard file system created using BuildRoot:

```
[root@xilinx-dom0 ~]# losetup /dev/loop0 /root/Dom1.img
[root@xilinx-dom0 ~]# xl create /etc/xen/dom1.cfg -c
```

   If you're using the Ubunt Core FS or Linaro OpenEmbedded FS they can be started similarly:

```
[root@xilinx-dom0 ~]# losetup /dev/loop1 /root/ubuntu-core-fs.img
[root@xilinx-dom0 ~]# xl create /etc/xen/ubuntu-core.cfg -c
```

   And

```
[root@xilinx-dom0 ~]# losetup /dev/loop2 /root/linaro-openembedded-fs.img
[root@xilinx-dom0 ~]# xl create /etc/xen/linaro-openembedded-fs.cfg -c
```

***TIP:*** *There are times when the domain is created in a paused state. To correct this problem, enter the following commands below:*

```
[root@xilinx-dom0 ~]# xl unpause dom1
[root@xilinx-dom0 ~]# xl console dom1
```

9. Verify that the new domain is running.

```
[root@xilinx-dom0 ~]# xl list
Name                                        ID   Mem VCPUs      State   Time(s)
dom0                                         0   512     1     r-----      36.7
dom1                                         1   128     1     -b----      10.3
```

## 5.9. Creating More Guests

To create more guests from source, just follow the steps in section 5.5, but anywhere "Dom1" is used, use the name of the new domain, such as Dom2.

# *Chapter 6 Xen on Zynq*

## 6.1. Xen Boot Process

Running Xen requires booting two kernels: the Xen kernel and the dom0 (or control) kernel, which at this point is a Linux kernel. Both kernels are loaded into the proper memory locations by the boot loader. Once the boot loader has finished the initialization of the system, it passes control of the boot process over to Xen.

Xen then performs some additional hardware initialization such as initializing the Zynq hardware so that Xen can map and handle requests from the device drivers used by the dom0 kernel. More details on the Xen boot process can be found on the Xen Wiki (http://www.xenproject.org/help/wiki.html).

Once Xen performs its initialization, it then loads the dom0 (usually Linux) kernel and the RAM disk into memory. The dom0 kernel is then booted by Xen inside the privileged virtual machine domain; from the perspective of the kernel and the user this boot process is identical to Linux booting directly on the system hardware. Once dom0 has booted, access to Xen can be configured for each unprivileged domain via the dom0 interface to Xen. Dom0 has special privileges allowing it to perform this configuration, among them being the ability to access the system hardware directly. It also runs the Xen management toolstack, briefly described below.

## 6.2. xl – Interfacing to Xen

xl provides an interface to interact with Xen. It is the standard Xen project supported toolstack provided with the Xen hypervisor for virtual machine configuration, management, and debugging.

### 6.2.1. Listing Domains

'xl list' is used to list the running domains and their states on the system.

```
# xl list

Output:
Name                        ID   Mem VCPUs      State  Time(s)
Domain-0                     0  2048     2     r-----    32.0
dom1                         1  1024     2     r-----     7.3
```

### 6.2.2. Creating a Guest Domain

The following command will start a new domain using the provided configuration file argument. The name of the domain is determined by the value set in the configuration file.

```
# xl create -c /etc/xen/dom1.cfg
```

### 6.2.3. Shutting Down or Destroying a Guest Domain

'xl shutdown' is the command that should be used to shut down a running guest domain on the system. It performs a graceful shutdown of the domain using the built-in shutdown features of the domain's operating system, allowing the domains files system to close safely, and releasing any hardware resources (RAM, CPU Time, etc.) back to the system.

```
# xl shutdown dom1
```

The 'xl destroy' command should only be used as a last resort on unresponsive domains that are not removed when given the 'xl shutdown' command. The destroy command does not perform a graceful shutdown and potentially could corrupt the guest domain's file system as well as any I/O devices to which the domain has been given access.

```
# xl destroy dom1
```

### 6.2.4. Switching Between Domains

To access the command-line interface of a running domain, use the following command:

```
# xl console <domain-name>
```

where <domain-name> is the name of the specific domain of interest (the names of all running domains can be viewed using the `xl list` command detailed above).

To return to the dom0 command-line interface, press the key combination: CTRL-].

## 6.3. xentop – Analyzing Domain Resource Utilization

xentop is part of the Xen tools and is located on the privileged domain, dom0. It can be used to analyze the performance of the various domains present on the system.

The default xentop display gives you some basic stats on the overall system such as number of domains, domain state, total memory, memory being used, CPU core count and speed, and Xen version number. It also provides some similar domain specific information like CPU usage and memory usage, network utilization (if using the default para-virtualized network bridge), and physical CPU (pCPU) usage per domain virtual CPU (vCPU).

To start xentop issue the following command in the dom0 terminal

```
# xentop
```

xentop also accepts several command line arguments that can alter is behavior and the information that is displays. Those command line arguments are as follows:

*-h, --help*
Show the help message and exit

*-V, --version*
Show version information and exit

*-d, --delay=SECONDS*
Sets the number of seconds between updates (default 3)

*-n, --networks*
Displays network information in xentop console

*-x, --vbds*
Show virtual block device usage data

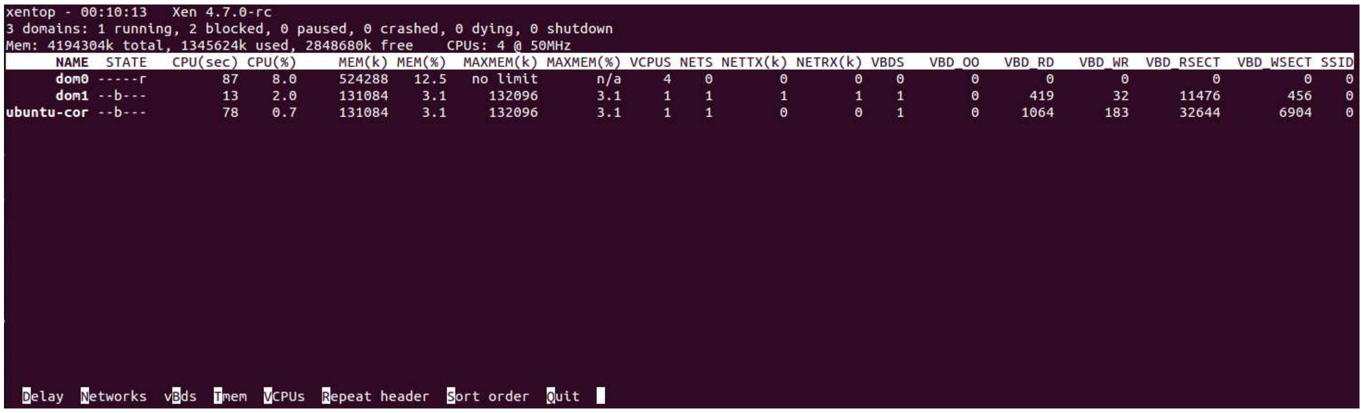
*-r, --repeat-header*
Repeat table header before each domain in the list

*-v, --vcpus*
Shows each vCPU's execution time in seconds below its assigned domain

*-f, --full-name*
Forces xentop to output the full domain name (default is truncated)

*-b, --batch*
Redirects xentop output data to stdout (batch mode)

*-i, --iterations=ITERATIONS*
Sets the maximum number of metric updates that xentop should produce before ending
Following is an example of the xentop console with no additional commandline arguments:



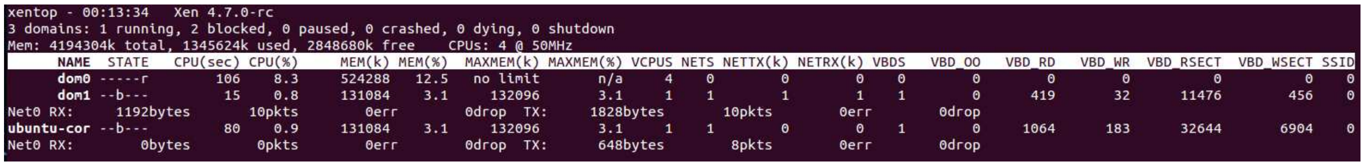
```
xentop - 00:10:13   Xen 4.7.0-rc
3 domains: 1 running, 2 blocked, 0 paused, 0 crashed, 0 dying, 0 shutdown
Mem: 4194304k total, 1345624k used, 2848680k free    CPUs: 4 @ 50MHz
      NAME  STATE   CPU(sec) CPU(%)     MEM(k) MEM(%)  MAXMEM(k) MAXMEM(%) VCPUS NETS NETTX(k) NETRX(k) VBDS   VBD_OO   VBD_RD   VBD_WR  VBD_RSECT  VBD_WSECT SSID
      dom0 -----r         87    8.0     524288   12.5   no limit       n/a     4    0        0        0    0        0        0        0          0          0    0
      dom1 --b---         13    2.0     131084    3.1     132096       3.1     1    1        1        1    1        0      419       32      11476        456    0
ubuntu-cor --b---         78    0.7     131084    3.1     132096       3.1     1    1        0        0    1        0     1064      183      32644       6904    0

  Delay  Networks  vBds  Tmem  VCPUs  Repeat header  Sort order  Quit
```

In addition to the commandline arguments, the display of xentop can be changed while it is running by pressing the following letters (also displayed in the menu at the bottom of the xentop console):

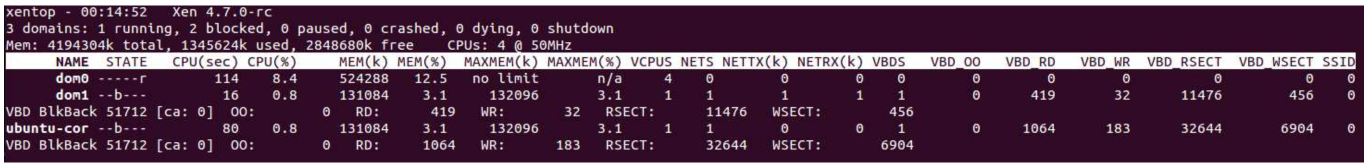
D – Prompts you to set the delay, in seconds, between performance metric updates

N - Toggles display of network information for each domain. Example:



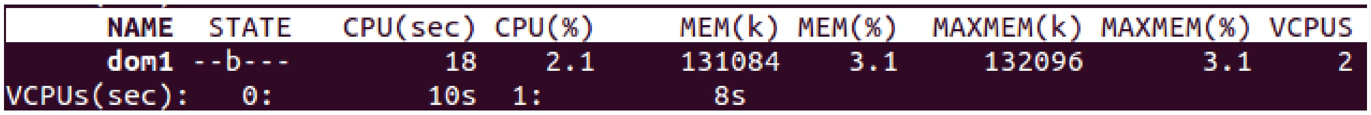
```
xentop - 00:13:34   Xen 4.7.0-rc
3 domains: 1 running, 2 blocked, 0 paused, 0 crashed, 0 dying, 0 shutdown
Mem: 4194304k total, 1345624k used, 2848680k free    CPUs: 4 @ 50MHz
      NAME  STATE   CPU(sec) CPU(%)     MEM(k) MEM(%)  MAXMEM(k) MAXMEM(%) VCPUS NETS NETTX(k) NETRX(k) VBDS   VBD_OO   VBD_RD   VBD_WR  VBD_RSECT  VBD_WSECT SSID
      dom0 -----r        106    8.3     524288   12.5   no limit       n/a     4    0        0        0    0        0        0        0          0          0    0
      dom1 --b---         15    0.8     131084    3.1     132096       3.1     1    1        1        1    1        0      419       32      11476        456    0
Net0 RX:      1192bytes      10pkts        0err       0drop  TX:     1828bytes       10pkts      0err       0drop
ubuntu-cor --b---         80    0.9     131084    3.1     132096       3.1     1    1        0        0    1        0     1064      183      32644       6904    0
Net0 RX:         0bytes       0pkts        0err       0drop  TX:      648bytes        8pkts      0err       0drop
```

B – Toggles display of virtual block device information for each domain. Example:



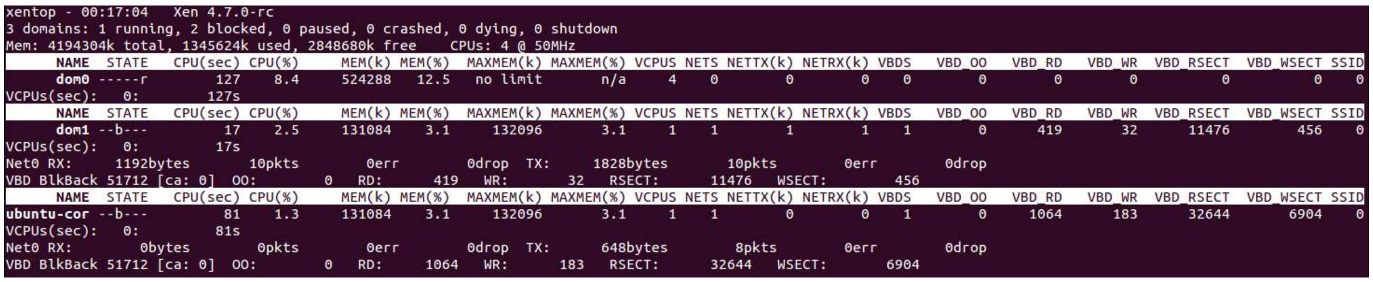
```
xentop - 00:14:52   Xen 4.7.0-rc
3 domains: 1 running, 2 blocked, 0 paused, 0 crashed, 0 dying, 0 shutdown
Mem: 4194304k total, 1345624k used, 2848680k free    CPUs: 4 @ 50MHz
      NAME  STATE   CPU(sec) CPU(%)     MEM(k) MEM(%)  MAXMEM(k) MAXMEM(%) VCPUS NETS NETTX(k) NETRX(k) VBDS   VBD_OO   VBD_RD   VBD_WR  VBD_RSECT  VBD_WSECT SSID
      dom0 -----r        114    8.4     524288   12.5   no limit       n/a     4    0        0        0    0        0        0        0          0          0    0
      dom1 --b---         16    0.8     131084    3.1     132096       3.1     1    1        1        1    1        0      419       32      11476        456    0
VBD BlkBack 51712 [ca: 0]  OO:         0   RD:       419   WR:        32   RSECT:       11476   WSECT:         456
ubuntu-cor --b---         80    0.8     131084    3.1     132096       3.1     1    1        0        0    1        0     1064      183      32644       6904    0
VBD BlkBack 51712 [ca: 0]  OO:         0   RD:      1064   WR:       183   RSECT:       32644   WSECT:        6904
```

V - Toggles display of vCPU run time in seconds. Example of a domain with 2 vCPUs (0 and 1):



```
      NAME  STATE   CPU(sec) CPU(%)     MEM(k) MEM(%)  MAXMEM(k) MAXMEM(%) VCPUS
      dom1 --b---         18    2.1     131084    3.1     132096       3.1     2
VCPUs(sec):   0:         10s   1:          8s
```

R - Toggles printing the table headers before each domain listing. Example:



```
xentop - 00:17:04   Xen 4.7.0-rc
3 domains: 1 running, 2 blocked, 0 paused, 0 crashed, 0 dying, 0 shutdown
Mem: 4194304k total, 1345624k used, 2848680k free    CPUs: 4 @ 50MHz
      NAME  STATE   CPU(sec) CPU(%)     MEM(k) MEM(%)  MAXMEM(k) MAXMEM(%) VCPUS NETS NETTX(k) NETRX(k) VBDS   VBD_OO   VBD_RD   VBD_WR  VBD_RSECT  VBD_WSECT SSID
      dom0 -----r        127    8.4     524288   12.5   no limit       n/a     4    0        0        0    0        0        0        0          0          0    0
VCPUs(sec):   0:        127s
      NAME  STATE   CPU(sec) CPU(%)     MEM(k) MEM(%)  MAXMEM(k) MAXMEM(%) VCPUS NETS NETTX(k) NETRX(k) VBDS   VBD_OO   VBD_RD   VBD_WR  VBD_RSECT  VBD_WSECT SSID
      dom1 --b---         17    2.5     131084    3.1     132096       3.1     1    1        1        1    1        0      419       32      11476        456    0
VCPUs(sec):   0:         17s
Net0 RX:      1192bytes      10pkts        0err       0drop  TX:     1828bytes       10pkts      0err       0drop
VBD BlkBack 51712 [ca: 0]  OO:         0   RD:       419   WR:        32   RSECT:       11476   WSECT:         456
      NAME  STATE   CPU(sec) CPU(%)     MEM(k) MEM(%)  MAXMEM(k) MAXMEM(%) VCPUS NETS NETTX(k) NETRX(k) VBDS   VBD_OO   VBD_RD   VBD_WR  VBD_RSECT  VBD_WSECT SSID
ubuntu-cor --b---         81    1.3     131084    3.1     132096       3.1     1    1        0        0    1        0     1064      183      32644       6904    0
VCPUs(sec):   0:         81s
Net0 RX:         0bytes       0pkts        0err       0drop  TX:      648bytes        8pkts      0err       0drop
VBD BlkBack 51712 [ca: 0]  OO:         0   RD:      1064   WR:       183   RSECT:       32644   WSECT:        6904
```

S – Cycles through which table header to use for sorting (default is to sort by domain name)

Arrow Keys - Scroll through the domain display

Q, Esc – Quit xentop

## 6.4. Shared Memory

The Xen Project Hypervisor uses what it calls "grant tables" to provide a generic mechanism for memory sharing between domains. This shared memory interface is used to implement several Xen features, including PV (split) drivers for various I/O such as block devices and network interface cards.

Each domain has its own grant table. This is a data structure that is shared with Xen (via xenstore) and allows the domain to communicate to Xen what permissions other domains have on its memory pages. Entries in the grant table are identified by grant references. A grant reference acts as a "capability" which the grantee (client) can use to perform operations on the granter's (server's) memory.

This capability-based system allows shared-memory communications between unprivileged domains. A grant reference also encapsulates the details of a shared page, removing the need for a domain to know the real machine address of a page it is sharing.

### 6.4.1. libvchan

The virtual channel library or libvchan, included with Xen since version 4.2, is designed to make it easier to setup a shared memory ring between two guest domains. This implementation of shared memory uses Xen grant tables and event channels to provide options for both streaming-based communication and packet-based communication. Even if your application requires a different communication implementation than those provided, libvchan serves as a good example of how to setup a robust interdomain communications method.

As mentioned above, libvchan relies on grant table and event channel drivers. Therefore you will need to make sure you have a kernel in each guest domain with the appropriate xenstore, grant table and event channel drivers enabled including: gntdev, gntalloc and evtchn. All of these drivers are already enabled in the default Dom1-Kernel included with XZD.

Shared memory communication using libvchan involves a server that grants access and a client that makes use of the granted access. The server offers the memory used for communication, and advertises its grant references and event channel information via xenstore. The client must know the server's domain id and service path so that it can then obtain the server's information from the xenstore. Using this information, the client then maps the server's shared memory pages into an identically sized memory buffer in its own virtual address space. The two domains can now begin to communicate with one another.

The server and client each have their own separate dedicated ring buffer for writes. This avoids any race conditions during concurrent writes. The server reads from client's ring buffer and vice versa. The credentials for each ring is stored in a shared data control structure.

Shared memory size is based on page granularity. libvchan allows users to specify the size of the ring, which is bounded by the grant table size limit configured for each domain. The server and client can also be configured to have blocking or non-blocking behavior.

**TIP:** *More specific documentation can be found in the libvchan library code located in the /$RELEASE_DIR/XenZynqDist/components/apps/xen/xen-src/tools/libvchan folder. The example programs used in 6.4.2 are built by the Makefile in that folder as well.*

### 6.4.2. Example: Using libvchan for Inter-domain Communication

**TIP:** *This example assumes you have completed the steps in Chapter 3 and Chapter 4 and have booted either a QEMU or ZCU102 setup. If you have not completed these steps, please do so before continuing.*

This section will walk you through setting up and using a basic program that utilizes libvchan to send character strings between two domains. This example requires that you build the Xen tools which will build the associated libvchan example program. Building the Xen tools can be accomplished by following all the steps in Section 5.1 and Section 5.2, and just Step 1 through Step 6 in Section 5.3; no additional steps in Section 5.3 are needed for this example. Once the Xen tools have been built, we can copy the necessary files to the existing dom0 and dom1 file system as follows:

1. Attach dom0's FS to a loop device, and mount it to a temporary directory.

```
$ cd $RELEASE_DIR
$ sudo losetup -o 1048576 /dev/loop0 $RELEASE_DIR/dist/images/dom0.img
$ mkdir -p $RELEASE_DIR/tmp/dom0_fs
$ sudo mount /dev/loop0 $RELEASE_DIR/tmp/dom0_fs
```

2. Make a temporary mounting directory for the Guest FS image.

```
$ mkdir -p $RELEASE_DIR/tmp/dom1_fs
```

3. Attach the Guest FS image to a loop device.

```
$ sudo losetup -o 1048576 /dev/loop1 $RELEASE_DIR/tmp/dom0_fs/root/Dom1.img
```

4. Mount the Guest FS image to the temporary mounting directory.

```
$ sudo mount /dev/loop1 $RELEASE_DIR/tmp/dom1_fs
```

5. Copy the libvchan files to the dom0 and dom1 file systems

```
$ sudo cp -r $RELEASE_DIR/XenZynqDist/components/apps/xen/xen-src/tools/libvchan/
$RELEASE_DIR/tmp/dom0_fs/root/libvchan-example
$ sudo cp -r $RELEASE_DIR/XenZynqDist/components/apps/xen/xen-src/tools/libvchan/
$RELEASE_DIR/tmp/dom1_fs/root/libvchan-example/
```

6. Unmount and detach the Guest and dom0 file systems

```
$ sudo umount /dev/loop1
$ sudo losetup -d /dev/loop1
$ sudo umount /dev/loop0
$ sudo losetup -d /dev/loop0
```

If you are using QEMU no further steps are required. If you plan to use the ZCU102 dev board, please follow the steps in Section 4.2 to place this updated dom0 image on your SD card.

Now that we have the necessary files on the file systems, boot into Xen (Section 4.1 for QEMU and Section 4.2 for the ZCU102) and start up dom1 according the instructions in Section 4.3.1. We will be using the program "vchan-node1" located in the /root/libvchan-example folder we copied to dom0 and dom1 to allow dom1 to receive libvchan messages from dom0. Please use the following steps:

1. Open the dom1 console, login (root, root) and enter the libvchan-example directory:

```
[root@xilinx-dom0 ~]# xl console dom1
[root@xilinx-dom1 ~]# cd /root/libvchan-example
```

2. Next, execute the vchan-node1 program in server mode, and tell it to read from dom0

```
[root@xilinx-dom1 ~]# ./vchan-node1 server read 0 data/vchan
```

3. Use the keyboard shortcut *Ctrl + ]* to return to the dom0 console, then enter the libvchan-example directory located there

```
[root@xilinx-dom0 ~]# cd /root/libvchan-example
```

4. Look up the domain ID for dom1 using the following command

```
[root@xilinx-dom0 ~]# xl domid dom1
```

5. We can now pass that ID as a commandline parameter to the vchan-node1 program in order to write a message to dom1. Fill in the domid reported in the previous step for ${domid} in the two locations in the command below

```
[root@xilinx-dom0 ~]# ./vchan-node1 client write ${domid} /local/domain/${domid}/data/vchan
```

6. Now we can type a message to dom1 and press return to send it

```
#Hello dom1, this is dom0!
#
```

7. Now press Ctrl + C to end the program and switch back to the dom1 console to verify the message was received in dom1

```
[root@xilinx-dom0 ~]# xl console dom1
```

If everything worked correctly, you will the message you typed in dom0 has appeared on the dom1 console. Below is an example run through with all the associated output:

```
[root@xilinx-dom0 ~]# losetup /dev/loop0 Dom1.img
[root@xilinx-dom0 ~]# xl create /etc/xen/dom1.cfg
Parsing config from /etc/xen/dom1.cfg
[  130.433327] device vif1.0 entered promiscuous mode
[  130.500572] IPv6: ADDRCONF(NETDEV_UP): vif1.0: link is not ready
 [  134.818034] xen-blkback: event-channel 3
[  134.823208] xen-blkback: /local/domain/1/device/vbd/51712:using single page: ring-ref 8
[  134.847195] xen-blkback: ring-pages:1, event-channel 3, protocol 1 (arm-abi) persistent
grants
[  135.360219] vif vif-1-0 vif1.0: Guest Rx ready
[  135.361487] IPv6: ADDRCONF(NETDEV_CHANGE): vif1.0: link becomes ready
[  135.368089] xenbr0: port 2(vif1.0) entered forwarding state
[  135.368936] xenbr0: port 2(vif1.0) entered forwarding state
(XEN) mm.c:1266:d0v0 gnttab_mark_dirty not implemented yet
[root@xilinx-dom0 ~]# xl console dom1
###### Dom1 boot log output appears before login
Welcome to Buildroot
xilinx-dom1 login: root
Password:
[root@xilinx-dom1 ~]# cd libvchan-example/
[root@xilinx-dom1 libvchan-example]# ./vchan-node1 server read 0 data/vchan
seed=160
##### Press Ctrl + ]
[root@xilinx-dom0 ~]#
[root@xilinx-dom0 ~]# cd /root/libvchan-example
[root@xilinx-dom0 libvchan-example]# xl domid dom1
1
[root@xilinx-dom0 libvchan-example]# ./vchan-node1 client write 1 /local/domain/1/data/vchan
seed=417
Hello dom1, this is dom0!
#^C
[root@xilinx-dom0 libvchan-example]# xl console dom1
#Hello dom1, this is dom0!
#read vchan: Success
[root@xilinx-dom1 libvchan-example]#
```

# Chapter 7 Bare Metal Guests

## 7.1. Introduction

In order to simplify the process of porting a "standalone" application, one that runs on bare metal without an operating system, to run as a Xen guest, DornerWorks has developed a Bare Metal Container (BMC). The BMC provides the libraries and tools for allowing bare metal applications developed using other tool flows, such as Xilinx's SDK (XSDK), to run as guests under Xen on the Xilinx Zynq UltraScale+ MPSoC. The container provides virtual memory mapping, stack, fault handling, and an API to print to Xen's console. After setting up the environment, the XZD Bare Metal container loads the payload application into virtual memory and then passes control to it.

### *7.1.1. Bare Metal Guest Bootup*

dom0 starts up the bare metal guest using the Xen tools, xl toolstack. As part of initializing the guest, Xen allocates the configured amount of memory for it, and then copies the bare metal image to that memory space, using information provided in the image header. At that point, Xen starts execution at Exception Level 1 at offset into the newly allocated memory space indicated by the image header.

The bare metal guest enters into src/head.S and initializes the low level register settings that the guest will need to be set. Currently this includes setting up a direct guest physical address(GPA) to virtual address (VA) mapping of UART1 (1 page at 0xFF010000) as well as 1 page of potentially shared memory at 0x7FFFF000. This file should be modified to add additional MMU mappings if other memory regions are shared or I/O devices are passed through to the guest. This file should also be modified, along with src/main.c if the 4MB limit needs to be changed or a different application execution space is desired.

Next, the bare metal container branches to the `arch_init` function in src/setup.c. This is the first C code that gets executed. Here the physical offset is saved to a global variable so it can be used for any direct memory writes. Then a message gets printed on Xen's console to indicate the bare metal guest has booted to a point where it is about to transfer control to the payload application. This is done using the console_io hypercall. This call places a length and a character buffer in the correct registers and then interrupts into Xen. Xen then prints that buffer onto its own console.

After this, the bare metal container calls the `main` function in src/main.c. Here the bare metal guest loads the 4MB of the payload application to memory at 0x40400000 and then passes control to the payload application. This is where the location and size of the payload application can be changed as long as it remains consistent with src/head.S and the application's link map.

### *7.1.2. Payload Application*

The bare metal container is designed to accept a payload application in the form of an ELF file, such as a standalone application generated by XSDK. The ELF file is converted to its binary equivalent and built with the bare metal container, creating a resulting binary image that can be run from Dom0's command line, by using the `./build_it` utility included in the repository.

Alternatively code can be modified in the src/setup.c and/or src/main.c to include the functionality directly. Application should begin at the `main` function. Any new C file can be added to the src directory and the Makefile will include it in the build without modifying any Makefiles or config files. The include directory is the only include directory, so that is where all header files should be placed.

Because of the nature of Xen, the bare metal guest only has access to what it is assigned to in the configuration file that is used to create the guest. There are a couple of options to get access to a device:

- **Direct Passthrough** - This gives a guest direct and usually exclusive access to a device. Information on device passthrough can be found in section 9.2 *Passthrough*. The example provided in this chapter includes passthough of UART1 to the guest(s). Note in this case since UART1 does not require interrupts or SMMU for proper use, it can be passed through to multiple guests. However, care must be taken to coordinate access of that device between the guests using it, or unspecified behavior could result.
- **Paravirtualized Devices** - Dom0 provides back-end drivers for some devices. To communicate with these drivers, front-end drivers need to be implemented for the bare metal application. This requires support for the XenStore, event channels, shared memory, and all of the related hypercalls. Support for Xen's virtual console has been added for FreeRTOS and bare metal guests.

## 7.2. Building the Bare Metal Guest

The following instructions assume you have followed Chapter 5 already.

### *7.2.1. Creating Payload Application*

You are free to create your own payload application using whatever toolchain or workflow you like, as long as the resulting application meets these requirements:

- Runs at EL1 or EL0,
- Is contained in a single ELF file,
- Uses less than 4MB of memory, and
- Is linked to, or can otherwise run at, address 0x40400000.

The first requirement is to ensure the guest runs at EL lower than Xen, which runs at EL2.

The second requirement can be mitigated with changes to the `build_it` script.

The last two requirements can be increased or altered with changes to the BMC source code.

An example application is provided in the XZD that can be built using Xilinx's SDK (XSDK) by following the following steps:

1. Create a `Hello World` application in XSDK targeting the ZCU102 platform called $PROJECT_NAME in the $XSDK_WORKSPACE. This should create three projects: the application project called $PROJECT_NAME, a BSP project called $PROJECT_NAME"_bsp", and a hardware platform project called `ZynqMP_ZCU102_hw_platform`.
2. Copy the files from the XZD to overwrite those in the XSDK workspace:
```
$ cp $RELEASE_DIR/misc/examples/baremetal/app/src/* $XSDK_WORKSPACE/$PROJECT_NAME/src/
$ cp $RELEASE_DIR/misc/xzd_bmc/xzd_bmc.h  $XSDK_WORKSPACE/$PROJECT_NAME/src/
$ cp -r $RELEASE_DIR/misc/examples/baremetal/bsp/* $XSDK_WORKSPACE/$PROJECT_NAME"_bsp"/
```
3. Clean and rebuild the XSDK project.

### 7.2.2. Building the Guest Image

To build the bare metal image, run the following command, targeting the ELF file you want to be your payload application:

```
$ cd $RELEASE_DIR/misc/xzd_bmc
$ ./build_it $XSDK_WORKSPACE/$PROJECT_NAME/Debug/$PROJECT_NAME.elf
```

Example: if you created your XSDK workspace called "workspace" in your home directory, and named your XSDK project "hello_world", then the resulting command would be:
```
./build_it ~/workspace/hello_world/Debug/hello_world.elf
```

This generates an `xzd_bare.elf` file and an `xzd_bare.img` binary image. The binary image is the file that will be used as the `kernel` for the Xen guest. The `xzd_bare.elf` file can be used for debugging the bare metal container portion and $PROJECT_NAME.elf can be used to debug the payload application.

## 7.3. Installing and Running the Guest Image in the XZD

### 7.3.1. Guest Image

The `xzd_bare.img` file needs to be transferred to dom0's file system, typically by adding it to the file system or by transferring it using TFTP or other network transfer protocol.

**SD Card File System**

This method is to simply mount the file system found on your SD card and copy the `xzd_bare.img` file directly to it. This is the easiest method if you already have your File System populated to its own partition on an SD card.

1. Insert SD card.
    a. Mount SD partition if it does not automatically do so.
2. Copy files to the SD partition.
    a. `su cp xzd_bare.img $YOUR_MEDIA_MOUNT_PATH/root/.`
3. Gracefully eject SD card using your host system's recommended method.

```
a. umount $YOUR_MEDIA_MOUNT_PATH
```

**File System Image**

This method is used to copy the `xzd_bare.img` file into the file system image file, dom0.img. This approach is good when you haven't populated an SD card yet, want to create a file system image populated with `xzd_bare.img` that many others will use, or are using QEMU.

1. Mount the file system image.
   ```
   $ sudo losetup -o 1048576 /dev/loop0 $RELEASE_DIR/XenZynqDist/images/dom0.img
   $ mkdir -p $RELEASE_DIR/tmp/dom0_fs
   $ sudo mount /dev/loop0 $RELEASE_DIR/tmp/dom0_fs
   ```
2. Move the bare metal image into dom0'S file system.
   ```
   $ sudo mv $RELEASE_DIR/xzd_bare.img $RELEASE_DIR/tmp/dom0_fs/root/xzd_bare1.img
   ```
3. Unmount and detach the dom0 file system.
   ```
   $ cd $RELEASE_DIR
   $ sudo umount /dev/loop0
   $ sudo losetup -d /dev/loop0
   ```

**Network Transfer**

This method is to use network transfer protocol, such as TFTP, to move the `xzd_bare.img` file to the target. This approach works very well with QEMU because the emulated network is always available, with the TFTP server reachable at 10.0.2.2.

1. Copy xzd_bare.img to the TFTP server's base directory.
   ```
   $ cp xzd_bare.img /tftpboot
   ```
2. From dom0, issue the command to retrieve the file.
   ```
   tftp –g –r xzd_bare.img <HOST_IP>
   ```

### *7.3.2. Guest Configuration*

Create a configuration file for the bare metal guest. An example Configuration File (/etc/xen/bare.cfg), which passes UART1 through to the guest, is shown below:

```
name = "bare"
kernel = "/root/xzd_bare.img"
memory = 8
vcpus = 1
iomem = ["0xff010,1"]
```

### *7.3.3. Running the Guest*

Boot up the XZD and in dom0 start up the bare metal guest with the following command:

```
xl create /etc/xen/bare.cfg
```

Terminate the bare metal guest with the following command:

```
xl destroy bare
```

## 7.4. XSDK Example

For the XSDK example, additional steps are needed to create a region of RAM for guests to share in the device tree. An easy way to do this is to add a reserved-memory node to the end of the AMBA node definition in the DTS file appropriate for your target (e.g., xen-qemu.dts or xen-zcu102.dts):

```
+               reserved-memory {
+                       #address-cells = <2>;
+                       #size-cells = <2>;
+                       ranges;
+                       guest_shared: guest@7ffff000{
+                               reg = <0 0x7ffff000 0 0x1000>;
+                               no-map;
+                       };
+               };
```

Examples of DTS files for both targets can be found in `$RELEASE_DIR/misc/examples/baremetal/config`.

To recompile the dts to dtb, assuming you are targeting the ZCU102 development board:

```
dtc -I dts -O dtb -o xen.dtb xen-zcu102.dts
```

Examples of DTB files for both targets can be found in `$RELEASE_DIR/misc/examples/baremetal/config/qemu` and `$RELEASE_DIR/misc/examples/baremetal/config/zcu102`. The resulting xen.dtb will need to be added to the SD card or /tftpboot directory depending on your target and method of booting it, see Chapter 4 Booting and Running XZD for more details.

To run multiple guests, the guest name needs to be unique. Furthermore, it is possible to pin guests to specific CPU cores with the `cpu` attribute. These examples assume that the `xzd_bare.img` file has been renamed to `bm.img`. A prebuilt version can be found in `$RELEASE_DIR/misc`.

bm0.cfg:

```
name = "bm0"
kernel = "bm.img"
memory = 8
vcpus = 1
cpus = [0]
iomem = [ "0x7ffff,1", "0xff010,1"]
```

bm1.cfg:

```
name = "bm1"
kernel = "bm.img"
memory = 8
vcpus = 1
cpus = [1]
iomem = [ "0x7ffff,1", "0xff010,1"]
```

bm2.cfg:

```
name = "bm2"
kernel = "bm.img"
memory = 8
vcpus = 1
cpus = [2]
iomem = [ "0x7ffff,1", "0xff010,1"]
```

bm3.cfg:

```
name = "bm3"
kernel = "bm.img"
memory = 8
vcpus = 1
cpus = [3]
iomem = [ "0x7ffff,1", "0xff010,1"]
```

All four guests can be brought up from dom0's command line:

```
$ xl create bm0.cfg
$ xl create bm1.cfg
$ xl create bm2.cfg
$ xl create bm3.cfg
```

Examples of these configuration files can be found in `$RELEASE_DIR/misc/examples/baremetal/config/`.

Please note that whenever multiple guests have access to the same resource, which could be shared memory or an I/O peripheral device, some method should be used to ensure proper coordination between the different threads of execution. The example XSDK project contains code for a spin lock to provide mutual exclusion for both the shared memory and the UART.

# Chapter 8 Other Guests

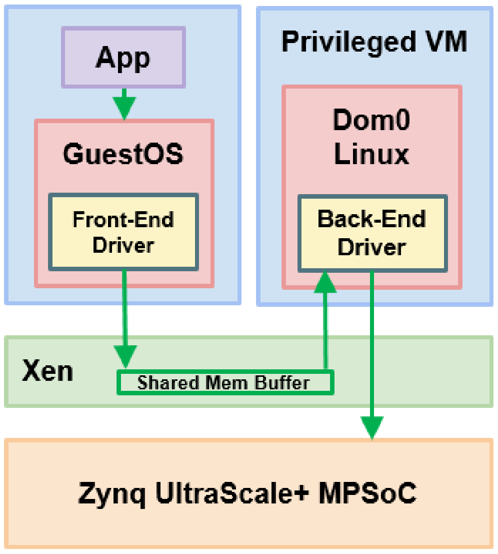
DornerWorks is constantly working on getting new operating systems to run as guests on Xen.

In addition to Linux and bare metal guests, the XZD also provides libraries for making guests using FreeRTOS run on Xen. The latest documentation can be found at http://xzdforums.dornerworks.com/showthread.php?tid=620. A copy is also provided in the distribution at $(RELEASE_DIR)/docs/XZD FreeRTOS Guest Guide.pdf.

# Chapter 9 Interacting with I/O Devices

## 9.1. Paravirtualization

Typically, *dom0* is a full featured OS like Linux, and provides a plethora of device drivers. Xen takes advantage of the availability of drivers by providing a means for guests to use those drivers through *dom0*. This requires modification of the drivers in the guest, and is an example of paravirtualization. I/O Paravirtualization uses software to share a device from a privileged guest, typically *dom0*, to any other guest that needs to access the device. The privileged guest is the only one that has direct access to the device and contains the normal device driver to interact with the device. Then what Xen calls a split driver is used to share the data from the privileged guest to the other guests. A split driver is made up of a backend driver in the privileged guest and a frontend

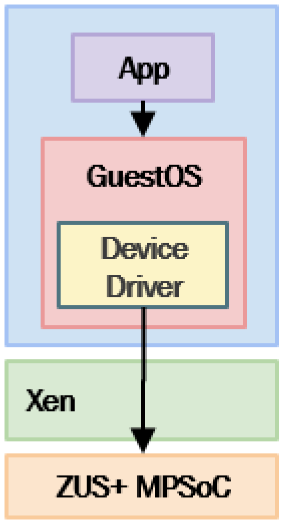


driver in the other guests that want to access the device. The backend driver sets up a shared ring buffer, and an event channel (a notification) for each guest that needs to access the device. The frontend driver in each guest then connects via a wrapper Application Program Interface (API) to those sharing mechanisms.

Since the privileged guest arbitrates access to the device, the data from the device can be shared across virtual machines without breaking partitioning. This is useful if multiple guests need to access the same I/O channel. Another advantage is that the frontend driver presents an abstraction of the specific device, so that guests can be more generic and thus more portable. This can be an initial drawback, because if the guest OS does not support that frontend driver, it needs to be developed. Paravirtualization also adds another layer to the device driver stack, therefore the performance will not be as fast as native OS usage of the device. If multiple guests are sharing the same device the privileged guest must implement an allocation scheme to prevent a guest from monopolizing that device. Since this method takes advantage of the strict memory sharing infrastructure of Xen, it is a safe and secure method for handling I/O.

## 9.2. Passthrough

### *9.2.1. Introduction*



Xen has the capability to pass access to peripheral devices through to guest domains, allowing that guest direct and unfettered access. For peripherals requiring high performance, Xen configures the system memory management unit (SMMU) to handle the necessary translations for any direct memory address (DMA) transactions initiated by that peripheral as well as configures the Generic Interrupt Controller (GIC) to pass interrupts to the guest domain. This allows the guest to use the device as if it was the only software running in the system. This passthrough capability allows for practically native performance and increases overall flexibility and stability in the system as the SMMU enforces the memory mapping, preventing a guest from using its DMA-capable peripheral to access data in another guest's memory space.

The instructions below provide a guide for how to configure the Xen system installed on the Xilinx UltraScale+ MPSoC (MPSoC) to pass an Ethernet peripheral device through to a guest.

### 9.2.2. Ethernet Passthrough

The instructions documented here are specific for the Ethernet device, however can be generalized to pass through any DMA-capable device via the SMMU.

It is assumed that the reader has an Ubuntu 14.04 or 16.04 host system and has downloaded the release image mentioned in Chapter 3 Host Setup.

#### 9.2.2.1. Modifying the Xen Device Tree

The first step in passing through an Ethernet device to a guest domain is to edit the xen.dts file located on the host computer in the XZD development system. The general process for enabling device passthrough is to disable the device from access to dom0 and to enable it for passthrough to a guest domain. Ethernet passthrough requires editing the two Ethernet devices in the xen.dts. We will disable the second Ethernet device and then enable it for device passthrough, but will also ensure that the Ethernet device assigned to domain 0 gets edited appropriately.

We need to convert the xen.dtb into a xen.dts file, which is a text file that we can edit. We will use the dtc command on the host computer to create the dts file.

```
$ dtc -I dtb -O dts -o $RELEASE_DIR/xen.dts /tftpboot/xen.dtb
```

Open the xen.dts file and find the first and second Ethernet devices for the MPSoC in xen.dts. The text found in the xen.dts should look similar to that below.

```
    ethernet@ff0b0000 {
    #stream-id-cells = <0x1>;
    compatible = "cdns,gem";
    status = "okay";
    interrupt-parent = <0x1>;
    interrupts = <0x0 0x39 0x4 0x0 0x39 0x4>;
                  .
                  .
                  .
    ethernet@ff0c0000 {
    #stream-id-cells = <0x1>;
    compatible = "cdns,gem";
    interrupt-parent = <0x1>;
    interrupts = <0x0 0x4b 0x4>;
    reg = <0x0 0xff0c0000 0x1000>;
    clock-names = "pclk", "hclk", "tx_clk";
    clocks = <0x2 0x2 0x2>;
    #address-cells = <0x1>;
    #size-cells = <0x0>;
    status = "okay";
};
```

**Figure 1: xen.dts**

Once the Ethernet devices are located, then edit the xen.dts to look like the text below. In some cases, the tags *gem0* and *gem1* are already included, in other cases, it is not. Ensure that the tags, *gem0* and *gem1* are added to the file in the location below, paying attention to the highlighted portions of the file.

```
gem0: ethernet@ff0b0000 {
      #stream-id-cells = <0x1>;
      compatible = "cdns,gem";
      status = "okay";
      interrupt-parent = <0x1>;
      interrupts = <0x0 0x39 0x4 0x0 0x39 0x4>;
      reg = <0x0 0xff0b0000 0x1000>;
      clock-names = "pclk", "hclk", "tx_clk";
      clocks = <0x2 0x2 0x2>;
      #address-cells = <0x1>;
      #size-cells = <0x0>;
      local-mac-address = [00 0a 35 00 b2 02];
      phy-handle = <0x3>;
      phy-mode = "rgmii-id";

      phy@0 {
            reg = <0x0>;
            max-speed = <0x64>;
            linux,phandle = <0x3>;
            phandle = <0x3>;
      };
};

gem1: ethernet@ff0c0000 {
    #stream-id-cells = <0x1>;
    compatible = "cdns,gem";
    interrupt-parent = <0x1>;
    interrupts = <0x0 0x4b 0x4>;
    reg = <0x0 0xff0c0000 0x1000>;
    clock-names = "pclk", "hclk", "tx_clk";
    clocks = <0x2 0x2 0x2>;
    #address-cells = <0x1>;
    #size-cells = <0x0>;
    xen,passthrough = <1>;
    status = "disabled";
};
```

**Figure 2: Edited xen.dts**

Since the Ethernet is a DMA-capable device, we will add this to the SMMU by modifying the SMMU section of the xen.dts appropriately. Search and find the SMMU section and modify the code in the file to look like the section below.

```
smmu0: smmu0@0xFD800000 {
         compatible = "arm,mmu-500";
         reg = <0x0 0xfd800000 0x20000>;
         #global-interrupts = <1>;
         interrupt-parent = <&gic>;
         interrupts = <0 157 4>,
                 <0 157 4>, <0 157 4>, <0 157 4>, <0 157 4>,
                 <0 157 4>, <0 157 4>, <0 157 4>, <0 157 4>,
                 <0 157 4>, <0 157 4>, <0 157 4>, <0 157 4>,
                 <0 157 4>, <0 157 4>, <0 157 4>, <0 157 4>;
          mmu-masters = <&gem0 0x874 &gem1 0x875>;
 };
```

**Figure 3: SMMU Modifications**

After the dts has been modified, we need to compile the dts back into the dtb and ensure that it is the saved in the correct location. Enter the following command to perform this task.

```
$ dtc -I dts -O dtb -o /tftpboot/xen.dtb $RELEASE_DIR/xen.dts
```

### 9.2.2.2. Modifying the Domain Configuration File

The next two files that require modification are done on the dom0 file system from the XZD development system. Locate the configuration file for the guest domain that will be using the passed-through Ethernet device. You will need to follow the steps in section 5.5.3, step 1 to mount the file system image. This will provide a file system that will contain the files for you to modify in the next few sections. Edit your dom1.cfg, located in $RELEASE_DIR/tmp/dom0_fs/fs/dom0/etc/xen, by adding the text below to the bottom of the file. In addition to adding the text below, make sure that you either comment out or delete the virtual device at approximately line 39. This line begins with 'vif'.

```
# Ethernet Device
# This will set up the Ethernet so that it is
# accessible to this guest domain.
dtdev = [ "/amba/ethernet@ff0c0000" ]
device_tree = "/etc/xen/xen-partial.dtb"
irqs = [ 91 ]
iomem = [ "0xff0c0,1" ]
```

**Figure 4: Editing the guest domain configuration file**

The lines above add gem1 exclusively to the guest domain, in this case, domain 1. The options needed for passthrough are defined below:

**dtdev** : The absolute path of the device to passthrough in the device tree
**device_tree** : dom0 path to partial device tree to be passed to the guest
**irqs** : IRQ number to be given to the guest
**iomem** : The physical pages to be passed in to the guest

### 9.2.2.3. Creating a Partial Device Tree

The last step in setting up the system for passthrough involves creating a device tree for the domain called a *partial device tree*.

Create a new file and name it *xen-partial.dts*. Ensure that this file is located in the **device_tree** path indicated in the configuration file shown in Figure 4. The entire *xen-partial.dts* should be saved in $RELEASE_DIR/tmp/dom0_fs/fs/dom0/etc/xen and look like the one below.

```
/dts-v1/;

/ {
    #address-cells = <0x2>;
    #size-cells = <0x1>;

    passthrough {
        compatible = "simple-bus";
        ranges;
        #address-cells = <0x2>;
        #size-cells = <0x1>;

        misc_clk {
            #clock-cells = <0x0>;
            clock-frequency = <0x17d7840>;
            compatible = "fixed-clock";
            linux,phandle = <0x2>;
            phandle = <0x2>;
        };

        ethernet@ff0c0000 {
            #stream-id-cells = <0x1>;
            compatible = "cdns,gem";
            reg = <0x0 0xff0c0000 0x1000>;
            interrupts = <0x0 0x3b 0x4 0x0 0x3b 0x4>;
            clock-names = "pclk", "hclk", "tx_clk";
            clocks = <0x2 0x2 0x2>;
            #address-cells = <0x1>;
            #size-cells = <0x0>;
            local-mac-address = [00 0a 35 00 00 01];
            phy-handle = <0x1>;
            phy-mode = "rgmii-id";

            phy@0 {
                reg = <0x0>;
                max-speed = <0x64>;
                linux,phandle = <0x1>;
                phandle = <0x1>;
            };
        };
    };
};
```

**Figure 5: xen-partial.dts**

The *xen-partial.dts* now needs to be compiled into a binary file known as a device tree blob (dtb). We will use the host's dtc compiler, mentioned above, to compile the dts file into a dtb file.

Generate the dtb file by entering the following command:

```
$ dtc -I dts -O dtb -o xen-partial.dtb xen-partial.dts
```

**Figure 6: Compile the DTS**

#### *9.2.2.4. Communicating with the Domains*

You are now ready to boot both domain 0 and domain 1, each one is assigned a unique Ethernet device. To test the Ethernet device passthrough in the QEMU environment, one can ping a domain when logged into another domain and visa-versa.

One can also use the *nc,* or *netcat,* command that tests the ability to communicate between the domains and the host computer. To log into the host computer from a domain, enter the following command on your host computer.

```
$ nc -l 4321
```

This will create a quick server that can be used to communicate to other systems via an IP address. On one of the domains, enter the following command.

```
# telnet 10.0.2.2 4321
```

This will set up a connection with the host and allow you to send characters between the domain and the host computer. Entering ctrl-c in the domain will break the connection. Now follow the same steps in the other domain to test the other Ethernet device.

# *Chapter 10 Troubleshooting*

## 10.1. Section To Be Written

# Chapter 11 Additional Support Solutions

## 11.1. Current Support Options

For more information and support, please visit our web site.

http://dornerworks.com/services/xilinxxen

Details on the Xilinx Zynq UltraScale+ MPSoC can be found here:

http://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html