FlexRAN Reference Solution Cloud-Native Setup

Installation Guide Software Release v21.03

March 2021

Revision 11.0

Intel Confidential

Document Number: 575834-11.0

Contents

Figures

Tables

Revision History

Revision Number	Description	Revision Date
11. 0	FlexRAN Software Release v21.03 changes: • Restored chapters on virtualization • Added a chapter to introduce the automation deployment solution – CIR	March 2021
10. 0	FlexRAN Software Release v20.11 changes: Removed virtualization related content	November 2020
9.0	FlexRAN Software Release v20.08 changes: • Revised Section 8.0 Intro • Section 8.4 revised • Revised code in Section 15.1 and 15.3 • Revised Section 16.0 • Revised code in Section 16.1 and 16.2	August 2020
8.0	FlexRan Software Release v20.04: • Add Kubernetes Native CPU Manager • Replaced Figure 10 • Add QAT Device Plugin	April 2020
7.0	FlexRan Software Release v20.02: • Revised Sections 2.4.3 updates to basic server set up • Revised Sections 7 and 8 revised	February 2020
6.0	Update FlexRAN build section to refer to 4G and 5G Doxygen Documents	October 2019
5.0	Flexran software release v18.12 Added Section 14: Running multiple FlexRAN PODs. Updated references to FlexRAN release 18.12 and Kubernetes* v1.11.5.	December 2018
4.0	Updated for FlexRAN software release 18.12.	October 2018
3.0	Added testing instructions.	June 2018
2.0	Q1 2018 release. Includes PTP support.	April 2018
1.0	Initial release.	January 2018

§

1.0 Introduction

Kubernetes* is an open-source software system that automates container operations in Linux*, including application deployment, scaling, and management. It serves as the basis for container management systems offered by numerous vendors. This document describes how to set up FlexRAN Reference Solution in Kubernetes containers.Terminology

Table 1. Terminology

Term	Description
BSP	Board Support Package
BMC	Baseboard Management Controller
CMK*	CPU Manager for Kubernetes*
DPDK*	Data Plane Development Kit*
FEC	Forward Error Correction
FH	Front Haul
FPGA	Field Programmable Gate Array
FRU	Field Replacement Unit
HT	Hyper Threading
ICC	Intel C++ Compiler
K8s*	Kubernetes*
NR	New Radio
OVS	Open VSwitch
PTP	Precision Time Protocol
RBAC	Role-Based Access Control
SDR	Sensor Data Record
SR-IOV	Single Root I/O Virtualization
SSH	Secure Shell
SRIOV	Single Root Input/Output Virtualization
TDP	Thermal Design Power
VF	Virtual Function
VM	Virtual Machine
WLS	Wireless subsystem interface
CIR	Common Infrastructure Release

1.1 References and Resources

Table 2. References and Resources

Document	Document No./ Location
FlexRAN 4G Reference Solution PHY Software Documentation	572318
FlexRAN Reference Solution Software Release Notes	575822
FlexRAN and Mobile Edge Compute (MEC) Platform Setup Guide	575891
FlexRAN 5GNR Reference Solution PHY Software Documentation	603577
CMK Operator Manual	https://github.com/intel/CPU-Manager-for-Kubernetes/blob/master/docs/operator.md
*CPU Manager for Kubernetes**	https://github.com/Intel-Corp/CPU-Manager-for-Kubernetes/
Node feature discovery for Kubernetes	https://github.com/kubernetes-sigs/node-feature-discovery
Intel® System Studio	https://software.intel.com/en-us/system-studio/choose-download
Kubeadm	https://github.com/kubernetes/kubeadm
CIR (Common Integration Repository)	https://gitlab.devtools.intel.com/system_integration/common_integration_repository
Kubespray	https://github.com/kubernetes-sigs/kubespray

§

2.0 Real-Time Host Installation

This section describes how to build a real-time host on the Intel® Server System.

2.1 Hardware Configuration

Table 3 lists the required Reference Platform Hardware Configuration.

Table 3. Reference Platform Hardware Configuration

Component	Specification
Board	Intel® Server Board S2600WFTF Family (Wolf Pass)
Memory	Micron*, about 192 GB DDR4 2400 MHz DIMMs
Chassis	2 U Rackmount Server Enclosure
Storage	960 Gb SSD M.2 SATA 6Gb/s
NIC1	1x Fortville NIC X710DA4 SFP+ (PCIe* Add-in-card direct to CPU-0)
NIC2	1× Fortville 40 Gbe Ethernet PCIe XL710-QDA2 Dual Port QSFP+(PCIe Add-in-card direct to CPU-0)

2.2 Software Configuration

Table 4 lists the required software packages for Bare metal.

Note: Use `kernel-rt-3.10.0-1127.19.1.rt56.1116.el7.x86_64*` for flexran releases before 20.08.

For releases after 20.08, `kernel-rt-3.10.0-1127.19.1.rt56.1116.el7.x86_64.rpm` works fine.

Table 4. Required Software for Bare Metal Configuration

Name	Required
kernel-rt-3.10.0-1127.19.1.rt56.1116.el7.x86_64.rpm*	M
kernel-rt-devel-3.10.0-1127.19.1.rt56.1116.el7.x86_64.rpm	M
kernel-rt-kvm-3.10.0-1127.19.1.rt56.1116.el7.x86_64.rpm	O
rtctl-1.13-2.el7.noarch.rpm	M
rt-setup-2.0-9.el7.x86_64.rpm	M
rt-tests-1.0-16.el7.x86_64.rpm	M
libcgroup-0.41-13.el7.x86_64.rpm	M
python-ethtool-0.8-8.el7.x86_64.rpm	M
qemu-kvm-tools-ev-2.9.0-16.el7_4.14.1.x86_64.rpm	O
tuna-0.13-9.el7.noarch.rpm	M
tuned-2.9.0-1.el7fdp.noarch.rpm	M

Name	Required
tuned-profiles-nfv-2.9.0-1.el7_5.2.noarch.rpm	O
tuned-profiles-nfv-host-2.9.0-1.el7_5.2.noarch.rpm	M
tuned-profiles-realtime-2.9.0-1.el7_5.2.noarch.rpm	M

NOTES: Use rt 3.10.0.957 for Bare Metal as we have seen L1 crashes for 1062 images in some cases.

- Use the above commands only to update the kernel. Do not use a .config file to build the kernel as it causes inconsistencies with FlexRAN software.

Note: Use `rt 3.10.0.1062` for Kubernetes only.

Table 5 Lists the required software packages for Kubernetes.

Table 5. Required Software for Kubernetes Configuration

Name	Required
kernel-rt-3.10.0-1127.19.1.rt56.1116.el7.x86_64.rpm	M
kernel-rt-devel-3.10.0-1127.19.1.rt56.1116.el7.x86_64.rpm	M
kernel-rt-kvm-3.10.0-1127.19.1.rt56.1116.el7.x86_64.rpm	O
rtctl-1.13-2.el7.noarch.rpm	M
rt-setup-2.0-9.el7.x86_64.rpm	M
rt-tests-1.0-16.el7.x86_64.rpm	M
libcgroup-0.41-13.el7.x86_64.rpm	M
python-ethtool-0.8-8.el7.x86_64.rpm	M
qemu-kvm-tools-ev-2.9.0-16.el7_4.14.1.x86_64.rpm	O
tuna-0.13-9.el7.noarch.rpm	M
tuned-2.9.0-1.el7fdp.noarch.rpm	M
tuned-profiles-nfv-2.9.0-1.el7_5.2.noarch.rpm	O
tuned-profiles-nfv-host-2.9.0-1.el7_5.2.noarch.rpm	M
tuned-profiles-realtime-2.9.0-1.el7_5.2.noarch.rpm	M

- Use the above commands only to update the kernel. Do not use a .config file to build the kernel as it causes inconsistencies with FlexRAN software.
- M – Mandatory
- O – Optional, only for the virtualized hostCentOS* image:

Go to: http://mirrors.oit.uci.edu/centos/7.8.2003/isos/x86_64/ and select a mirror link that fits.

Example select: http://mirrors.oit.uci.edu/centos/7.8.2003/isos/x86_64/CentOS-7-x86_64-Everything-2003.iso

Real-time packages and tools are located in:

http://linuxsoft.cern.ch/cern/centos/7.8.2003/rt/x86_64/Packages/

tuna-0.13-9.el7.noarch.rpm could be found in the ISO image or:

https://centos.pkgs.org/7/centos-aarch64/tuna-0.13-9.el7.noarch.rpm.html

2.3 BIOS Version

Table 6. BIOS Information

BIOS version	Wolf Pass Server
IFWI	SE5C620.86B.00.01.0013.030920180427
Baseboard Management Controller (BMC)	1.43.91f76955
Download link	https://downloadcenter.intel.com/download/27632/Intel-Server-Board-S2600WF-Family-BIOS-and-Firmware-Update-Package-for-UEFI-?product=89005

2.4 System Installation and Configuration

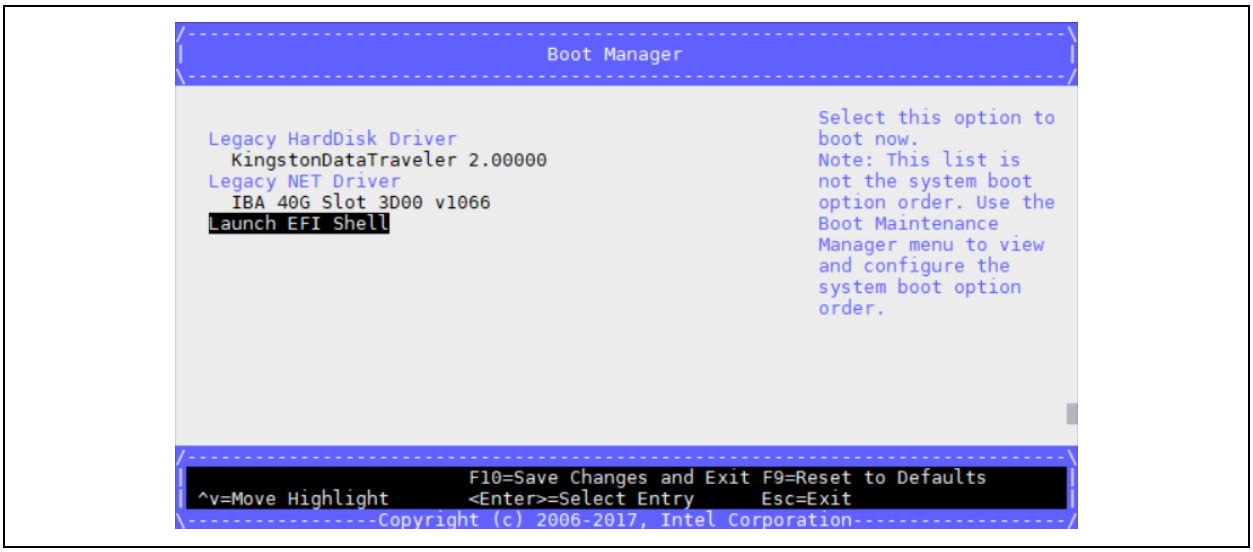
To prepare a system for Kubernetes*, do the following, which is described in the following subsections:

- Upgrade the BIOS
- Configure the BIOS
- Install and configure the Real-Time Operating System (OS)
- Test and verify the Real-Time OS

2.4.1 Upgrade BIOS

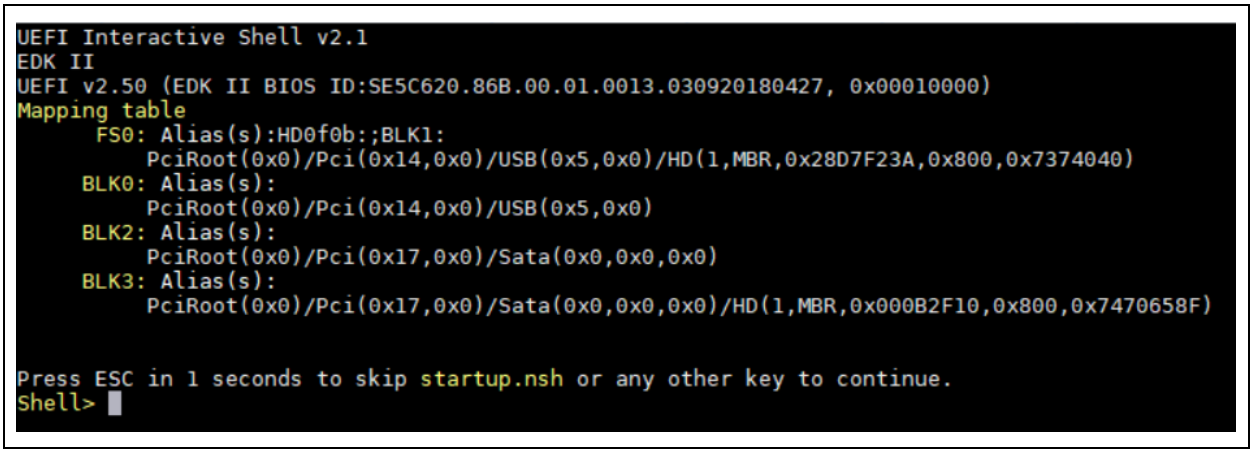
1. Download the S2600WF_EFI-BIOSR0013_ME04.00.04.294 package from Intel, as shown in Table 6.
2. Copy the BIOS image of S2600WF_EFI-BIOSR0013_ME04.00.04.294 to the USB stick.
3. Insert the USB stick into the server and boot the server into the Launch EFI shell (press **F6** during the booting phrase to enter the Boot Manager).
4. Launch the EFI Shell from the Boot Manager (refer to Figure 1).

Figure 1. Boot Manager: Launch EFI Shell Screen



1. Login using the EFI shell. (Refer to Figure 2)

Figure 2. UEFI Interactive Shell Interface



Note: FS0 in the shell interface indicates mapping to the USB stick.

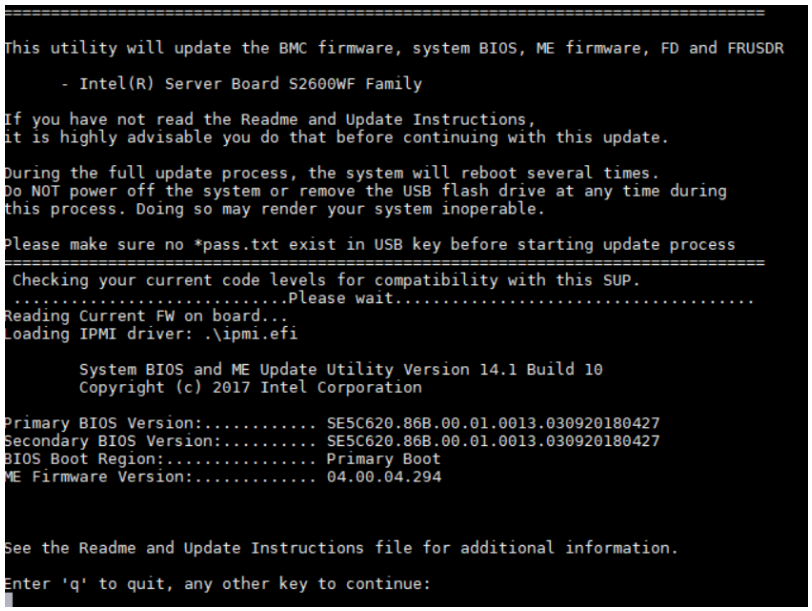
2. Run the ./Stat./Startup.nsh utility:

```
S2600WF_EFI_BIOSR0013_ME04.00.04.294.0_BMC1.43.91f76955_FRUSFR1.43.
```

The BIOS and BMC update wizard guides you through the procedure (refer to Figure 3).

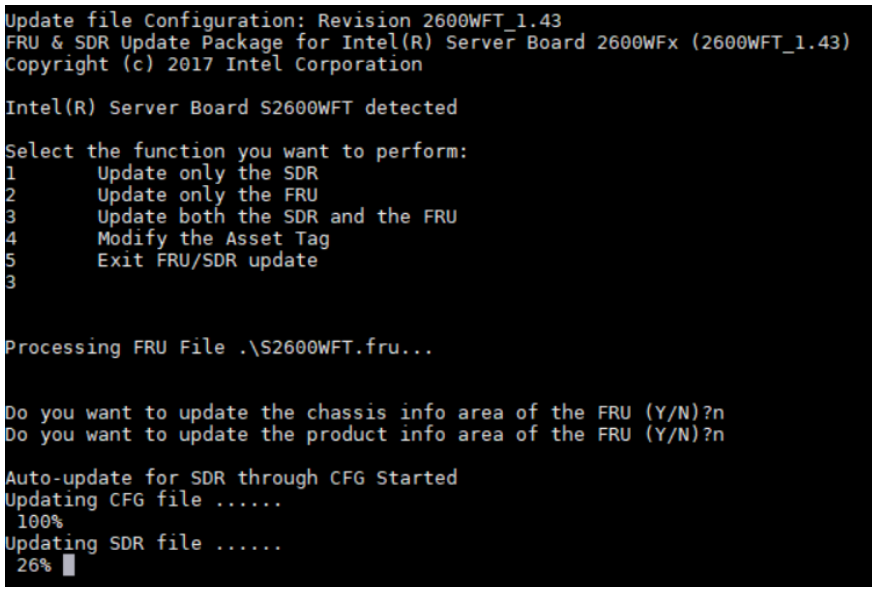
Warning: If you have not read the Readme and Update Instructions, Intel highly recommends you do that before continuing with this update. During this update, the system will reboot several times, **DO NOT** power off the system or remove the USB flash drive at any time during this process, doing so may render the system inoperable.

Figure 3. BIOS and BMC Update Utility



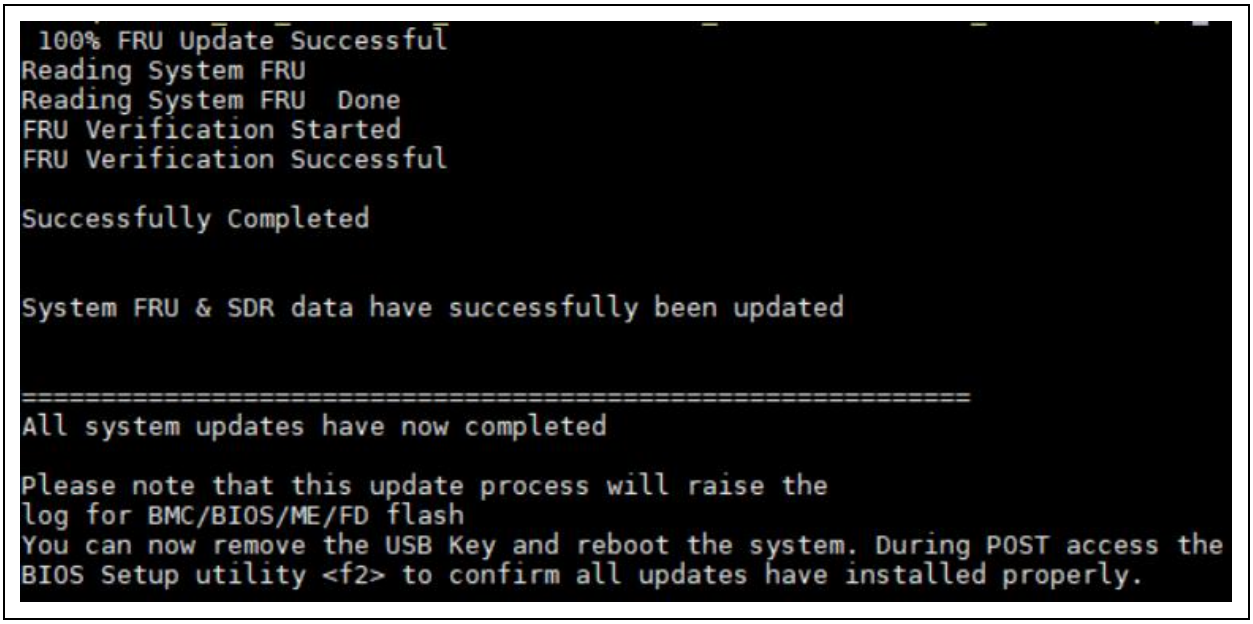
3. Press any key (other than q) to display the next screen.
4. Then press 3 to upgrade the Sensor Data Record (SDR) and Field Replacement Unit (FRU) (refer to Figure 4).

Figure 4. Upgrading SDR and FRU



5. When the update procedure is complete, unplug the USB stick and reboot the server. Refer to Figure 5.

Figure 5. FRU Update Successful Screen



2.4.2 BIOS General Configuration

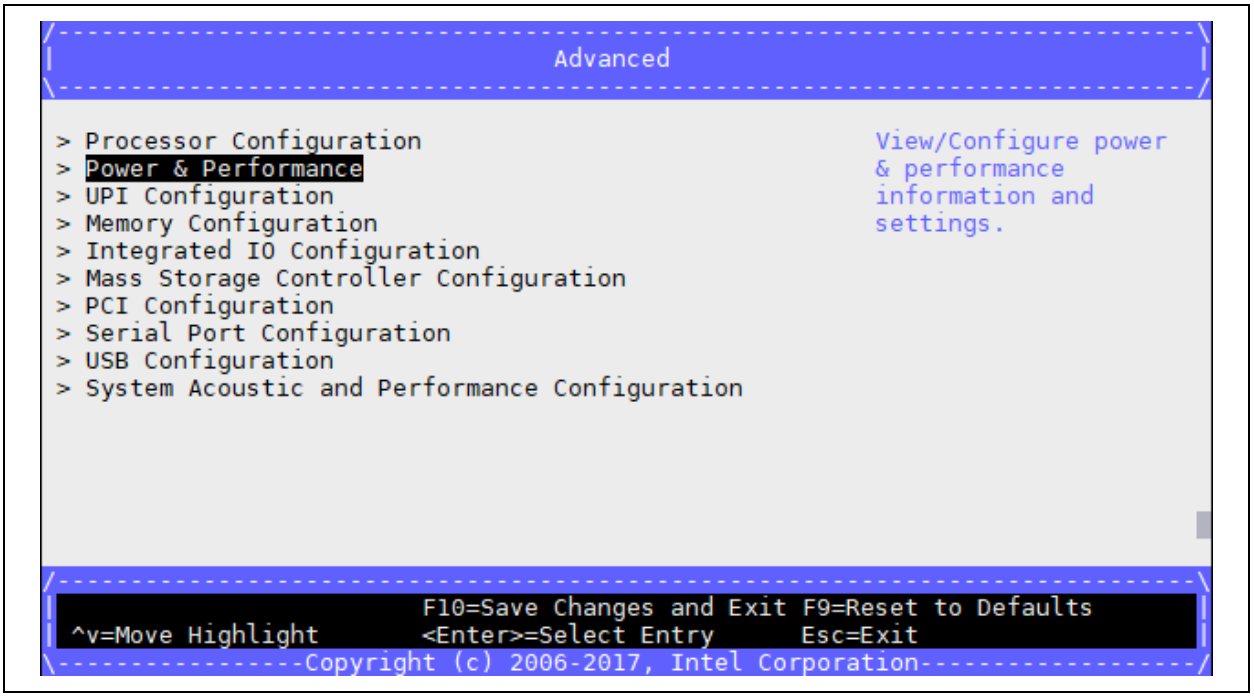
The BIOS general configuration is for general workload other than L1, which uses AVX512, which in turn needs unique Thermal Design Power (TDP) settings for the Skylake platform.

Press **F2** during the server boot phase to enter the BIOS setup. Then, set up the BIOS of the Wolf Pass server.

1. Use load default BIOS setting (F9) to reset BIOS settings to the default.
2. From the Advanced menu, select **Power and Performance** (refer to Figure 6 and Figure 7).

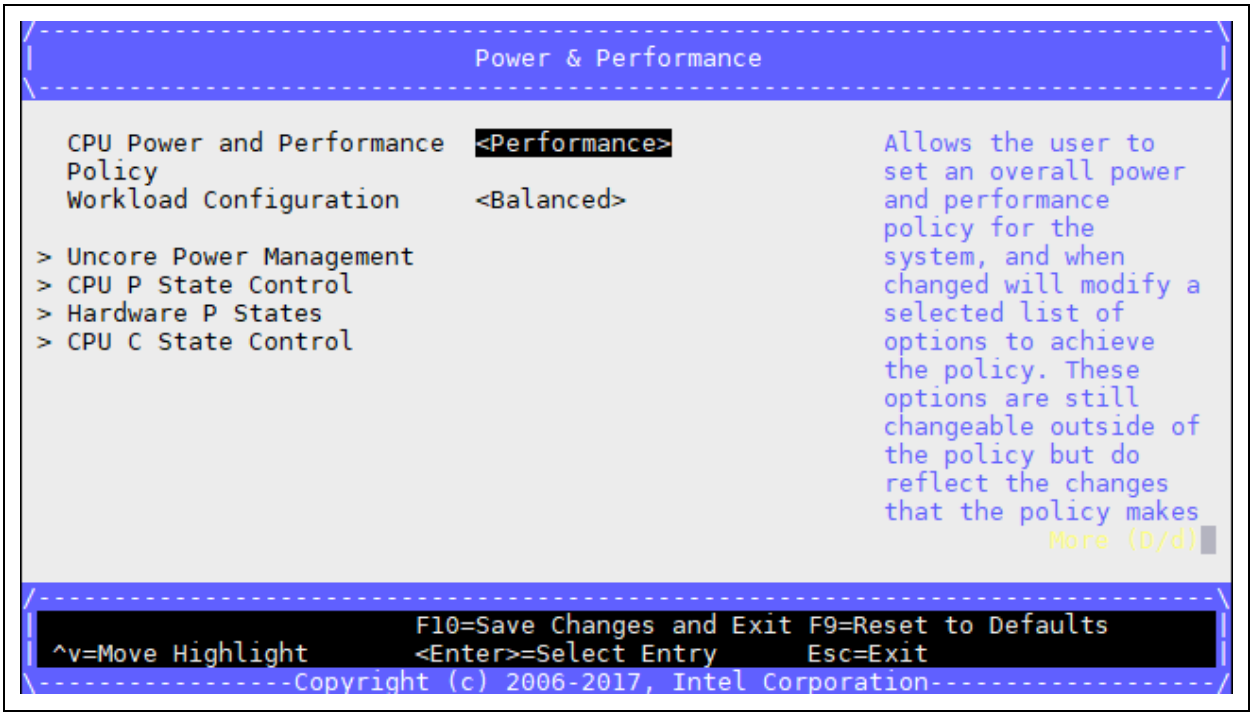
Figure 6. From BIOS configuration to Advanced Settings

Figure 7. Advanced Settings



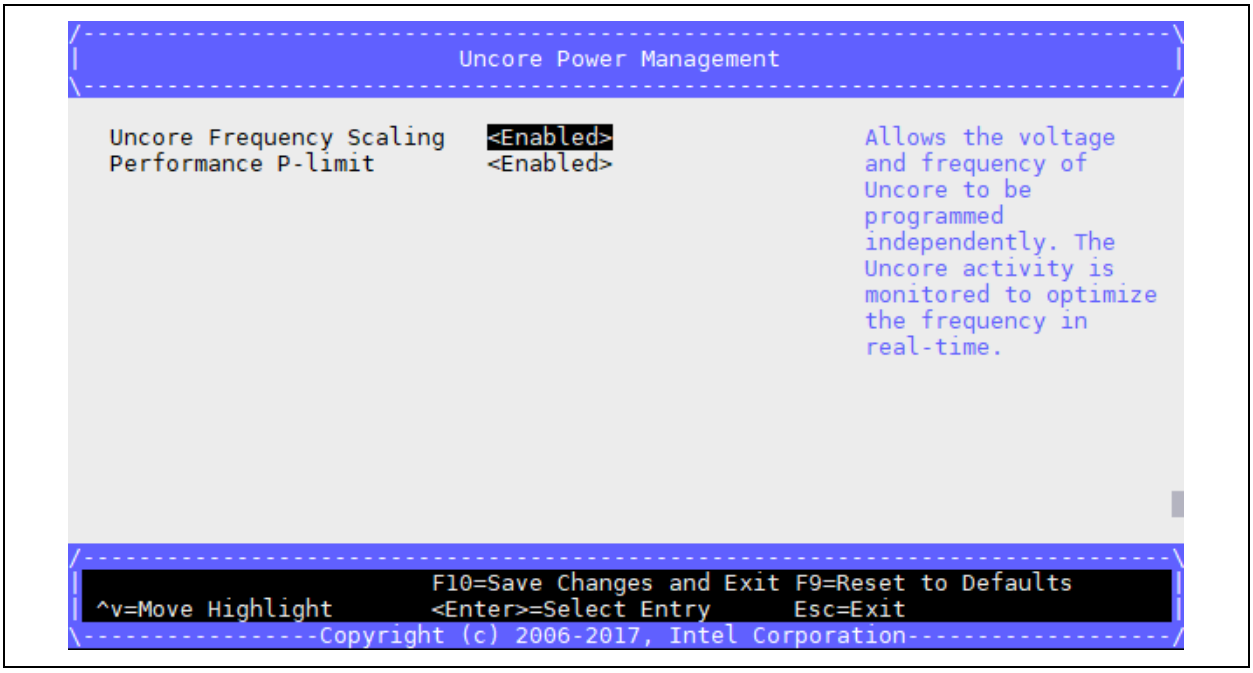
3. Confirm that the BIOS configuration matches Figure 8. through the pathway: Advanced -> Power & Performance -> CPU Power and Performance Policy.

Figure 8. Power and Performance



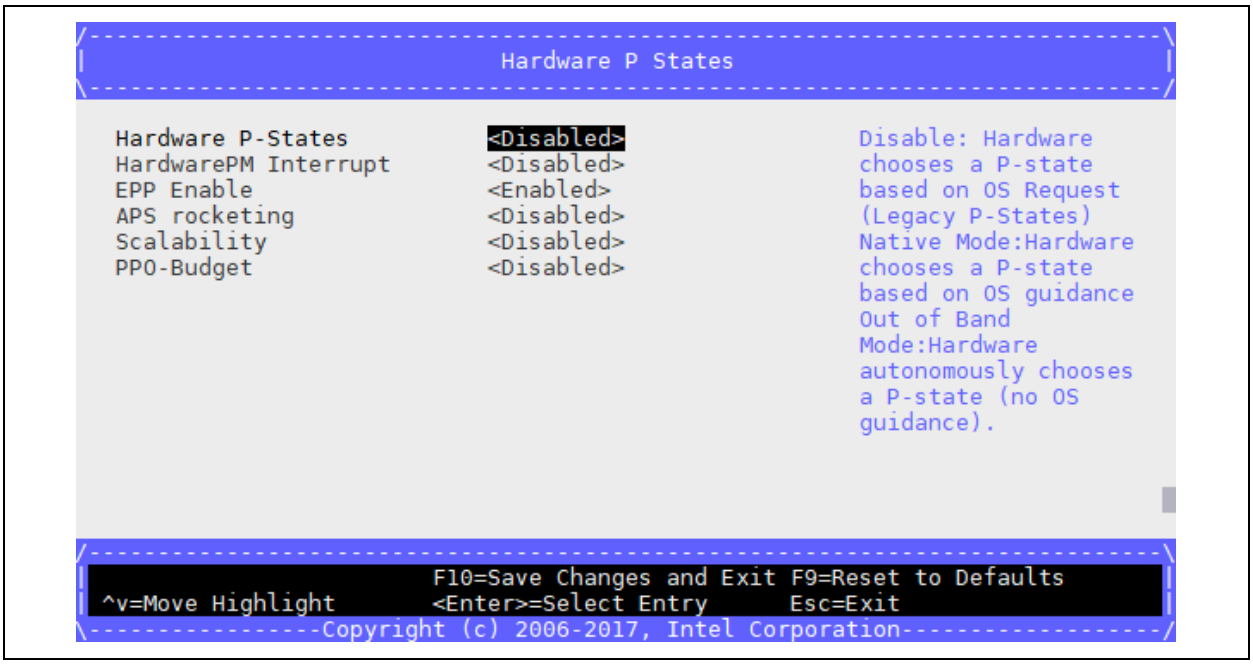
4. Confirm the BIOS configuration matches Figure 9 through the pathway: Advanced -> Power & Performance -> Uncore Power Management.

Figure 9. Uncore Power Management



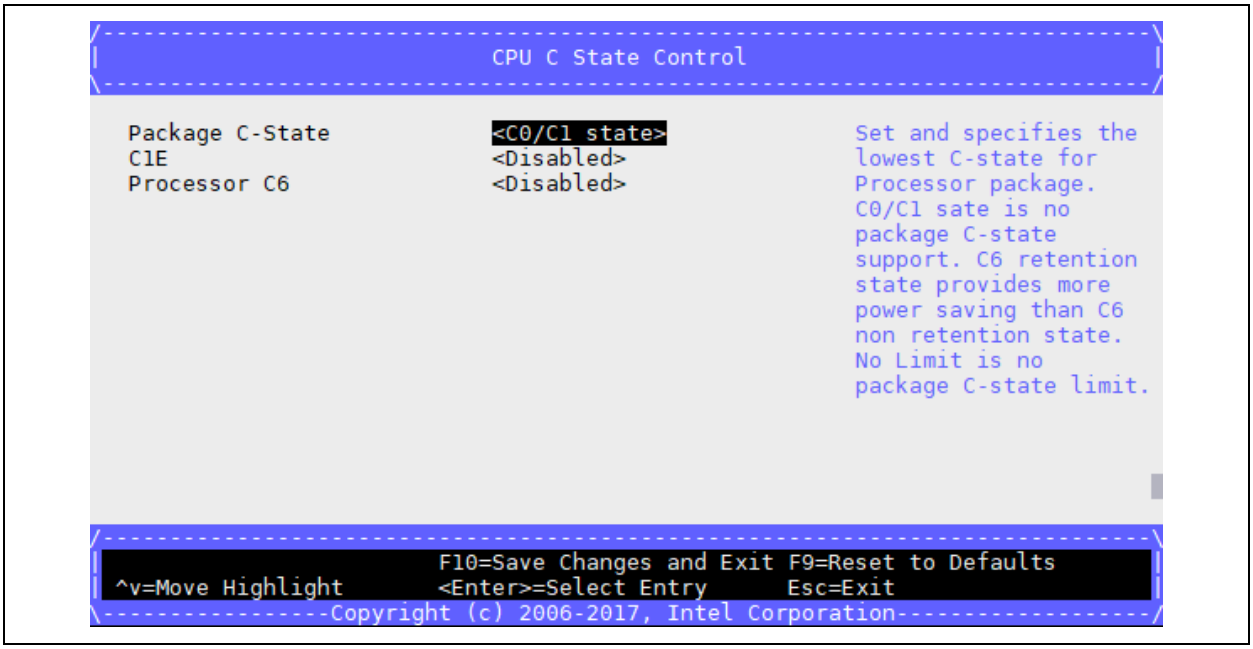
5. Confirm the BIOS configuration matches Figure 10 through the pathway: Advanced -> Power & Performance -> Hardware P States.

Figure 10. Hardware P States



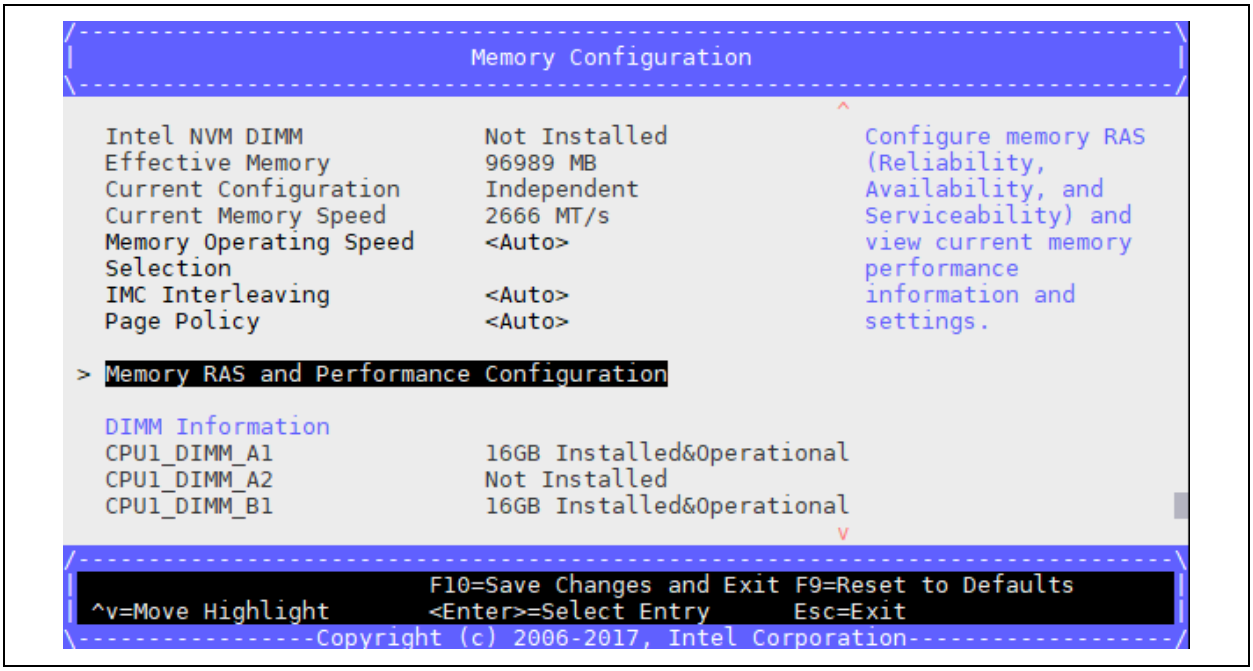
6. Confirm the BIOS configuration matches Figure 11, through the pathway: Advanced -> Power &Performance -> CPU C State Control.

Figure 11. CPU C State Control



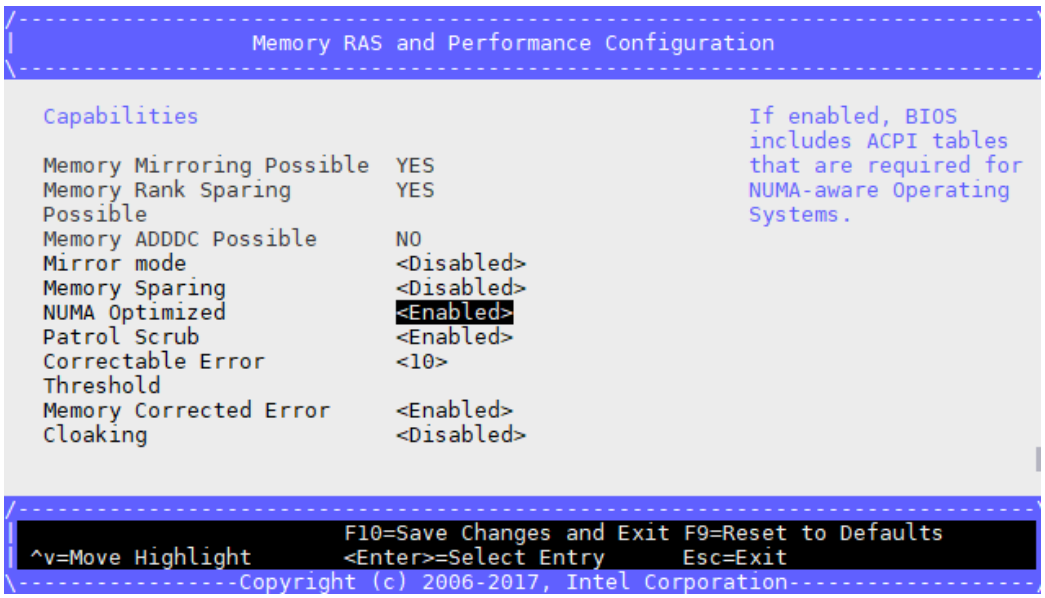
7. Confirm the BIOS configuration matches Figure 12 through the pathway: Advanced -> Power & Performance -> Memory Configuration.

Figure 12. Memory Configuration



8. Confirm the BIOS configuration matches Figure 13 through the pathway: Advanced -> Power & Performance -> Memory Configuration -> Memory RAS and performance Configuration.

Figure 13. Memory RAS and Performance Configuration



9. Confirm the BIOS configuration matches Figure 14 and Figure 15 through the pathway: Advanced -> Processor Configuration.

Note: Reloading the default (see Step 1) may reset the Hammer Test (HT) setting, confirm HT is <Disabled> before quitting BIOS settings.

Figure 14. Processor Configuration

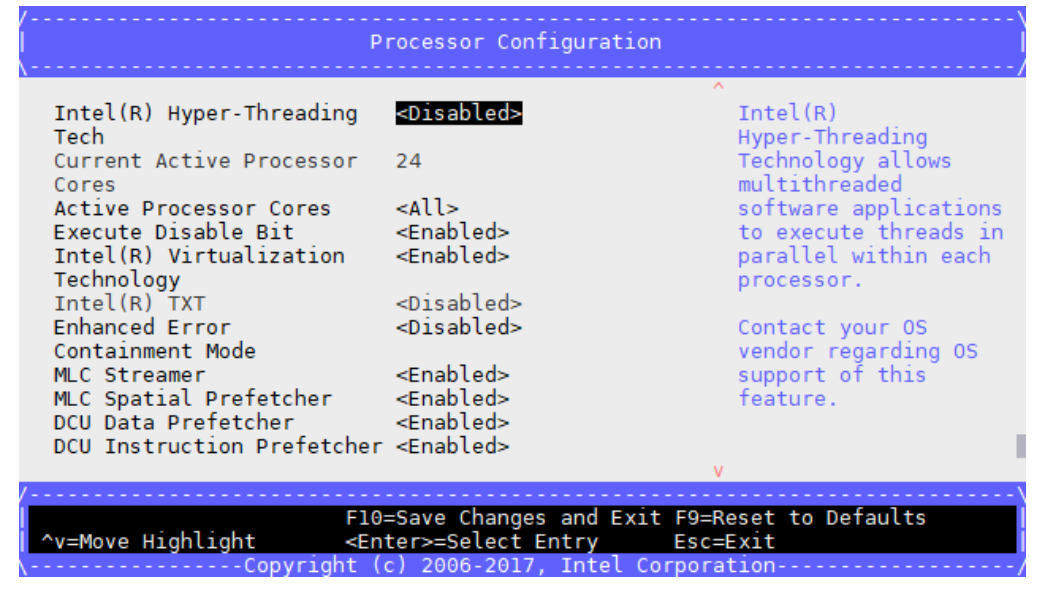
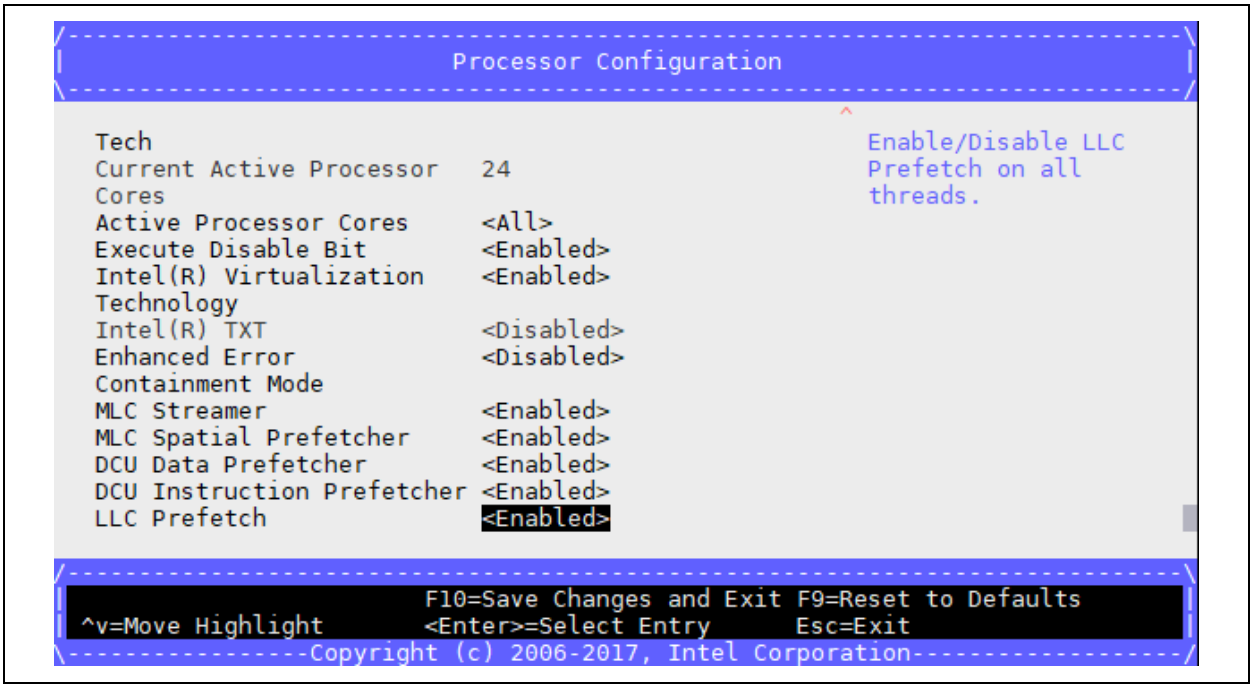
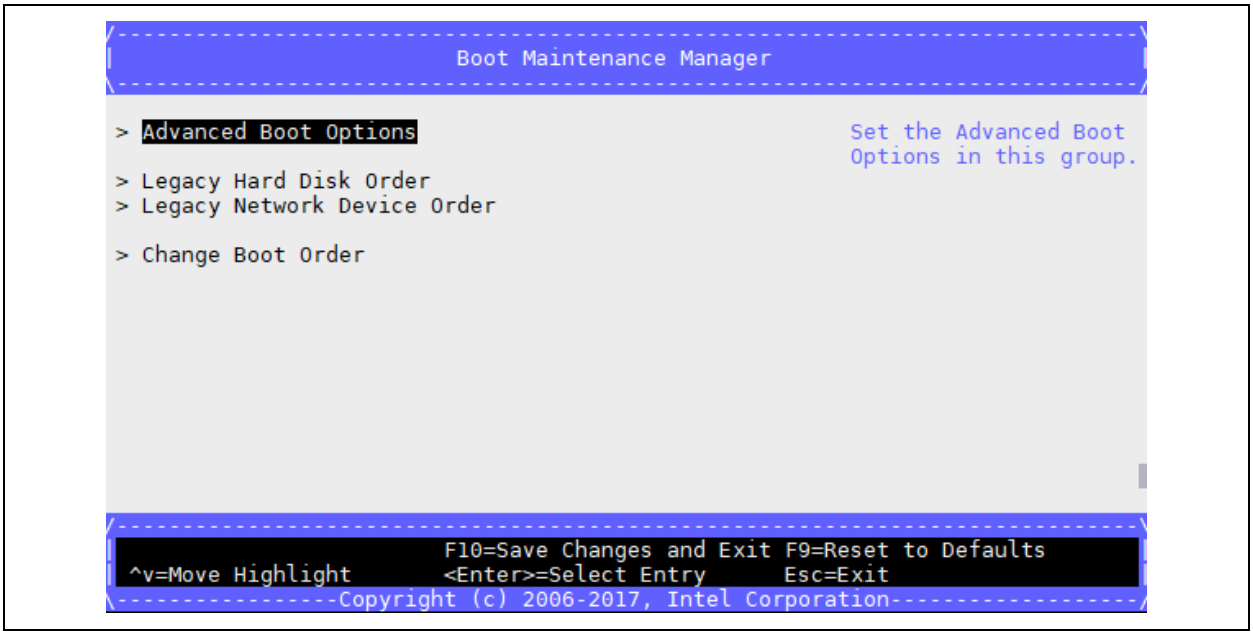


Figure 15. Processor Configuration cont.



10. Confirm that BIOS configuration matches Figure 16 through the pathway: Advanced -> Boot Maintenance Manager.

Figure 16. Boost Maintenance Manager

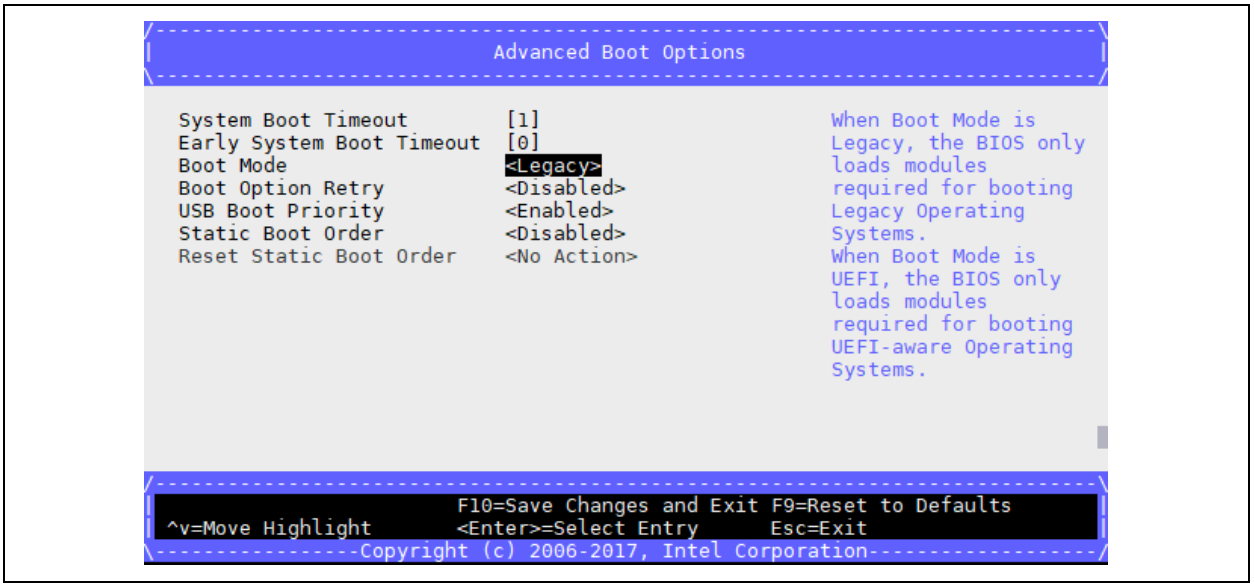


11. Confirm the BIOS configuration matches Figure 17 through the pathway: Advanced -> Advanced Boot Options,

Note: Check the boot order after changing a boot option. Checking the boot order is to make sure it still boots

from the hard drive and not over the PXE, which can overwrite any boot partition on the hard drive.

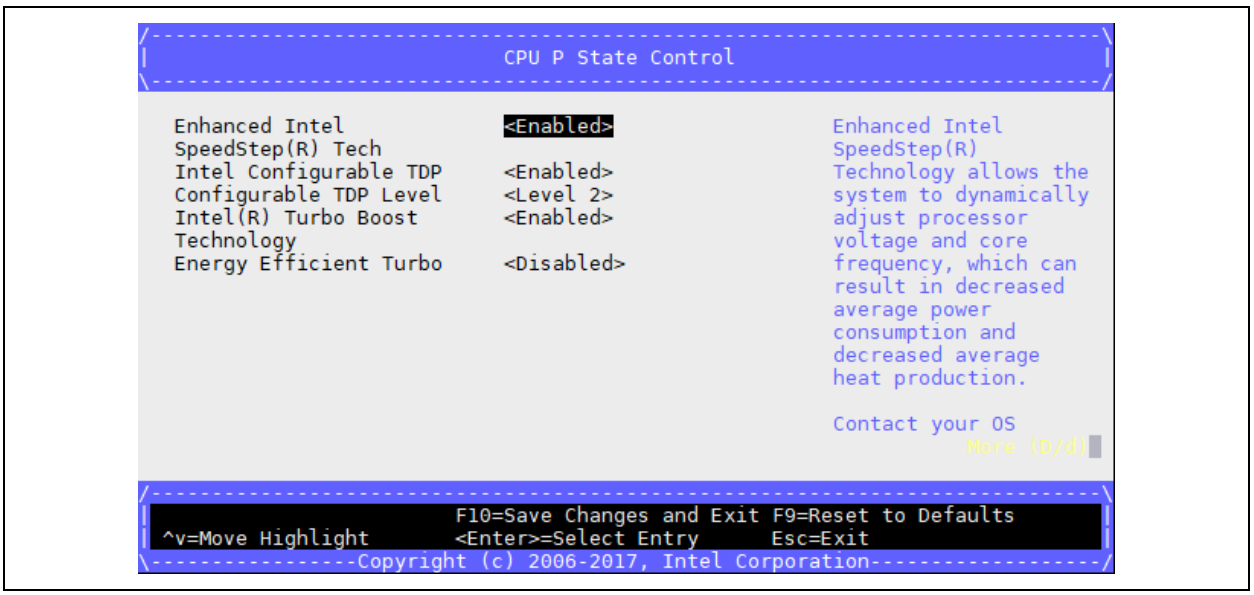
Figure 17. Advanced Boot Options -> Legacy Boot Mode



2.4.2.1 Special BIOS Configuration (Enable TDP)

Enabling TDP is required for the L1 application. The pathway to enable TDP is Advanced -> Power (refer to Figure 18).

Figure 18. CPU P State Control



2.4.3 Real-Time OS Installation and Configuration

Before installing the Real-time OS, it is vital to set up proxies on the server, especially if it's on an isolated network with its proxy servers. Proxies can be placed in /etc/.

2.4.3.1 CentOS* Installation with USB Stick

1. Download Centos7* (CentOS* Linux* release v7.8.2003 (Core)) image to the local folder, for example, /opt/directory.

   http://mirrors.oit.uci.edu/centos/7.8.2003/isos/x86_64/CentOS-7-x86_64-Everything-2003.iso

2. Make an ISO image of the USB stick, using the following command:

```
dd if=/opt/CentOS-7-x86_64-Everything-2003.iso of=/dev/sdb bs=8M
/dev/sdb is the USB flash disk. Check this using the fdisk -l command.
```

3. Insert the USB stick and boot the server from the USB stick.

   Select Install **CentOS7** and press **Enter**.

   a. Select the **Installation Process Language** and click **Continue**.

   b. Configure the Date & Time, Language support, Installation Source (Local media), and Installation Destination (in Installation Summary according to user's preference).

   c. Begin installing and set the password for the root user,

Note: Create an additional user account if required.

   d. Once installed, unplug the USB stick and reboot.

      After rebooting, log in using the username and password for the root or user account created during the install procedure.

2.4.3.2 Real-Time Packages Installation

Create the file CentOS-rt.repo in /etc/yum.repo.d and add the example text from this section (base-os is not needed as it is already taken care of in CentOS-Base/repo).

The required packages can be downloaded from the CentOS* archive repos, as shown in Section 2.2, Software Configuration.

- Use the rpm –ivh <package list> to install downloaded all packages from the localhost.
- Alternatively, to install packages online - use yum to install the –y <package list>.

Note: Using yum to install the packages online requires an additional step to configure yum repos.

Users can also download the RT repo from http://linuxsoft.cern.ch/cern/centos/7/rt/CentOS-RT.repo

Follow the example to set up the RT repo:

```
#
#
#CERN CentOS 7 RealTime repository at http://linuxsoft.cern.ch/
#

[rt]
name=CentOS-$releasever - RealTime
baseurl=http://linuxsoft.cern.ch/cern/centos/$releasever/rt/$basearch/
```

```
gpgcheck=1
enabled=1
protect=1
priority=10
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-cern

[rt-debug]
name=CentOS-$releasever - RealTime - Debuginfo
baseurl=http://linuxsoft.cern.ch/cern/centos/$releasever/rt/Debug/$basearch/
gpgcheck=1
enabled=0
protect=1
priority=10
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-cern

[rt-source]
name=CentOS-$releasever - RealTime - Sources
baseurl=http://linuxsoft.cern.ch/cern/centos/$releasever/rt/Sources/
gpgcheck=1
enabled=0
protect=1
priority=10
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-cern

[rt-testing]
name=CentOS-$releasever - RealTime Testing
baseurl=http://linuxsoft.cern.ch/cern/centos/$releasever/rt-testing/$basearch/
gpgcheck=1
enabled=0
protect=1
priority=10
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-cern

[rt-testing-debug]
name=CentOS-$releasever - RealTime Testing - Debuginfo
baseurl=http://linuxsoft.cern.ch/cern/centos/$releasever/rt-testing/Debug/$basearch/
gpgcheck=1
enabled=0
protect=1
priority=10
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-cern

[rt-testing-source]
name=CentOS-$releasever - RealTime Testing - Sources
baseurl=http://linuxsoft.cern.ch/cern/centos/$releasever/rt-testing/Sources/
gpgcheck=1
enabled=0
protect=1
priority=10
gpgkey=file:///etc/pki/rpm-gpg/RPM-GPG-KEY-cern
```

If you have a proxy server to access external network, yum requires proxy settings to be set. The following lines are an example and need to be added to /etc/yum.conf:

```
proxy=https://[Enter the proxy information here]
http_proxy=http://[Enter the proxy information here]
```

Similarly, if wget also requires proxy setting, then the following lines must be added to wgetrc in /etc. (You may need to create wgetrc if it is not present.)

```
proxy= http://[Enter the proxy information here]
http_proxy= http://[Enter the proxy information here]
```

The following example shows the yum install command. By default, the latest version of packages and tools will be installed (or specify the version along with the package name to install a specific version).

```
yum install -y vim gcc-c++ libhugetlbfs* libstdc++* kernel-devel numa* gcc git mlocate cmake wget
ncurses-devel hmaccalc zlib-devel binutils-devel elfutils-libelf-devel numactl-devel
libhugetlbfs-devel bc
yum groupinstall "Development Tools"
```

If the command doesn't work, try running:

```
yum clean all
yum grouplist | grep Development
yum groups mark install "Development Tools"
yum groups mark convert "Development Tools"
yum groupinstall "Development Tools"
```

Note: If this doesn't work, run yum update, then reboot system (Will only encounter issues when compiling DPDK)

If there is an error regarding a 'tuned' package version, do the following:

```
yum remove tuned.noarch
wget ftp://ftp.icm.edu.pl/vol/rzm3/linux-slc/centos/7.5.1804/os/x86_64/Packages/tuned-2.9.0-1.el7.noarch.rpm
rpm -ivh tuned-2.9.0-1.el7.noarch.rpm
yum install -y kernel-rt kernel-rt-devel kernel-rt-kvm rtctl rt-setup rt-tests tuna
yum install -y tuned-profiles-nfv tuned-profiles-nfv-host tuned-profiles-nfv-guest qemu-kvm-tools-ev net-tools
```

Use the above yum commands to install the latest versions of packages and tools or refer to the commands in the table below to assign specific versions using yum.

Table 7. Yum commands for Packages and Tools

Command Name
yum install -y kernel-rt-3.10.0-1127.19.1.rt56.1116.el7.x86_64
yum install -y kernel-rt-devel-3.10.0-1127.19.1.rt56.1116.el7.x86_64.rpm
yum install –y kernel-rt-kvm-3.10.0-1127.19.1.rt56.1116.el7.x86_64.rpm
yum install –y rtctl-1.13-2.el7.noarch
yum install –y rt-setup-2.0-9.el7.x86_64
yum install –y rt-tests-1.0-16.el7.x86_64
yum install –y libcgroup-0.41-13.el7.x86_64
yum install –y python-ethtool-0.8-8.el7.x86_64
yum install –y qemu-kvm-tools-ev-2.9.0-16.el7_4.14.1.x86_64
yum install –y tuna-0.13-9.el7.noarch
yum install –y tuned-2.9.0-1.el7fdp.noarch
yum install –y tuned-profiles-nfv-guest-2.9.0-1.el7_5.2.noarch
yum install –y tuned-profiles-nfv-host-2.9.0-1.el7_5.2.noarch
yum install –y libhugetlbfs-devel-2.16-12.el7.i686
yum install –y libhugetlbfs-2.16-12.el7.x86_64
yum install –y libhugetlbfs-devel-2.16-12.el7.x86_64
yum install –y libhugetlbfs-utils-2.16-12.el7.x86_64
yum install –y libstdc++-4.8.5-16.el7.x86_64
yum install –y libstdc++-devel-4.8.5-16.el7.x86_64
yum install –y kernel-rt-devel-3.10.0-1062.12.1.rt56.1042.el7.x86_64
yum install –y numactl-devel-2.0.9-6.el7_2.x86_64
yum install –y numactl-2.0.9-6.el7_2.x86_64
yum install -y gcc-c++ 4.8.5-28.el7_5.1

Table 8. Yum RPM Names

RPM Name
rpm –ivh kernel-rt-3.10.0-1127.19.1.rt56.1116.el7.x86_64.rpm
rpm –ivh kernel-rt-kvm-3.10.0-1127.19.1.rt56.1116.el7.x86_64.rpm
rpm –ivh rtctl-1.13-2.el7.noarch.rpm
rpm –ivh rt-setup-2.0-9.el7.x86_64.rpm
rpm –ivh rt-tests-1.0-16.el7.x86_64.rpm
rpm –ivh libcgroup-0.41-13.el7.x86_64.rpm
rpm –ivh python-ethtool-0.8-8.el7.x86_64.rpm
rpm –ivh qemu-kvm-tools-ev-2.9.0-16.el7_4.14.1.x86_64.rpm
rpm –ivh tuna-0.13-9.el7.noarch.rpm
rpm –ivh tuned-2.9.0-1.el7fdp.noarch.rpm
rpm –ivh tuned-profiles-nfv-guest-2.9.0-1.el7_5.2.noarch.rpm
rpm –ivh tuned-profiles-nfv-host-2.9.0-1.el7_5.2.noarch.rpm
rpm –ivh libhugetlbfs-devel-2.16-12.el7.i686.rpm
rpm –ivh libhugetlbfs-2.16-12.el7.x86_64.rpm
rpm –ivh libhugetlbfs-devel-2.16-12.el7.x86_64.rpm
rpm –ivh libhugetlbfs-utils-2.16-12.el7.x86_64.rpm
rpm –ivh libstdc++-4.8.5-16.el7.x86_64.rpm
rpm –ivh libstdc++-devel-4.8.5-16.el7.x86_64.rpm
rpm –ivh kernel-rt-devel-3.10.0-1127.19.1.rt56.1116.el7.x86_64.rpm
rpm –ivh numactl-devel-2.0.9-6.el7_2.x86_64.rpm
rpm –ivh numactl-2.0.9-6.el7_2.x86_64.rpm
rpm –ivh gcc-c++ 4.8.5-28.el7_5.1.rpm

2.4.3.3 Configuration Example for Wolf Pass (Intel® Xeon® Gold 6148 CPU @ 2.4GHz)

1. Isolate the cores for Real-Time Tasks:

```
lscpu|grep NUMA
NUMA node(s):          2
NUMA node0 CPU(s):     0-19
NUMA node1 CPU(s):     20-39
```

   It shows there are two physical sockets, each has 20 cores, with ids numbered as 0-19 and 20-39 separately.

2. Edit /etc/tuned/realtime-virtual-host-variables.conf to add isolated_cores=1-39:

```
# Examples:
# isolated_cores=2,4-7
# isolated_cores=2-23
isolated_cores=1-19, 21-39
```

   Core 1-39 are isolated from the host OS and dedicated for Real-Time tasks.

3. To activate Real-Time Profile, run command:

```
tuned-adm profile realtime-virtual-host
```

4. Then check the Wolf Pass server:

```
grep tuned_params= /boot/grub2/grub.cfg
set tuned_params="isolcpus=1-39 intel_pstate=disable nosoftlockup skew_tick=1 nohz=on
nohz_full=1-19, 21-39 rcu_nocbs=1-19, 21-39"
```

5. Configure kernel command line.

   Example for Wolf Pass:

   a. Edit /etc/default/grub and append the following to the GRUB_CMDLINE_LINUX:

```
"processor.max_cstate=1 intel_idle.max_cstate=0 intel_pstate=disable idle=poll
default_hugepagesz=1G hugepagesz=1G hugepages=16 intel_iommu=on iommu=pt selinux=0
enforcing=0 nmi_watchdog=0 audit=0 mce=off kthread_cpus=0 irqaffinity=0 idle=poll"
```

   Above method is used for legacy mode, for UEFI mode, please edit /boot/efi/EFI/centos/grub.cfg and make change if anything different.

   b. Add the following:

```
GRUB_CMDLINE_LINUX_DEFAULT="${GRUB_CMDLINE_LINUX_DEFAULT:+$GRUB_CMDLINE_LINUX_DEFAULT
}\$tuned_params"

GRUB_INITRD_OVERLAY="${GRUB_INITRD_OVERLAY:+$GRUB_INITRD_OVERLAY }\$tuned_initrd"
```

   c. For example:

```
GRUB_TIMEOUT=5

GRUB_DISTRIBUTOR="$(sed 's, release .*$,,g' /etc/system-release)"

GRUB_DEFAULT=saved

GRUB_DISABLE_SUBMENU=true

GRUB_TERMINAL_OUTPUT="console"

GRUB_CMDLINE_LINUX="crashkernel=auto rhgb quiet"

GRUB_DISABLE_RECOVERY="true"

GRUB_CMDLINE_LINUX_DEFAULT="${GRUB_CMDLINE_LINUX_DEFAULT:+$GRUB_CMDLINE_LINUX_DEFAULT
}\$tuned_params"

GRUB_INITRD_OVERLAY="${GRUB_INITRD_OVERLAY:+$GRUB_INITRD_OVERLAY }\$tuned_initrd"

GRUB_CMDLINE_LINUX=" crashkernel=auto rd.lvm.lv=centos/root rd.lvm.lv=centos/swap rhgb
quiet processor.max_cstate=1 intel_idle.max_cstate=0 intel_pstate=disable idle=poll
default_hugepagesz=1G hugepagesz=1G hugepages=16 intel_iommu=on selinux=0 enforcing=0
nmi_watchdog=0 audit=0 mce=off kthread_cpus=0 irqaffinity=0 console=tty0
console=ttyS0,115200n8"
```

   d. After the change, the grub file runs the following command to update the grub:

```
grub2-mkconfig -o /boot/grub2/grub.cfg
```

   e. Reboot the server, and check the kernel parameter, which should look like:

```
cat /proc/cmdline

BOOT_IMAGE=/vmlinuz-3.10.0-1062.12.1.rt56.1042.el7.x86_64 root=UUID=9b3e69f6-88af-4af1-
8964-238879b4f282 ro crashkernel=auto rd.lvm.lv=centos/root rd.lvm.lv=centos/swap rhgb
quiet processor.max_cstate=1 intel_idle.max_cstate=0 intel_pstate=disable idle=poll
default_hugepagesz=1G hugepagesz=1G hugepages=16 intel_iommu=on selinux=0 enforcing=0
nmi_watchdog=0 audit=0 mce=off kthread_cpus=0 irqaffinity=0 console=tty0
```

```
console=ttyS0,115200n8 skew_tick=1 isolcpus=1-39 intel_pstate=disable nosoftlockup
nohz=on nohz_full=1-39 rcu_nocbs=1-39
```

f. Set CPU frequency using msr-tools

```
git clone https://github.com/intel/msr-tools/
cd msr-tools/
git checkout msr-tools-1.3
make
cat <<EOF > turbo-2.2G.sh
#!/bin/sh

for i in {0..39}
do
#Set core 0-39 to 2.2GHz (0x1600). Please change according to your CPU model
    ./wrmsr -p \${i} 0x199 0x1600
done

#Set Uncore to Max
./wrmsr -p 0 0x620 0x1e1e
./wrmsr -p 39 0x620 0x1e1e
EOF
chmod 755 turbo-2.2G.sh
sh turbo-2.2G.sh
```

g. Set CPU Frequency Policy to Performance.

Use the command below to set:

```
 "cpupower frequency-set -g performance"
```

2.5 RT Test and Verify

Wolf Pass server Real-Time test result:

```
cyclictest -m -n -p95 -d0 -a 1-16 -t 16
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 0.00 0.01 0.05 1/702 25564

T: 0 (25549) P:95 I:1000 C:  5796 Min:   4 Act:    5 Avg:   4 Max:   6
T: 1 (25550) P:95 I:1000 C:  5797 Min:   4 Act:    5 Avg:   4 Max:   6
T: 2 (25551) P:95 I:1000 C:  5791 Min:   4 Act:    5 Avg:   4 Max:   6
T: 3 (25552) P:95 I:1000 C:  5788 Min:   4 Act:    4 Avg:   4 Max:   6
T: 4 (25553) P:95 I:1000 C:  5785 Min:   4 Act:    4 Avg:   4 Max:   6
T: 5 (25554) P:95 I:1000 C:  5782 Min:   4 Act:    5 Avg:   4 Max:   6
T: 6 (25555) P:95 I:1000 C:  5778 Min:   4 Act:    5 Avg:   4 Max:   6
T: 7 (25556) P:95 I:1000 C:  5775 Min:   4 Act:    5 Avg:   4 Max:   6
T: 8 (25557) P:95 I:1000 C:  5772 Min:   4 Act:    5 Avg:   4 Max:   6
T: 9 (25558) P:95 I:1000 C:  5768 Min:   4 Act:    5 Avg:   4 Max:   6
```

```
T:10 (25559) P:95 I:1000 C:  5765 Min:    4 Act:    5 Avg:    4 Max:    6
T:11 (25560) P:95 I:1000 C:  5762 Min:    4 Act:    5 Avg:    4 Max:    6
T:12 (25561) P:95 I:1000 C:  5758 Min:    5 Act:    5 Avg:    5 Max:    5
T:13 (25562) P:95 I:1000 C:  5758 Min:    4 Act:    5 Avg:    4 Max:    5
T:14 (25563) P:95 I:1000 C:  5758 Min:    4 Act:    5 Avg:    4 Max:    5
T:15 (25564) P:95 I:1000 C:  5758 Min:    4 Act:    5 Avg:    4 Max:    5
```

Note: The `-D` parameter specifies the duration the test will last. Set the `-D` parameter test duration to a minimum of 12 hours, and 24 hours + for rigid performance validation. For quick performance validation, 15 minutes is recommended.

Note: Pay attention to the Avg. and Max. on a well-tuned platform, the numbers should be similar.

§

3.0 Installation Guide for Kubernetes*

Kubernetes* (k8s*) is an open-source container orchestration system for automating application deployment, scaling, and management of application containers across clusters of hosts. k8s* works with a range of container tools, including Docker*. Many cloud services offer a Kubernetes-based platform or infrastructure as a service (PaaS or IaaS) on which k8s* can be deployed as a platform-providing service.

There are several methods to install Kubernetes and even it's plugins: kubeadm, kubspray or CIR. Following sub-charpts will give the introduction of kubeadm (chapters 3.2 through chapters 0) and CIR.

3.1 Hardware Platforms

Note: Kubernetes Docker* was tested on dual-socket Broadwell and Skylake SP platforms.

Broadwell – EP (Wildcat Pass)	Skylake-SP (Wolf Pass)
Architecture: x86_64	Architecture: x86_64
CPU op-modes: 32-bit, 64-bit	CPU op-modes: 32-bit, 64-bit
Byte Order: Little-endian	Byte Order: Little-endian
CPUs: 44	CPUs: 40
Online CPUs list: 0-43	Online CPUs list: 0-39
Threads per core: 1	Threads per core: 1
Cores per socket: 22	Cores per socket: 20
Sockets: 2	Sockets: 2
NUMA nodes: 2	NUMA nodes: 2
Vendor ID: Genuine Intel	Vendor ID: Genuine Intel
CPU family: 6	CPU family: 6
Model: 79	Model: 85
Model name: Intel® Xeon® CPU E5-2699 v4 @ 2.20 GHz	Model name: Intel® Xeon® Gold 6148 CPU @ 2.40 GHz
Stepping: 1	Stepping: 4
CPU MHz: 1703.453	CPU MHz: 1600.000
BogoMIPS: 4395.89	BogoMIPS: 3204.84
Virtualization: VT-x	Virtualization: VT-x
L1d cache: 32 K	L1d cache: 32 K
L1i cache: 32 K	L1i cache: 32 K
L2 cache: 256 K	L2 cache: 1024 K
L3 cache: 56320 K	L3 cache: 28160 K
NUMA node0 CPUs: 0-21	NUMA node0 CPUs: 0-19
NUMA node1 CPUs: 22-43	NUMA node1 CPUs: 20-39

Broadwell – EP (Wildcat Pass)	Skylake-SP (Wolf Pass)
Kernel: kernel-rt-3.10.0-1127.19.1.rt56.1116.el7.x86_64 OS Image: CentOS* Linux release 7.8.2003 (Core) OS: Linux* Architecture: Intel Container Runtime Version: Client: Docker* Engine – Community - 19.03.3 Server: Docker Engine – Community - 18.09.9 Kubelet* Version: v1.18.6 Kube*-Proxy Version: v1.18.6	Kernel: kernel-rt-3.10.0-1127.19.1.rt56.1116.el7.x86_64 OS Image: CentOS* Linux release 7.8.2003 (Core) OS: Linux* Architecture: Intel Container Runtime Version: Client: Docker* Engine – Community - 19.03.3 Server: Docker Engine – Community - 18.09.9 Kubelet* Version: v1.18.6 Kube*-Proxy Version: v1.18.6

3.1.1 Configure the Operating System and Add Hosts

K8S* must be run with SELinux disabled and swap off enabled on all nodes. After SELinux is disabled, you can still use the sestatus command to check the status of SELinux and make sure it is actually disabled.

1. To set SELinux to disabled, add this parameter to the command line:

```
setenforce 0
```

2. To enable swap off, run this command:

```
swapoff -a
```

3. Add known hosts to /etc/hosts accordingly as per host – example for configuration on a master node.

```
  127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
  ::1         localhost localhost.localdomain localhost6 localhost6.localdomain6
       192.168.0.100 k8s-master
       192.168.0.101 k8s-worker1
```

Note: All IP addresses in this guide are examples. Use the actual IP addresses associated with the nodes in your installation.

3.2 Install Docker*

To install Docker, run the following commands on all nodes:

```
# Install Docker CE
## Set up the repository
### Install required packages.
yum install yum-utils device-mapper-persistent-data lvm2

### Add Docker repository.
yum-config-manager \
  --add-repo \
  https://download.docker.com/linux/centos/docker-ce.repo

## Install Docker CE.
yum install docker-ce-19.03.12 -y

## Create /etc/docker directory.
mkdir /etc/docker
```

```
# Setup daemon.
cat > /etc/docker/daemon.json <<EOF
{
  "exec-opts": ["native.cgroupdriver=systemd"],
  "log-driver": "json-file",
  "log-opts": {
    "max-size": "100m"
  },
  "storage-driver": "overlay2",
  "storage-opts": [
    "overlay2.override_kernel_check=true"
  ]
}
EOF

mkdir -p /etc/systemd/system/docker.service.d

# Restart Docker
systemctl enable docker
systemctl daemon-reload
systemctl restart docker
```

3.3 Install Kubernetes - kubeadm*

To install Kubernetes, run the following on all nodes:

```
cat <<EOF > /etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=1
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
EOF

# Set SELinux in permissive mode (effectively disabling it)
setenforce 0
sed -i 's/^SELINUX=enforcing$/SELINUX=permissive/' /etc/selinux/config

yum install -y kubelet-1.18.6 kubeadm-1.18.6 kubectl-1.18.6 --disableexcludes=kubernetes

systemctl enable --now kubelet
systemctl start kubelet

cat <<EOF >  /etc/sysctl.d/k8s.conf
net.bridge.bridge-nf-call-ip6tables = 1
net.bridge.bridge-nf-call-iptables = 1
EOF
```

```
sysctl --system
```

3.4 Configure Kubernetes* and Docker* to Run with Proxy

Intel recommends the proxy related to environmental variables be added to /root/.bashrc file in the Note.

Note: This section is optional. It is intended for environments with a proxy server (identified with <proxy-url>).

1. Add the IP address of the master node to the no proxy environment variable, as follows:

   ```
   export no_proxy=localhost,127.0.0.1,192.168.0.100
   ```

   Where 192.168.0.100 = IP address of the host.

2. Confirm NO_PROXY is set in the relevant files across all configured nodes (refer to Sections 3.4.1, Configure Docker and 3.4.2, Configure Kubernetes).

3.4.1 Configure Docker

1. To add a proxy to Docker:

   ```
   mkdir -p /etc/systemd/system/docker.service.d
   vi /etc/systemd/system/docker.service.d/http-proxy.conf
   ```

2. Add the following to /etc/systemd/system/docker.service.d/http-proxy.conf:

   ```
   [Service]
   Environment="HTTP_PROXY=<proxy-url>"
   ```

3. Run these commands:

   ```
   sudo systemctl daemon-reload
   sudo systemctl restart docker
   ```

3.4.2 Configure Kubernetes

To configure K8s to work with a proxy, on all nodes, add the following configuration in the [Service] tab in /etc/systemd/system/kubelet.service:

```
Environment=HTTP_PROXY=<proxy-url>
Environment=NO_PROXY=192.168.0.100,localhost
```

Where 192.168.0.100 = IP address of the host.

3.5 Kubernetes Initialization

The following sections describe how to initialize K8s on the master node and other nodes.

Note: Instructions in each section assume that all steps in the previous sections have been followed.

3.5.1 Kubernetes Initialization on Master

1. To initialize the K8s on the master node, run the following commands:

   ```
   kubeadm init --kubernetes-version=v1.18.6 --pod-network-cidr=10.244.0.0/16 --apiserver-
   advertise-address=192.168.0.100 --token-ttl 0
   ```

   The output looks like:

```
Your Kubernetes control-plane has initialized successfully!
To start using your cluster, you need to run the following as a regular user:

mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join <master-ip>:<master-port> --token <token> \
--discovery-token-ca-cert-hash sha256:<hash>
```

And then follow the hint provided in above result to start using cluster.

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

To make kubectl work for your non-root user, run these commands, which are also part of above output.

Note: If the yum update command fails with a dependency error, run the command:

```
yum -skip-broken update
```

Note: Use the IP address 10.244.0.0 for the POD network.

Note: The generated token (kubeadm join) is required to connect from non-master nodes to the master node.

Note: In a case where the K8s cluster must be re-initialized, follow the procedure in Section 3.5.1.1, Reinitializing a Kubernetes Cluster, and then repeat the steps in this section, starting at Step 1.

2. Run the following command (assuming it is run from root, otherwise follow the instructions provided by kubeadm init):

```
export KUBECONFIG=/etc/Kubernetes/admin.conf
```

3. Before initialization is finished, the container network (calico or flennel) must be set up. please refer to below chapter for enabling container network.

4. To check if the master node is ready, run:

```
kubectl get nodes
```

5. To see if all system PODs are up and running:

```
kubectl get pods --all-namespaces
```

Note: Step 6 is needed only in a proxy environment.

```
Look for the Calico network IP address it is trying to connect to (for example, 10.244.0.12)
then add this address to configuration in the [Service] tab in
/etc/systemd/system/kubelet.service.
Environment=HTTP_PROXY=<proxy-url> NO_PROXY=192.168.0.100,10.102.248.16,localhost,10.244.0.12
```

6. Run: (if and only if the kubeadm init process completed, kubelet can be activated ultimately)

```
systemctl restart kubelet
systemctl status kubelet
```

3.5.1.1 Reinitializing a Kubernetes Cluster

1. Set swap off and disable SELinux (refer to Section 3.1.1, Configure the Operating System and Add Hosts).
2. Make sure Docker and Kubernetes* are correctly configured (refer to Section 3.4, Configure Kubernetes* and Docker* to Run with Proxy)
3. Run:

```
kubeadm reset
```

3.5.1.2 Common Issues

Common issues observed have been:

```
[WARNING Firewalld]: firewalld is active, please ensure ports [6443 10250] are open or your
cluster may not function correctly.
[WARNING HTTPPROXYCIDR]: Connection to 10.96.0.0/12 uses proxy http://192.168.10.1:3128. This may
lead to malfunction in cluster setup. Make sure that Pod and Services IP ranges specified
correctly as exceptions in proxy configuration.
[WARNING Hostname]: hostname iswlpbc135534 lookup iswlpbc135534 on [::1]:53 read udp [::1]:51215-
>[::1]:53: read: connection refused error making master: timed out waiting for the condition.
```

3.5.2 Kubernetes Initialization – Master/Non-Master node (on the same machine)

There is an option to allow the K8s Master node to be used as both the Master and Worker on a single server. To allow the Master node to be used as both Master/Worker, run the following command after the Master node initialization:

```
kubectl taint nodes --all node-role.kubernetes.io/master-
```

The normal response to this command is:

```
node <node name> untainted
```

Note: In a scenario when one node is used for Master/Worker, further instructions throughout the document dedicated for either Master or Worker node can be assumed to be run from the same node.

Section 3.5.3, Kubernetes Initialization on Non-Master Nodes can be skipped if using a Master/Worker set up on a single node.

3.5.3 Kubernetes Initialization on Non-Master Nodes

Do not use this initialization method for a single server used as both Master and Worker; instead, refer to Section 3.5.2, Kubernetes Initialization – Master/Non-Master node (on the same machine).

To initialize a Kubernetes non-master node:

1. Run kubeadm* to join, where **kubeadm join** is the command with the token provided by the kubeadm init command runs on the master node:

```
kubeadm join <master-ip>:<master-port> --token <token> --discovery-token-ca-cert-hash
sha256:<hash>
```

2. (**Optional**) In cases where the token provided by the master node has expired. After 24 hours, generate a new token from the master node with the following command:

```
kubeadm token create
```

3. On the non-master node, run this command:

```
kubeadm join <master-ip>:<master-port> --token <token>
```

4. To test that the non-master node successfully connected to the master, run the following command, and check the corresponding hostname appears as ready:

```
kubectl get nodes
```

Note: "kubeadm join" failed sometimes since the time of master and work node was out of sync. Under this scenario, you can sync the time between master and work node by using "date" command.

3.5.4 Testing the Kubernetes Master/Node Setup

To test master-to-nonmaster-node communication, deploy a sample POD.

1. Create the following file and name it busybox.yaml.

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default

spec:
  containers:
  - image: busybox
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
    name: busybox
  restartPolicy: Always
```

2. Create the POD from the `busybox.yaml` file.

```
kubectl create -f busybox.yaml
kubectl get pods
```

3. If all is working successfully, you should see the following output:

```
busybox         1/1     Running   0                   5s
```

4.0 Installation Guide for Common Plugins of Kubernetes*

4.1 Setting up the Multus Plugin with Kubernetes*

Multus (Multi Network) plugin is a Kubernetes container network interface (CNI) plugin. It enables multiple network interfaces to pods. Without Multus, each pod can only have one network interface, which is the default pod network (flannel*, calico*, etc.). With Multus, you can create a multi-homed pod that has multiple interfaces. This is accomplished by Multus acting as a "meta-plugin", a CNI plugin that can call multiple other CNI plugins.

4.1.1 Installation

Intel's recommended quick-start method to deploy Multus uses a Daemonset* (a method of running Pods on each node in a cluster); this spins up Pods, which install a Multus binary and configure Multus for usage.

The detailed information is available from multus GitHub - https://github.com/intel/multus-cni

```
        Download Multus from GitHub:
cd /root
git clone https://github.com/intel/multus-cni
cd /root/multus-cni/images
git checkout v3.3
kubectl create -f multus-daemonset.yml
```

4.1.2 Configuring Network Interface Using Customer Resource Definition

Using Kubernetes Customer Resource Definition (CRD) can easily add additional interfaces besides the default network interface (such as `macvlan`*).

1. Create CNI configurations, with the configuration files provided in `multus-cni/examples` folder.
2. Create CNI network attachment definitions:

```
kubectl create -f macvlan-conf.yml
```

Below is an example of macvlan configuration and enp0s31f6 is the network interface on a host in this example.

```
apiVersion: apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: macvlan-conf
spec:
  config: '{
      "cniVersion": "0.3.0",
      "type": "macvlan",
      "master": "enp0s31f6",
      "mode": "bridge",
      "ipam": {
        "type": "host-local",
        "subnet": "192.168.1.0/24",
```

```
        "rangeStart": "192.168.1.200",
        "rangeEnd": "192.168.1.216",
        "routes": [
          { "dst": "0.0.0.0/0" }
        ],
        "gateway": "192.168.1.1"
      }
    }'
```

3. Verify the CRD objects are created by:

```
kubectl get net-attach-def
```

The output is

```
NAME           AGE
macvlan-conf   3s
```

4.1.3 Test Multus Plugin

1. To test the setup, create a pod configuration file named `multus-test.yaml` with the following content:

```
apiVersion: v1
kind: Pod
metadata:
  name: multus-test
  annotations:
    k8s.v1.cni.cncf.io/networks: macvlan-conf
spec:  # specification of the pod's contents
  restartPolicy: Never
  containers:
  - name: test1
    image: "busybox"
    command: ["top"]
    stdin: true
    tty: true
```

2. Check the network interfaces after Pod created, you will see two interfaces: macvlan (net1) and calico* (eth0, default).

```
kubectl create -f multus-test.yaml
kubectl exec -it multus-test sh
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr EA:CC:C9:0D:53:84
          inet addr:192.168.151.21  Bcast:0.0.0.0  Mask:255.255.255.255
          UP BROADCAST RUNNING MULTICAST  MTU:1440  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 B)  TX bytes:0 (0.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
```

```
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

net1      Link encap:Ethernet  HWaddr 3A:F8:62:03:3C:E2
          inet addr:192.168.1.205  Bcast:0.0.0.0  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:18 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:2458 (2.4 KiB)  TX bytes:42 (42.0 B)
```

4.2 Setting up Calico Plugin for Kubernetes*

The instructions can be found from http://docs.projectcalico.org.

4.2.1 Building the Calico Plugin for Kubernetes

1. Download the yaml file for calico plugin

```
wget https://docs.projectcalico.org/v3.4/getting-
started/kubernetes/installation/hosted/calico.yaml
```

2. Open calico.yaml file and change the version of corresponding calico docker image (calico/node, calico/cni and calico/kube-controllers from v3.4.4 to v3.15.2) and add imagePullPoligy as “IfNotPreset”:

```
      ……
            - name: install-cni
              image: calico/cni:v3.15.2
              imagePullPolicy: IfNotPresent
      ……
```

```
      …….
            - name: calico-node
              image: calico/node:v3.15.2
              imagePullPolicy: IfNotPresent
      ……….
```

```
      …….
            - name: calico-kube-controllers
              image: calico/kube-controllers:v3.15.2
              imagePullPolicy: IfNotPresent
      ……….
```

3. Set up calico network

```
      Kubectl apply -f calico.yaml
```

4.2.2 Install calico CLI – calicoctl

Calicoctl allows you to create, read, update calico objects from the command line.

1. Install calicoctl as a binary:

```
curl -O -L  https://github.com/projectcalico/calicoctl/releases/download/v3.15.2/calicoctl
```

2. Change mode and place it to the /usr/local/bin

```
chmod +x calicoctl
cp calicoctl /usr/local/bin
```

3. Use calicoctl to check node status

```
DATASTORE_TYPE=kubernetes KUBECONFIG=~/.kube/config calicoctl get nodes -o yaml
```

Note: Sometimes, you may encounter an issue when try to ping through POD ip of each work node. Under this scenario, you can check the following parts to figure out root cause:

1. Check if kernel enable the forward

```
sysctl -a|grep forward|grep ipv4
```

to check if the following settings were enabled

```
net.ipv4.ip_forward = 1
net.ipv4.conf.tunl0.forwarding = 1
net.ipv4.conf.xxx.forwarding = 1
```

2. Check if iptables rule is wrong

Flush iptables rules at the node which can't be ping through

```
iptables -F
iptables -X
iptables -Z
iptables -t nat -F
iptables -t nat -X
iptables -t nat -Z
iptables -P FORWARD ACCEPT
```

3. Check if the node ip identified by calico is the same as the one identified by kubelet

At master node, execute below calico command to get the node information:

```
calicoctl get nodes -o wide
```

execute below kubectl command to get the node information:

```
kubectl get nodes -o wide
```

compare and check if there is different. If so, you need modify the environment variable of calico-node ds as following:

```
kubectl -n kube-system edit ds calico-node
```

to add following parts:

```
- name: CALICO_IPV4POOL_IPIP
  value: "can-reach=8.8.8.8"
```

4.3 Setting up SR-IOV CNI and Network Device Plugin for Kubernetes*

The setup instructions can be found from the sriov-cni GitHub https://github.com/intel/sriov-network-device-plugin. The SRIOV network device plugin is the Kubernetes device plugin for discovering and advertising SRIOV network virtual functions (VFs) available on a Kubernetes host.

4.3.1 Building the SRIOV CNI Plugin for Kubernetes

1. Download the plugin source code from GitHub and build:

```
mkdir -p $GOPATH/src/github.com/intel
cd $GOPATH/src/github.com/intel
git clone https://github.com/intel/sriov-cni
cd sriov-cni
git checkout v2.2
mkdir bin
cp $GOPATH/bin/golint bin/
make
```

2. Copy the binaries into the CNI folder of each worker node:

```
cd build
cp sriov /opt/cni/bin
```

Note: Make sure go language had been installed in your system. Or else, please download go package from golang.org

4.3.2 Build SRIOV Network Device Plugin

1. Clone the sriov-network-device-plugin from GitHub:

```
git clone https://github.com/intel/sriov-network-device-plugin
```

2. Build docker* image binary using make:s

```
cd sriov-network-device-plugin
git checkout v3.1
mkdir bin
cp $GOPATH/bin/golint bin/
make
make image
```

On a successful build, a docker image with tag `nfvpe/sriov-device-plugin:latest` will be created, build this image on each node. Alternatively, you could use a local docker registry to host this image.

If you build the image failed due to network issue, you can also download the image using docker pull:

```
docker pull nfvpe/sriov-device-plugin
```

4.3.3 Setting Up SRIOV and Run SRIOV Network Device Plugin

Note: Make sure the prerequisites for Single Root Input/Output Virtualization (SRIOV) are in place.

1. Load the driver:

```
modprobe vfio
modprobe vfio-pci
```

2. Set up Virtual Functions – in this scenario, set up one port for interface `enp103s0f0` (Intel® Ethernet Controller X710 for 10GbE SFP+ 1572) with device name `0000:67:00.0`. Set up one port for interface `enp103s0f1` (Intel® Ethernet Controller X710 for 10GbE SFP+ 1572) with device name `0000:67:00.1`.

```
echo 4 >  /sys/bus/pci/devices/0000\:67\:00.0/sriov_numvfs
echo 4 >  /sys/bus/pci/devices/0000\:67\:00.1/sriov_numvfs
```

3. Check to make sure that the VFs were created and bind VF interfaces.

```
ip link
```

The result should be four VFs visible under interface `enp103s0f0` and `enp103s0f0`, like the following:

```
5: enp103s0f0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop portid 3cfdfeb0ade8 state DOWN mode
DEFAULT qlen 1000
    link/ether 3c:fd:fe:b0:ad:e8 brd ff:ff:ff:ff:ff:ff
    vf 0 MAC 00:00:00:00:00:00, spoof checking on, link-state auto, trust off
    vf 1 MAC 00:00:00:00:00:00, spoof checking on, link-state auto, trust off
    vf 2 MAC 00:00:00:00:00:00, spoof checking on, link-state auto, trust off
    vf 3 MAC 00:00:00:00:00:00, spoof checking on, link-state auto, trust off
6: enp103s0f1: <BROADCAST,MULTICAST> mtu 1500 qdisc noop portid 3cfdfeb0ade9 state DOWN mode
DEFAULT qlen 1000
    link/ether 3c:fd:fe:b0:ad:e9 brd ff:ff:ff:ff:ff:ff
    vf 0 MAC 00:00:00:00:00:00, spoof checking on, link-state auto, trust off
    vf 1 MAC 00:00:00:00:00:00, spoof checking on, link-state auto, trust off
    vf 2 MAC 00:00:00:00:00:00, spoof checking on, link-state auto, trust off
    vf 3 MAC 00:00:00:00:00:00, spoof checking on, link-state auto, trust off
```

Check interface status:

```
$RTE_SDK/usertools/dpdk-devbind.py -s
#VF interfaces for PF enp103s0f0 with device name 67:00.0
0000:67:02.0 'XL710/X710 Virtual Function 154c' if=enp103s2 drv=i40evf unused=igb_uio,vfio-
pci
0000:67:02.1 'XL710/X710 Virtual Function 154c' if=enp103s2f1 drv=i40evf unused=igb_uio,vfio-
pci
0000:67:02.2 'XL710/X710 Virtual Function 154c' if=enp103s2f2 drv=i40evf unused=igb_uio,vfio-
pci
0000:67:02.3 'XL710/X710 Virtual Function 154c' if=enp103s2f3 drv=i40evf unused=igb_uio,vfio-
pci
#VF interfaces for PF enp103s0f1 with device name 67:00.1
0000:67:06.0 'XL710/X710 Virtual Function 154c' unused=i40evf,igb_uio,vfio-pci
0000:67:06.1 'XL710/X710 Virtual Function 154c' unused=i40evf,igb_uio,vfio-pci
0000:67:06.2 'XL710/X710 Virtual Function 154c' unused=i40evf,igb_uio,vfio-pci
0000:67:06.3 'XL710/X710 Virtual Function 154c' unused=i40evf,igb_uio,vfio-pci
```

Bind four VFs of `enp103s0f0` to `i40evf`:

```
$RTE_SDK/usertools/dpdk-devbind.py -b i40evf 67:02.0 67:02.1 67:02.2 67:02.3
```

Bind four VFs of `enp103s0f1` to `igb_uio`:

```
$RTE_SDK/usertools/dpdk-devbind.py -b vfio-pci 67:06.0 67:06.1 67:06.2 67:06.3
```

4. Create a `ConfigMap` that defines SR-IOV resource pool configuration:

```
cd /root/sriov-network-device-plugin/
cat <<EOF > deployments/configMap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: sriovdp-config
  namespace: kube-system
data:
  config.json: |
    {
        "resourceList": [{
                "resourceName": "intel_sriov_netdevice",
                "selectors": {
```

```
                    "vendors": ["8086"],
                    "devices": ["154c"],
                    "drivers": ["i40evf"],
                    "pfNames": ["enp103s0f0"]
                }
            },
            {
                "resourceName": "intel_sriov_dpdk",
                "selectors": {
                    "vendors": ["8086"],
                    "devices": ["154c"],
                    "drivers": ["vfio-pci"],
                    "pfNames": ["enp103s0f1"]
                }
            }
        ]
    }
EOF
kubectl create -f deployments/configMap.yaml
```

5. Deploy SRIOV network device plugin Daemonset:

```
kubectl create -f deployments/k8s-v1.18/sriovdp-daemonset.yaml
```

Once pods run successfully, you can see the allocable resource list for the worker node, which is discovered by the SRIOV network device plugin. The resource name is appended with the prefix "intel.com/intel_sriov_*".

```
kubectl get node <your-k8s-worker> -o json | jq '.status.allocatable' {
  "cpu": "28",
  "ephemeral-storage": "143494008185",
  "hugepages-1Gi": "48Gi",
  "intel.com/intel_sriov_dpdk": "4",
  "intel.com/intel_sriov_netdevice": "4",
  "memory": "48012416Ki",
  "pods": "110"
}
```

Note: A separate application of jq is required to get the result in above format.

4.3.4 Testing the SRIOV Network Device Plugin

1. Create the SRIOV Network CRD:

```
cat <<EOF > deployments/sriov-netdevice.yaml
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: sriov-netdevice1
  annotations:
    k8s.v1.cni.cncf.io/resourceName: intel.com/intel_sriov_netdevice
spec:
  config: '{
  "type": "sriov",
  "cniVersion": "0.3.1",
```

```
  "name": "sriov-network",
  "ipam": {
    "type": "host-local",
    "subnet": "10.56.217.0/24",
    "routes": [{
      "dst": "0.0.0.0/0"
    }],
    "gateway": "10.56.217.1"
  }
}'
EOF
kubectl create -f deployments/sriov-netdevice.yaml

cat <<EOF > deployments/sriov-dpdk-crd.yaml
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: sriov-dpdk1
  annotations:
    k8s.v1.cni.cncf.io/resourceName: intel.com/intel_sriov_dpdk
spec:
  config: '{
  "type": "sriov",
  "cniVersion": "0.3.1",
  "name": "sriov-dpdk"
}'
EOF
kubectl create -f deployments/sriov-dpdk-crd.yaml
```

2. Create test Pod for SRIOV device plugin:

```
cat <<EOF > deployments/pod-tc1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: testpod1
  annotations:
    k8s.v1.cni.cncf.io/networks: sriov-netdevice1, sriov-dpdk1
spec:
  containers:
  - name: appcntr1
    image: centos/tools
    imagePullPolicy: IfNotPresent
    command: [ "/bin/bash", "-c", "--" ]
    args: [ "while true; do sleep 300000; done;" ]
    resources:
      requests:
        intel.com/intel_sriov_netdevice: '1'
        intel.com/intel_sriov_dpdk: '1'
      limits:
        intel.com/intel_sriov_netdevice: '1'
        intel.com/intel_sriov_dpdk: '1'
```

```
EOF
The kubectl create -f deployments/pod-tc1.yaml
```

3. Check interfaces inside pod/container. After Pod is created and running, execute a shell from the pod and check configured interfaces like below::

```
kubectl exec testpod1 -it bash
[root@testpod1 /]# printenv | grep PCIDEVICE
PCIDEVICE_INTEL_COM_INTEL_SRIOV_DPDK=0000:67:06.3
PCIDEVICE_INTEL_COM_INTEL_SRIOV_NETDEVICE=0000:67:02.3
[root@testpod1 /]# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1440
        inet 192.168.151.13  netmask 255.255.255.255  broadcast 0.0.0.0
        ether f6:bc:6d:43:c1:b8  txqueuelen 0  (Ethernet)
        RX packets 8  bytes 648 (648.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

net1: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 10.56.217.10  netmask 255.255.255.0  broadcast 10.56.217.255
        ether d2:df:55:37:74:b0  txqueuelen 1000  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

4.4 Setting Up CMK for CPU Isolation

CPU Manager for Kubernetes* (CMK*) is used to aid in isolating CPUs and assigning tasks to them.

More information is available at CPU Manager for Kubernetes in Section 1.1.

4.4.1 Building CMK

1. Download CMK:

```
git clone https://github.com/intel/CPU-Manager-for-Kubernetes
git checkout v1.4.0
```

2. Go to directory:

```
cd CPU-Manager-for-Kubernetes
```

3. Edit the Dockerfile in the directory. The file must have the following content (omit proxy-related lines in **boldface** if there is no proxy server in your deployment environment):

```
# Copyright (c) 2017 Intel Corporation
#
```

```
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

FROM python:3.4.6

ADD requirements.txt /requirements.txt
RUN pip install -r /requirements.txt --proxy <your proxy server>

ADD . /cmk
WORKDIR /cmk

RUN chmod +x /cmk/cmk.py

RUN /cmk/cmk.py --help && echo ""

CMD [ "/cmk/cmk.py" ]
```

4. Build the CMK:

```
make
```

4.4.2 Initialize Cluster with CMK

CMK is used to initialize single or multiple nodes.

Note: For detailed instructions, refer to the *CMK Operator Manual* in Table 2, which describes the setup for one node.

In this example, the name of the node is `k8s-worker-skl`.

1. To create the necessary authorizations run:

```
cd CPU-Manager-for-Kubernetes/resources/authorization
kubectl create -f cmk-namespace.yaml
kubectl create -f cmk-serviceaccount.yaml
```

2. Create `ClusterRole` and `ClusterRoleBinding`:

```
kubectl create -f cmk-rbac-rules.yaml
```

3. Prepare key and certification using Openssl. Transfer `ca.key/ca.crt` to `base64` and paste to `webhook/cmk-webhook-certs.yaml` and `webhook/ cmk-webhook-config.yaml`:

```
openssl genrsa -out ca.key 2048
#change $MASTER_IP to your k8s master IP address
openssl req -x509 -new -nodes -key ca.key -subj "/CN=$MASTER_IP" -days 10000 -out ca.crt
cat ca.key | base64
cat ca.crt | base64
```

4. To Initialize the CMK using `cluster-init`:

- To configure the `cmk-cluster-init-pod.yaml`
- To set `--host-list=k8s-worker-skl --num-exclusive-cores=12 --num-shared-cores=1`

```
cat <<EOF > cmk-cluster-init-pod.yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: cmk-cluster-init-pod
  name: cmk-cluster-init-pod
  namespace: cmk-namespace
spec:
  serviceAccountName: cmk-serviceaccount
  containers:
  - args:
      # Change this value to pass different options to cluster-init.
      - "/cmk/cmk.py cluster-init --host-list=k8s-worker-skl --num-exclusive-cores=12 --num-
shared-cores=1 --saname=cmk-serviceaccount --namespace=cmk-namespace"
    command:
    - "/bin/bash"
    - "-c"
    image: cmk:v1.4.0
    name: cmk-cluster-init-pod
  restartPolicy: Never
EOF
kubectl create -f cmk-cluster-init-pod.yaml
```

Remember to remove below taint if the pod is pending to schedule.

```
kubectl taint nodes k8s-worker-skl cmk=true:NoSchedule-
```

It takes a few of minutes to finish the initialization. After done, below pods are running or completed.

```
NAMESPACE       NAME                                                READY   STATUS
RESTARTS   AGE   IP               NODE              NOMINATED NODE   READINESS GATES
cmk-namespace   cmk-cluster-init-pod                                0/1     Completed   0
82m   192.168.151.55    k8s-worker-skld   <none>           <none>
cmk-namespace   cmk-init-install-discover-pod-k8s-worker-skld       0/2     Completed   0
80m   192.168.151.56    k8s-worker-skld   <none>           <none>
cmk-namespace   cmk-reconcile-nodereport-ds-k8s-worker-skld-hgwls   2/2     Running     0
78m   192.168.151.57    k8s-worker-skld   <none>           <none>
cmk-namespace   cmk-webhook-deployment-bf66477bd-b8qn7              1/1     Running     0
78m   192.168.235.199   k8s-master        <none>
```

5. Run a test Pod to request CPU from the cluster. Change `Pods/cmk-isolate-pod.yaml` then create the Pod:

```
# NOTE: To be used with k8s >= 1.9.0 (if webhook is running).

apiVersion: v1
kind: Pod
metadata:
  labels:
    app: cmk-isolate-pod
  name: cmk-isolate-pod
# Consumed by mutating webhook
  annotations:
```

```
    cmk.intel.com/mutate: "true" # accepted values to trigger mutation: "true", "True", "1"
  namespace: cmk-namespace
spec:
  restartPolicy: Never
  containers:
  - name: cmk-isolate-exclusive
    image: cmk:v1.4.0
    command:
    - "/bin/bash"
    - "-c"
    args:
    - "/opt/bin/cmk isolate --conf-dir=/etc/cmk --pool=exclusive env && sleep 10000"
    resources:
      requests:
        cmk.intel.com/exclusive-cores: '4'
      limits:
        cmk.intel.com/exclusive-cores: '4'
  - name: cmk-isolate-shared
    image: cmk:v1.4.0
    command:
    - "/bin/bash"
    - "-c"
    args:
    - "/opt/bin/cmk isolate --conf-dir=/etc/cmk --pool=shared env && sleep 10000"
  - name: cmk-isolate-infra
    image: cmk:v1.4.0
    command:
    - "/bin/bash"
    - "-c"
    args:
    - "/opt/bin/cmk isolate --conf-dir=/etc/cmk --pool=infra env && sleep 10000"
kubectl create -f cmk-isolate-pod.yaml
```

After the Pod is running, check the CPU numbers allocated to the Pod:

```
kubectl logs -n cmk-namespace cmk-isolate-pod -c cmk-isolate-exclusive | grep -i "cpu"
CMK_CPUS_ASSIGNED=1,15,10,24,11,25,12,26
CMK_CPUS_ASSIGNED_MASK=7009C02
CMK_CPUS_INFRA=0,14
kubectl logs -n cmk-namespace cmk-isolate-pod -c cmk-isolate-shared | grep -i "cpu"
CMK_CPUS_ASSIGNED=13,27
CMK_CPUS_ASSIGNED_MASK=8002000
CMK_CPUS_INFRA=0,14
kubectl logs -n cmk-namespace cmk-isolate-pod -c cmk-isolate-infra | grep -i "cpu"
CMK_CPUS_ASSIGNED=0,14
CMK_CPUS_ASSIGNED_MASK=4001
CMK_CPUS_INFRA=0,14
```

Note: In this case, hyperthread is enabled. Two logical cores (0 and 14, 1 and 15, 10 and 24, etc) are taken as one physical core.

Note: CMK should base on “isolcpus” setting in GRUB to configure the exclusive or none-exclusive core among containers

Note: The none-exclusive core configured by CMD is totally different with the core not configured in “isolcpus”

setting of GRUB. No matter none-exclusive core or exclusive core, they are all derive from “isolcpus”.

4.5 Native Huge Pages Support in Kubernetes*

Native Kubernetes Huge Pages allocation is supported. Kubernetes Huge Pages isolation occurs in a Pod scope and their consumption via container level resource requirement.

Note: Huge Pages must be allocated at the start (1 Gi) or shortly after the start.

4.5.1 Running a Sample Pod with Native Huge Pages

The Huge Pages are consumed at the container level and need to be specified in required resources within Pod specification.

1. To run a sample native Huge Pages pod that requires 3 GB Huge Page, create a pod configuration file:

```
vim sample-hp-pod.yaml
```

   The file should include the following specifications:

```
apiVersion: v1
kind: Pod
metadata:
  generateName: hugepages-volume-
spec:
  containers:
  - image: centos:latest
    command:
    - sleep
    - inf
    name: example
    volumeMounts:
    - mountPath: /hugepages
      name: hugepage
    resources:
      limits:
        hugepages-1Gi: 3Gi
        memory: "3Gi"
  volumes:
  - name: hugepage
    emptyDir:
      medium: HugePages
```

2. Create the Pod:

```
kubectl create -f sample-hp-pod.yaml
```

   For an example of FlexRAN Pod specification with native Huge Pages allocation, refer to Section 5.2.2.

4.6 Native CPU Management Support in Kubernetes*

Since Kubernetes v1.16.1, there is the support of the CPU manager and topology manager. In this section, steps are provided on how to enable and use these features. You can get more information from the Kubernetes document:

https://kubernetes.io/docs/tasks/administer-cluster/cpu-management-policies/

https://kubernetes.io/docs/tasks/administer-cluster/topology-manager/

These features are controlled by the kubelet* on the worker node. To enable these features, change the kuberlet configuration of worker node and restart kubelet.

4.6.1 Update kubelet configuration files and restart kubelet

Enable topologyManagerPolicy: best-effort in configuration file /var/lib/kubelet/config.yaml. Change /usr/lib/systemd/system/kubelet.service.d/10-kubeadm.conf with below parameters:

```
Environment="KUBELET_CONFIG_ARGS=--config=/var/lib/kubelet/config.yaml --feature-gates
TopologyManager=true --cpu-manager-policy=static --system-reserved=cpu=1,memory=1Gi --topology-
manager-policy=best-effort"
```

restart kubelet by

```
systemctl daemon-reload
systemctl restart kubelet
```

If you see below issues from kubelet status,

```
Mar 12 05:05:17 ironic-node-53 kubelet: E0312 05:05:17.510725 235690
container_manager_linux.go:328] failed to initialize cpu manager: could not initialize checkpoint
manager: could not restore state from checkpoint: configured policy "static" differs from state
checkpoint policy "none"
Mar 12 05:05:17 ironic-node-53 kubelet: Please drain this node and delete the CPU manager
checkpoint file "/var/lib/kubelet/cpu_manager_state" before restarting Kubelet.
Mar 12 05:05:17 ironic-node-53 kubelet: F0312 05:05:17.510741 235690 server.go:271] failed to run
Kubelet: could not initialize checkpoint manager: could not restore state from checkpoint:
configured policy "static" differs from state checkpoint policy "none"
Mar 12 05:05:17 ironic-node-53 kubelet: Please drain this node and delete the CPU manager
checkpoint file "/var/lib/kubelet/cpu_manager_state" before restarting Kubelet.
Mar 12 05:05:17 ironic-node-53 systemd: kubelet.service: main process exited, code=exited,
status=255/n/a
Mar 12 05:05:17 ironic-node-53 systemd: Unit kubelet.service entered failed state.
Mar 12 05:05:17 ironic-node-53 systemd: kubelet.service failed.
```

Run below command from Kubernetes master to drain this node and recover this node.

```
kubectl drain k8s-worker --ignore-daemonsets
kubectl uncordon k8s-worker
rm -rf /var/lib/kubelet/cpu_manager_state
```

4.6.2 Create a test POD to test CPU manager

```
cat <<EOF > test-cpu-manager.yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: test-cpu-manager
  name: test-cpu-manager
spec:
  containers:
  - image: centos:centos7.8.2003
    command:
    - sleep
    - inf
```

```
    name: example
    resources:
      requests:
        cpu: "4"
        memory: "1Gi"
      limits:
        cpu: "4"
        memory: "1Gi"
EOF
kubectl create -f test-cpu-manager.yaml
```

Login the POD/container and check the taskset:

```
# kubectl exec test-cpu-manager -it bash
# taskset -p 1
pid 1's current affinity mask: 1e
```

So cores 1,2,3,4 are assigned to this container exclusively.

4.7 Using Node Feature Discovery

The Node Feature Discovery enables the detection of hardware features available on each node. With this feature enabled, the user can deploy the PODs on nodes with specific hardware requirements by providing relevant node labels in the POD specification. To deploy a POD on any node that provides a given feature, the node label for this feature must be provided in the POD specification.

Refer to *Node Feature Discovery* in Table 2.

4.7.1 Getting Source and Docker Image of Node Feature Discovery

To build Node Feature Discovery:

1. Get source:

   ```
   git clone https://github.com/kubernetes-sigs/node-feature-discovery
   ```

2. Go into the directory (assumed source cloned to /root):

   ```
   cd /root/node-feature-discovery
   #go to release version 0.5.0
   git checkout v0.5.0
   ```

3. Build a docker container image:

   If there are issues when building the NFD docker image, pull the image from the public registry:

   ```
   docker pull "quay.io/kubernetes_incubator/node-feature-discovery:v0.5.0"
   ```

   If you're working with a proxy, the Dockerfile* must be edited to add a proxy server:

   ```
   vim Dockerfile
   ```

4. Edit the Dockerfile to add a proxy:

   ```
   FROM golang:1.8

   ENV http_proxy <your_http_proxy>
   ENV no_proxy "localhost,127.0.0.1"
   ENV https_proxy <your_https_proxy>
   ```

```
…
```

5. Build the NFD:

```
make


After done, you will have a docker image available with tag
"quay.io/kubernetes_incubator/node-feature-discovery:v0.5.0".
```

4.7.2 Running Node Feature Discovery

1. To run nfd-master as k8s DaemonSet:

   Use the template Specification provided in the source to deploy the nfd-master.

```
kubectl create -f nfd-master.yaml.template
```

   When the nfd-master-* POD is running, it listens for connections from nfd-workers then connects to the K8s API server to add node-specific labels, which are provided by the nfd-workers.

2. Running the nfd-workers as DaemonSet:

   Similar to nfd-master, nfd-workers is run as a DaemonSet. Use the template spec in the source to deploy the nfd-workers.

```
kubectl create -f nfd-worker-daemonset.yaml.template
```

   After nfd-worker-* PODs are running, they connect to the nfd-master service to advertise hardware-specific features.

   Alternatively, the feature discovery can be configured as a one-time job. There is an example script in the source that demonstrates how to deploy the job by running one script:

```
./label-nodes.sh [<IMAGE_TAG>]
```

3. Check the labels:

   Check the labels of each worker node by running below command from a master node

```
kubectl get nodes -o json | jq .items[].metadata.labels
```

   The output will be similar to below:

```
{
  "feature.node.kubernetes.io/cpu-<feature-name>": "true",
  "feature.node.kubernetes.io/iommu-<feature-name>": "true",
  "feature.node.kubernetes.io/kernel-<feature name>": "<feature value>",
  "feature.node.kubernetes.io/memory-<feature-name>": "true",
  "feature.node.kubernetes.io/network-<feature-name>": "true",
  "feature.node.kubernetes.io/pci-<device label>.present": "true",
  "feature.node.kubernetes.io/storage-<feature-name>": "true",
  "feature.node.kubernetes.io/system-<feature name>": "<feature value>",
  "feature.node.kubernetes.io/<file name>-<feature name>": "<feature value>"
}
```

   A separate application of jq is required to get the result in above format.

4.7.3 Testing Node Feature Discovery

For a POD to be deployed on any node that provides a given feature, the POD specification must contain the node label for the feature.

1. To discover if a POD requires a given feature, enter the following command:

```
kubectl get nodes -o json | jq .items[].metadata.labels
```

   For example, if the POD requires multithreading, the feature list includes the line:

```
feature.node.kubernetes.io/cpu-hardware_multithreading: "true"
```

2. To deploy a POD on a node that provides a given feature, create a POD specification file:

```
vim sample-nfd-pod.yaml
```

   The following is sample content to be added to the created file:

Note: The nodeSelector section must include the required label (labels can be added or changed based on the features required).

```
apiVersion: v1
kind: Pod
metadata:
  name: nfd-test
  namespace: default
  labels:
    app: test
spec:
 containers:
   - name: nfd-test-cont
     image: golang
     imagePullPolicy: IfNotPresent
     command:
        - sleep
        - infinity
 nodeSelector:
    feature.node.kubernetes.io/cpu-hardware_multithreading: "true"
```

3. Create the POD:

```
kubectl create -f sample-nfd-pod.yaml
```

   Upon success, the POD will be deployed and running on a node supporting multithreading.

4.8 Device Plugin Usage (LTE FPGA FEC and NR FPGA FEC/FH)

As part of the FlexRAN Board Support Package (BSP), a device plugin allowing a user to pass the FEC Field Programmable Gate Array (FPGA) device (MAP 80/100 for LTE) and FEC and FrontHaul (FH) FPGA devices (Terasic* for 5G NR).

The code for the device plugin can be accessed from the/misc/device_plugins directory of the mentioned package. Instructions on how to set up and use/test the plugins can be found under /misc/device_plugins/ReadMe.

The instructions in the ReadMe file must be completed before proceeding to start FlexRAN with the FPGA device.

This section provides the instructions for creating the specification file and starting the test POD with the desired resources (Starting FlexRAN inside the said container is outside the scope of this section).

4.8.1 LTE FEC FPGA Device Plugin

It is assumed the device plugin is set up according to the section in the ReadMe on LTE FEC FPGA device plugin and that programmed MAP 80/100 FPGA Device is configured and available in the platform.

1. To create a POD consuming, an LTE FEC VF resources the following POD specification busybox.yaml file needs to be created:

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default
spec:
  containers:
  - image: busybox
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
    name: busybox
    resources:
      requests:
        intel.com/fec_lte_vf: '1'
      limits:
        intel.com/fec_lte_vf: '1'
  restartPolicy: Always
```

2. Create the POD from the busybox.yaml file.

```
kubectl create -f busybox.yaml
kubectl get pods
If all is working successfully, you should see the following output:
Busybox    1/1    Running    5    5s
```

3. The FEC device should be seen under /dev directory (bound to IGB_UIO). Use below command below to see the PCI address of the FPGA VF:

```
kubectl exec busybox -it sh
printenv | grep FEC_LTE_FPGA_VF_PCI_ADDR
```

4.8.2 5G NR FEC/FH FPGA Device Plugin

It is assumed the device plugin is set up according to the section in the ReadMe on the 5G New Radio (NR) FEC/FH FPGA device plugin and that programmed Terasic FPGA devices are configured and available in the platform.

1. Create a POD consuming NR FEC PF and NR FH PF resources, the following POD specification busybox.yaml file needs to be created.

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox
  namespace: default

spec:
```

```
  containers:
  - image: busybox
    command:
      - sleep
      - "3600"
    imagePullPolicy: IfNotPresent
    name: busybox
    resources:
      requests:
        intel.com/fec_nr_pf: '1'
        intel.com/fh_nr_pf: '1'
      limits:
        intel.com/fec_nr_pf: '1'
        intel.com/fh_nr_pf: '1'
  restartPolicy: Always
```

2. To create the PODs, run:

```
kubectl create -f busybox.yaml
```

Use below command to see the PCI address of the FPGA PF:

```
kubectl exec busybox -it sh
printenv | grep FH_NR_FPGA_PF_PCI_ADDR
printenv | grep FEC_NR_FPGA_PF_PCI_ADDR
```

§

4.9 Enable SRIOV FPGA Device Plugin

Except for FPGA device plugin described in chapter 4.8, SRIOV based FPGA device plugin also can be used.

This section explains how to enable FPGA orchestration using the open source SRIOV network device plugin. The setup instructions can be found from the GitHub https://github.com/intel/sriov-network-device-plugin. The SRIOV network device plugin is the Kubernetes device plugin for discovering and advertising the SRIOV network and FPGA accelerator virtual functions (VFs) available on a Kubernetes host. In this guide, this device plugin is used to manage SRIOV FPGA devices.

4.9.1 Building the SRIOV CNI Plugin for Kubernetes

Please following chapter 4.3.1 to build the SRIOV CNI plugin for Kubernetes.

4.9.2 Pull the SRIOV Network Device Plugin

You can download image using docker pull as below:

```
docker pull nfvpe/sriov-device-plugin
```

4.9.3 Setting Up SRIOV and Run SRIOV Network Device Plugin

Note: Make sure the prerequisites for Single Root Input/Output Virtualization (SRIOV) are in place.

1. Load the driver:

```
modprobe vfio
modprobe vfio-pci
```

2. Setup Intel PAC N3000 FPGA SRIOV functions

   Assume you have followed FlexRAN release document to program the FPGA card with 5G user image.

   Find device 0d8f and bind to dpdk igb_uio driver

```
lspci | grep 0d8f
     1f:00.0 Processing accelerators: Intel Corporation Device 0d8f (rev 01)
/opt/dpdk/usertools/dpdk-devbind.py -b igb_uio 1f:00.0
```

   Use the pf-bb-config application from GitHub to configure FPGA resources for different VFs. Here assume 2 VFs are configured for the FPGA (Please refer to README.md in pf-bb-config for creating VFs). Resources are equally assigned to the 2 VFs.

```
git clone https://github.com/intel/pf-bb-config.git
cd pf-bb-config
cat fpga_5gnr/fpga_5gnr_config_vf.cfg
[MODE]
pf_mode_en = 0

[UL]
bandwidth = 3
load_balance = 128
vfqmap = 16,16,0,0,0,0,0,0

[DL]
bandwidth = 3
load_balance = 128
vfqmap = 16,16,0,0,0,0,0,0

[FLR]
flr_time_out = 610

./pf_bb_config FPGA_5GNR -c fpga_5gnr/fpga_5gnr_config_vf.cfg -p 0000:1f:00.0
```

   Bind the two VFs to vfio-pci driver.

```
lspci | grep 1f:00
1f:00.1 Processing accelerators: Intel Corporation Device 0d90 (rev 01)
1f:00.2 Processing accelerators: Intel Corporation Device 0d90 (rev 01)
/opt/dpdk/usertools/dpdk-devbind.py -b vfio-pci 1f:00.2 1f:00.1
```

3. Create a ConfigMap that defines SR-IOV resource pool configuration:

```
cd /root/sriov-network-device-plugin/
cat <<EOF > deployments/configMap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: sriovdp-config
  namespace: kube-system
data:
  config.json: |
```

```
    {
        "resourceList": [{
               "resourceName": "intel_fpga",
                "deviceType": "accelerator",
                "selectors": {
                    "vendors": ["8086"],
                    "devices": ["0d90"]
                }
            }
        ]
    }
EOF
kubectl create -f deployments/configMap.yaml
```

4. Deploy SRIOV network device plugin Daemonset:

```
kubectl create -f deployments/k8s-v1.18/sriovdp-daemonset.yaml
```

Once pods run successfully, you can see the allocable resource list for the worker node, which is discovered by the SRIOV network device plugin.

```
kubectl get node <your-k8s-worker> -o json | jq '.status.allocatable' {
  "cpu": "28",
  "ephemeral-storage": "143494008185",
  "hugepages-1Gi": "48Gi",
  "intel.com/intel_fpga": "2",
  "memory": "48012416Ki",
  "pods": "110"
}
```

A separate application of jq is required to get the result in above format.

4.9.4 Testing the SRIOV Network Device Plugin

1. Create test POD for SRIOV device plugin:

```
cat <<EOF > deployments/pod-tc1.yaml
apiVersion: v1
kind: Pod
metadata:
  name: testpod1
spec:
  containers:
  - name: appcntr1
    image: centos/tools
    imagePullPolicy: IfNotPresent
    command: [ "/bin/bash", "-c", "--" ]
    args: [ "while true; do sleep 300000; done;" ]
    resources:
      requests:
        intel.com/intel_fpga: '1'
      limits:
        intel.com/intel_fpga: '1'
EOF
kubectl create -f deployments/pod-tc1.yaml
```

2. Check allocated FPGA device inside pod/container. After Pod is created and running, execute a shell from the pod and check allocated resources like below:

```
kubectl exec testpod1 -it bash
[root@testpod1 /]# printenv | grep PCIDEVICE
PCIDEVICE_INTEL_COM_INTEL_FPGA=0000:1f:00.1
```

4.10 Setup Intel® QAT Device Plugin for Kubernetes

To easily manage Intel® QAT available resources under Kubernetes, Intel developed the QAT device plugin as part of the Intel device plugins for Kubernetes. The supported devices are determined by the VF device drivers available in the Linux Kernel. Below is a list of supported devices:

- Intel® Xeon® with Intel® C62X Series Chipset
- Intel® Atom™ Processor C3000
- Intel® Communications Chipset 8925 to 8955 Series

The Intel® QAT device plugin provides access to Intel® QAT VF to the container. Like the FEC device plugin, each container can request a certain number of Intel® QAT VF resources and run network applications, with these resources.

The Intel® QAT device plugin supports two modes of using the Intel® QAT VF resources – DPDK and kernel. In this document, we only show the DPDK mode. The kernel-mode usage is following the same procedure. Refer to the Intel® QAT device plugin github for all the details:

https://github.com/intel/intel-device-plugins-for-kubernetes/blob/master/cmd/qat_plugin/README.md

4.10.1 Getting the Source Code From Github

Assume you have set up the go environment already. Run below command to get source code from Github:

```
export MY_GOPATH=$(go env GOPATH)
mkdir -p $MY_GOPATH/src/github.com/intel
git clone https://github.com/intel/intel-device-plugins-for-kubernetes
cd $MY GOPATH/src/github.com/intel/intel-device-plugins-for-kubernetes
git checkout v0.15.0
```

4.10.2 Build or Download the Docker Image

Update the Intel® QAT device plugin Dockerfile ($MY_GOPATH/src/github.com/intel/intel-device-plugins-for-kubernetes/build/docker/intel-qat-plugin.Dockerfile) with below changes:

```
ARG
CLEAR_LINUX_BASE=clearlinux/golang@sha256:3b7841bb4fc15d6b6cfb3bdef12f385696efa9915541223848774a071c29be03
FROM ${CLEAR_LINUX_BASE} as builder
ARG CLEAR_LINUX_VERSION="--version=30970"
ARG TAGS_KERNELDRV=
ENV http_proxy <your_proxy>
ENV https_proxy <your_proxy>
…
Go to folder "$MY_GOPATH/src/github.com/intel/intel-device-plugins-for-kubernetes" and build QAT device plugin Docker image:
```

```
cd $MY_GOPATH/src/github.com/intel/intel-device-plugins-for-kubernetes/
make intel-qat-plugin
…
Successfully tagged intel/intel-qat-plugin:0.15.0
```

If you meet Docker image build issues in the step above, download the pre-built image from Docker Hub. In this document, version 0.15.0 is used. Download the correct version of the image using the below command:

```
docker pull intel/intel-qat-plugin:0.15.0
```

You can download the correct version of your need.

4.10.3 Deploy QAT device plugin as a DaemonSet

1. Configure the SR-IOV VF on the worker node. This worker node has integrated the Intel® QAT in the CPU and three Intel® QAT PFs b5:00.0, b7:00.0, b9:00.0

   ```
   echo 16 > /sys/bus/pci/drivers/c6xx/0000:b5:00.0/sriov_numvfs
   echo 16 > /sys/bus/pci/drivers/c6xx/0000:b7:00.0/sriov_numvfs
   echo 16 > /sys/bus/pci/drivers/c6xx/0000:b9:00.0/sriov_numvfs
   ```

   So a total 48 VFs are configured on the worker node, and these VFs are bound to VF kernel driver initially.

2. Update the DPDK driver and max VF devices number in the `configmap` of the Intel® QAT plugin. Here we use `vfio` since it's more robust and secure (another reason is - we met issue when using `igb_uio`).

   ```
   vi $MY_GOPATH/src/github.com/intel/intel-device-plugins-for-
   kubernetes/deployments/qat_plugin/qat_plugin_default_configmap.yaml
   apiVersion: v1
   kind: ConfigMap
   metadata:
     name: intel-qat-plugin-config
   data:
     DPDK_DRIVER: "vfio-pci"
     KERNEL_VF_DRIVERS: "dh895xccvf,c6xxvf,c3xxxvf,d15xxvf"
     MAX_NUM_DEVICES: "32"
     DEBUG: "false"
   ```

3. Create `configmap` and `demonset`:

   ```
   cd $GOPATH/src/github.com/intel/intel-device-plugins-for-kubernetes
   kubectl create -f deployments/qat_plugin/qat_plugin_default_configmap.yaml
   kubectl create -f deployments/qat_plugin/qat_plugin.yaml
   ```

4. After the Intel® QAT device plugin POD is running on correct nodes, verify the Intel® QAT device plugin is registered, and the Intel® QAT resources are available:

   ```
   kubectl describe node <node name> | grep qat.intel.com/generic
    qat.intel.com/generic: 32
    qat.intel.com/generic: 32
   ```

4.10.4 Run DPDK Crypto Test to Consume QAT Device Plugin

Run the DPDK crypto sample app to show how to consume Intel® QAT device plugins in the container environment. Create a below POD configuration file.

```
cat <<EOF > dpdk-qat.yaml
kind: Pod
apiVersion: v1
```

```
metadata:
  name: dpdk
spec:
  containers:
  - name: dpdk
    image: centos:centos7.8.2003
    imagePullPolicy: IfNotPresent
    command: [ "/bin/bash", "-c", "--" ]
    args: [ "while true; do sleep 300000; done;" ]
    securityContext:
      readOnlyRootFilesystem: true
      privileged: false
      capabilities:
        add:
          ["IPC_LOCK", "SYS_ADMIN"]
    volumeMounts:
    - mountPath: /var/run/dpdk
      name: dpdk-runtime
    - mountPath: /dev/hugepages
      name: hugepage
    - name: dpdk-path
      mountPath: /opt/dpdk-5g
    - name: usr-path
      mountPath: /root/usr
    resources:
      requests:
        cpu: "3"
        memory: "128Mi"
        qat.intel.com/generic: '4'
        hugepages-1Gi: "1Gi"
      limits:
        cpu: "3"
        memory: "128Mi"
        qat.intel.com/generic: '4'
        hugepages-1Gi: "1Gi"
  restartPolicy: Never
  volumes:
  - name: dpdk-runtime
    emptyDir:
      medium: Memory
  - name: hugepage
    emptyDir:
      medium: HugePages
  - hostPath:
      path: "/opt/dpdk-5g"
    name: dpdk-path
  - hostPath:
      path: "/usr"
    name: usr-path
EOF
```

In the POD configuration, mount the prebuilt DPDK from host in to container (/opt/dpdk-5g). To keep it simple, some dynamic libraries (/usr/lib and /usr/lib64) needed by DPDK sample app runtime, are mounted to the container (/root/usr). Change the "hugepages-1Gi" part to your configuration (like hugepage-2Mi). Also change qat.intel.com/generic: '4' for your need.

Create the POD/container with above configuration, execute into the container and see the allocated PCIe devices. Get the device address of 4 Intel® QAT devices allocated.

```
kubectl create -f dpdk-qat.yaml
#execute into the container
kubectl exec -it dpdk bash
printenv | grep QAT
QAT2=0000:b7:01.1
QAT3=0000:b5:01.2
QAT0=0000:b7:02.6
QAT1=0000:b7:02.3
```

Finally inside container run the dpdk-test-crypto-perf sample app and view the output

```
cd /opt/dpdk-5g/x86_64-native-linuxapp-icc/app
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/root/usr/lib64/:/root/usr/lib
./dpdk-test-crypto-perf -c 0xf -w $QAT0 -w $QAT1 -w $QAT2 -w $QAT3 -- --ptest throughput --
devtype crypto_qat --optype cipher-only --cipher-algo aes-cbc --cipher-op encrypt --cipher-key-sz
16   --total-ops 10000000 --burst-sz 32 --buffer-sz 1024
```

5.0 FlexRAN Run on Container through Kubernetes*

5.1 FlexRAN Installation Guide

For FlexRAN to work inside Kubernetes* PODs, FlexRAN and the Data Plane Development Kit* (DPDK) must first be built and installed on the non-master nodes. In this scenario, it was compiled on CentOS Linux*, with the RT kernel. Also, the Intel® C++ Compiler must be installed. Supported versions of FlexRAN, DPDK, and ICC are specified in the *FlexRAN Reference Solution Release Notes*, Table 2.

5.1.1 Pre-Configuration

1. Install Intel® System Studio (ICC)
2. Edit command line parameters: /etc/grub/default

   Huge pages and CPU-core-related parameters may vary depending on the setup. Refer to Real-Time OS Installation and Configuration for information on setting up the platform with correct parameters.
3. Update GRUB and restart the system.

5.1.2 Building and Installing DPDK

DPDK v20.11 is used in this scenario to work with FlexRAN. To build the DPDK on the non-master node, follow these steps:

1. Download and unpack DPDK:

```
wget https://fast.dpdk.org/rel/dpdk-20.11.tar.xz
tar xf /dpdk_install_path/dpdk-20.11.tar.xz
cd dpdk-20.11
```

2. The dpdk_install_path can anywhere except for /root.
3. Prepare the compile env for DPDK:

```
pip3 install meson
yum install ninja-build
```

4. Apply patch for 20.11 – dpdk-20.11.patch

```
export RTE_SDK=/opt/dpdk
cd $RTE SDK
patch -p1 < dpdk-20.11.patch
```

5. Prepare the dependent lib for DPDK compile

```
git clone https://gitlab.devtools.intel.com/flexran/wireless_sdk.git
```

6. Prepare the dependent lib for DPDK compile

```
cat <<EOF > dpdk_build.sh
#! /bin/bash

work_path=$PWD
dpdk_path=/opt/dpdk_20.11
sdk_path=$work_path/wireless_sdk
```

```
echo "------------build base dpdk20.11 -------------------"
cd $dpdk_path; meson build;cd build ; meson configure -Dflexran_sdk=$sdk_path/build-avx512-icc/install; ninja
```

Note: If the DPDK is located in the /root directory, the patch procedure will fail.
Note: Refer to the FlexRAN 4G Reference Solution PHY Software Documentation Table 2 to apply the DPDK patch and built for 4G use. Assume the 4G DPDK is built and located in /opt/dpdk-lte.
Note: Follow the FlexRAN 5GNR Reference Solution PHY Software Documentation Table 2 to apply the DPDK patch and build for 5G use. Assume the 5G DPDK is built and located in /opt/dpdk-5g.

5.1.3 Building and Installing FlexRAN

For the latest 4G and 5G build instructions, refer to these two documents:

- For 4G compilation, refer to release document *FlexRAN 4G Reference Solution PHY Software Documentation (Doxygen)* Table 2.
- For 5G compilation, refer to *FlexRAN 5GNR Reference Solution PHY Software Documentation (Doxygen)* Table 2.

Note: In this scenario (as can be seen from the script), Wireless Subsystem (WLS) is built as a DPDK mode instead of kernel module mode, which allows the FlexRAN POD to run in a non-root container.

5.1.4 Testing FlexRAN LTE L1 and Testmac on the Host

1. To run FlexRAN, huge pages* must be set as follows:

```
mkdir /mnt/huge
mount -t hugetlbfs nodev /mnt/huge
```

2. The example from step 1 should be set and also added to /root/.bashrc.

Note: Paths may vary depending on the name and location of the FlexRAN directory.

3. To test open two separate terminals, run these commands from the first terminal:

   For 4G LTE,

```
cd /flexan_install_path/bin/lte/l1
./l1.sh -e
```

   Or, for 5G NR,

```
cd /flexan install path/bin/nr5g/gnb/l1
./l1.sh -e
```

4. In the second terminal, run the following commands (the name of test file may vary):

   For 4G LTE,

```
cd /flexran_install_path/bin/lte/testmac
./l2.sh --testfile=<test.cfg>
```

   Or, for 5G NR,

```
cd /flexran_install_path/bin/nr5g/gnb/testmac
./l2.sh --testfile=<test.cfg>
```

5. For latest instructions on how to run FlexRAN code, please refer release documents:

   - For the 4G test, refer to release document *FlexRAN 4G Reference Solution PHY Software Documentation (Doxygen)* Table 2.

- For the 5G test, refer to *FlexRAN 5GNR Reference Solution PHY Software Documentation (Doxygen)* Table 2.

Note: The following is an example test profile. Pick and choose between profiles depending on the desired spec (for example, platform (SKL-SP/D, CSL-SP), hyper-threading on or off, FEC offload simulation on or off).

```
./l2.sh --testfile=skylake-sp/sklsp_p5_htoff_fecon.cfg
```

Upon a successful run, communication between two layers should be observed.

Note: To avoid the "zombie" processes, always use the exit command to stop the test.

5.1.5 Setting up the Precision Time Protocol (PTP)

The Precision Time Protocol (PTP) provides synchronization services to the environment. It is a mandatary configuration for FlexRAN radio mode (E2E) setup.

Installing the linuxptp package provides the ptp4l and phc2sys applications.

PTP must be configured on the Grandmaster clock first, and then the non-master clock is set up and synchronized to it.

To verify the systems, NIC uses hardware timestamps, run ethtool. Output similar to the following should appear:

```
ethtool -T eno4
Time stamping parameters for eno4:
Capabilities:
        hardware-transmit     (SOF_TIMESTAMPING_TX_HARDWARE)
        software-transmit     (SOF_TIMESTAMPING_TX_SOFTWARE)
        hardware-receive      (SOF_TIMESTAMPING_RX_HARDWARE)
        software-receive      (SOF_TIMESTAMPING_RX_SOFTWARE)
        software-system-clock (SOF_TIMESTAMPING_SOFTWARE)
        hardware-raw-clock    (SOF_TIMESTAMPING_RAW_HARDWARE)
PTP Hardware Clock: 3
Hardware Transmit Timestamp Modes:
        off                   (HWTSTAMP_TX_OFF)
        on                    (HWTSTAMP_TX_ON)
Hardware Receive Filter Modes:
        none                  (HWTSTAMP_FILTER_NONE)
        ptpv1-l4-sync         (HWTSTAMP_FILTER_PTP_V1_L4_SYNC)
        ptpv1-l4-delay-req    (HWTSTAMP_FILTER_PTP_V1_L4_DELAY_REQ)
        ptpv2-event           (HWTSTAMP_FILTER_PTP_V2_EVENT)
```

After the host synchronizes with the Grandmaster clock, time in the containers gets aligned with the host machine time.

PTP requires the following kernel configuration options to be enabled:

- CONFIG_PPS
- CONFIG_NETWORK_PHY_TIMESTAMPING
- CONFIG_PTP_1588_CLOCK

5.1.5.1 Grandmaster Clock

On the system with the Grandmaster clock side, look at `the /etc/sysconfig/ptp4l` file (the last character is a lowercase **L**). It is the daemon configuration file that provides starting options to PTP. Its content should look like:

```
OPTIONS="-f /etc/ptp4l.conf -i <if_name>"
```

Where <if_name> is the interface name that will be used for time stamping and /etc/ptp4l.conf is the PTP4L configuration file.

PTP uses a BMC algorithm to choose a Grandmaster clock, and it isn't apparent to determine which timer is chosen by default. To specify a given timer as a Grandmaster clock, edit /etc/ptp4l.conf file, setting the priority1 property to **127**.

Then, start the ptp4l service.

```
service ptp4l start
Output from the service can be checked at /var/log/messages. Output for the master clock should look like:
Mar 16 17:08:57 localhost ptp4l: ptp4l[23627.304]: selected /dev/ptp2 as PTP clock
Mar 16 17:08:57 localhost ptp4l: [23627.304] selected /dev/ptp2 as PTP clock
Mar 16 17:08:57 localhost ptp4l: [23627.306] port 1: INITIALIZING to LISTENING on INITIALIZE
Mar 16 17:08:57 localhost ptp4l: ptp4l[23627.306]: port 1: INITIALIZING to LISTENING on INITIALIZE
Mar 16 17:08:57 localhost ptp4l: [23627.307] port 0: INITIALIZING to LISTENING on INITIALIZE
Mar 16 17:08:57 localhost ptp4l: ptp4l[23627.307]: port 0: INITIALIZING to LISTENING on INITIALIZE
Mar 16 17:08:57 localhost ptp4l: [23627.308] port 1: link up
Mar 16 17:08:57 localhost ptp4l: ptp4l[23627.308]: port 1: link up
Mar 16 17:09:03 localhost ptp4l: [23633.664] port 1: LISTENING to MASTER on ANNOUNCE_RECEIPT_TIMEOUT_EXPIRES
Mar 16 17:09:03 localhost ptp4l: ptp4l[23633.664]: port 1: LISTENING to MASTER on ANNOUNCE_RECEIPT_TIMEOUT_EXPIRES
Mar 16 17:09:03 localhost ptp4l: ptp4l[23633.664]: selected best master clock 001e67.fffe.d2f206
Mar 16 17:09:03 localhost ptp4l: ptp4l[23633.665]: assuming the grand master role
Mar 16 17:09:03 localhost ptp4l: [23633.664] selected best master clock 001e67.fffe.d2f206
Mar 16 17:09:03 localhost ptp4l: [23633.665] assuming the grand master role
```

The next step is to synchronize PHC timer to the system time, using phc2sys daemon.

1. Edit configuration file at /etc/sysconfig/phc2sys. Replace <if_name> with interface name.

   ```
   OPTIONS="-c <if_name> -s CLOCK_REALTIME -w"
   ```

2. Start phc2sys service.

   ```
   service phc2sys start
   ```

3. Logs can be viewed at /var/log/messages and look like:

   ```
   phc2sys[3656456.969]: Waiting for ptp4l...
   phc2sys[3656457.970]: sys offset -6875996252 s0 freq  -22725 delay   1555
   phc2sys[3656458.970]: sys offset -6875996391 s1 freq  -22864 delay   1542
   phc2sys[3656459.970]: sys offset       -52 s2 freq  -22916 delay   1536
   phc2sys[3656460.970]: sys offset       -29 s2 freq  -22909 delay   1548
   phc2sys[3656461.971]: sys offset       -25 s2 freq  -22913 delay   1549
   ```

4. The grandmaster clock is configured.

5.1.5.2 Non-Master clock

Non-master clock configuration is the same as for the Grandmaster clock except in /etc/ptp4l.conf the priority1 property value for ptp4l is the default value 128.

1. Run the ptp4l service.
2. To keep the system time synchronized to PHC time, change the phc2sys options in /etc/sysconfig/phc2sys to:

```
OPTIONS=''phc2sys -s <if_name> -w"
```

3. Replace <if_name> with the interface name.

```
Logs will be available at /var/log/messages.
phc2sys[28917.406]: Waiting for ptp4l...
phc2sys[28918.406]: phc offset -42928591735 s0 freq +24545 delay 1046
phc2sys[28919.407]: phc offset -42928611122 s1 freq +5162 delay 955
phc2sys[28920.407]: phc offset 308 s2 freq +5470 delay 947
phc2sys[28921.407]: phc offset 408 s2 freq +5662 delay 947
phc2sys[28922.407]: phc offset 394 s2 freq +5771 delay 947
```

After this, both clocks should be synchronized. Docker is using the same clock as the host so its clock will be synchronized as well.

5.2 Deploy FlexRAN Timer mode on Container through Kubernetes*

Run FlexRAN in Kubernetes, to create the Docker* image containing the necessary tools to run FlexRAN. In this scenario, the Docker image was a local image stored on the non-master host, but it could also be pushed and used from a private Docker repository. The configuration for the FlexRAN PODs is stored on the master node.

5.2.1 Generating LTE/5G Docker Images with pre-build FlexRAN

The following prerequisites must be organized within the directory used for building the Docker image. Here, Intel® System Studio and DPDK* are still based on the host due to the size being too big to put in the Docker image.

In the following steps untar the FlexRAN release package and the built binaries.

Follow the steps below to build a pre-build image:

1. Untar the FlexRAN release package to build binaries:

```
mkdir /root/flexran
cd /root/flexran
tar  -zxvf   <flexran-release-tarball>
./extract.sh
```

2. Build the FlexRAN LTE Testmac and l1app:

```
export isa=avx512
source ./set_env_var.sh -i ${isa}
export RTE_SDK=/opt/dpdk-4g
./flexran_build.sh -v -e -i ${isa} -r lte -b -m all
```

This will build all the binaries for LTE timer mode.

3. If you are building a Docker image for FlexRAN 5GNR testmac and l1app, follow the command below instead of the above Step.

```
export isa=avx512
source ./set_env_var.sh -i ${isa}
export RTE_SDK=/opt/dpdk-5g
#build 5gnr sub6
./flexran_build.sh -v -e -i ${isa} -r 5gnr_sub6 -b -m all
```

```
#or build 5gnr mmwave
./flexran_build.sh -v -e -i ${isa} -r 5gnr_mmw -b -m all
```

This will build all the binaries for 5GNR timer mode.

4. Create a FlexRAN directory in the Docker build directory and copy the FlexRAN package into it (assuming the package is at /root/flexran).

```
mkdir /root/FlexRAN_prebuild/
cp -r /root/flexran /root/FlexRAN_prebuild/flexran
```

Note: The FlexRAN folder is the source code and binaries from the FlexRAN release package. Here the flexran/tests folder is not built into Docker image because its size is larger than 9 GB, which will make the Docker images size too big. Instead, this test folder will be mounted from the host into the container. The Kubernetes master yaml file does the configuration. Below folders/files will be copied to the Docker image.

```
bin  framework  libs  ReadMe.txt  set_env_var.sh  xran  build  flexran_build.sh
Intel_OBL_Commercial_Use_License.txt  nfapi  misc  sdk  source  wls_mod
```

5. Create a Dockerfile to be used to build the Docker image.

```
vim /root/FlexRAN_prebuild/Dockerfile
```

With the following content:

Note: The proxy settings highlighted in bold font are optional and specific to your network configuration):

```
FROM centos:centos7.8.2003

ENV ftp_proxy <your proxy>
ENV https_proxy <your proxy>
ENV http_proxy <your proxy>
ENV no_proxy "localhost,127.0.0.1,192.168.0.100"

RUN yum update -y && yum install -y hugepages libhugetlbfs-utils lbhugetlbfs-devel
ibhugetlbfs numactl-devel ethtool gcc make g++ module-init-tools kmod wget patch xz-utils
iproute pciutils python vim cmake unzip nano mc iputils-ping libaio libaio-devel git net-
tools

WORKDIR /root/
COPY flexran ./flexran
COPY set-l1-env.sh ./
COPY set-l2-env.sh ./
```

The content of set-l1-env.sh:

```
#MODE will be set as “LTE” or “5G”
MODE=$1
cd /root
export WORKSPACE=`pwd`/flexran
export isa=avx512
cd $WORKSPACE
source ./set_env_var.sh -i ${isa}
source  /opt/intel/system_studio_2019/bin/iccvars.sh intel64
if [ $MODE = LTE ]; then
    cd $WORKSPACE/bin/lte/l1/
fi
if [ $MODE = 5G ]; then
    cd $WORKSPACE/bin/nr5g/gnb/l1
```

```
fi
```

The content of set-l2-env.sh:

```
MODE=$1
cd /root
export WORKSPACE=`pwd`/flexran
export isa=avx512
cd $WORKSPACE
source ./set_env_var.sh -i ${isa}
source  /opt/intel/system_studio_2019/bin/iccvars.sh intel64
if [ $MODE = LTE ]; then
     cd $WORKSPACE/bin/lte/testmac
fi
if [ $MODE = 5G ]; then
     cd $WORKSPACE/bin/nr5g/gnb/testmac
fi
```

6. Create the Docker image:

   For LTE:

   ```
   docker build -t flexran.docker.registry/flexran-lte:v1 .
   ```

   For 5GNR:

   ```
   docker build -t flexran.docker.registry/flexran-5g:v1 .
   ```

5.2.2 Creating FlexRAN Pods

Kubernetes can label the nodes in Kubernetes to create PODs designed for a specific node. In a case where multiple nodes are to be used, they can be labeled in the following way (not needed if only one worker node used, or the same node used for master/worker).

1. To label a node:

   ```
   kubectl label nodes worker1 testnode=worker-skl-sp
   ```

   Node labels can be checked as follows:

   ```
   kubectl get nodes --show-labels
   ```

   To deploy on a specific node the following should be added to .yaml specification files under "spec:" (name of node label as an example):

   ```
   spec:
     nodeSelector:
       testnode: worker-skl-sp
   ```

2. Create a specification file for the FlexRAN POD called flexran.yaml:

   ```
   apiVersion: v1
   kind: Pod
   metadata:
     labels:
       app: flexran-nr-pod
     name: flexran-nr-pod
   spec:
     containers:
     - securityContext:
   ```

```
    privileged: false
    capabilities:
      add:
        - SYS_ADMIN
        - IPC_LOCK
        - SYS_NICE
  command:
    - sleep
    - infinity
  tty: true
  stdin: true
  image: flexran.docker.registry/flexran-5g:v1
  name: flexran-container1
  resources:
    requests:
      memory: "4Gi"
      hugepages-1Gi: 4Gi
    limits:
      memory: "4Gi"
      hugepages-1Gi: 4Gi
  volumeMounts:
  - name: hugepage
    mountPath: /hugepages
  - name: varrun
    mountPath: /var/run/dpdk
    readOnly: false
  - name: sys
    mountPath: /sys/
    readOnly: false
  - name: usrpath
    mountPath: /usr/
    readOnly: true
  - name: icc-path
    mountPath: /opt/intel/system studio 2019
    readOnly: true
  - name: flexran
    mountPath: /root/flexran/
    readOnly: false
volumes:
- name: dpdk
  hostPath:
    path: "/opt/dpdk-5g"
- name: flexran
  hostPath:
    path: "/opt/flexran"
- name: sys
  hostPath:
    path: "/sys"
- name: hugepage
  emptyDir:
    medium: HugePages
```

```
  - name: varrun
    emptyDir: {}
  - name: usrpath
    hostPath:
      path: "/usr"
  - name: icc-path
    hostPath:
      path: "/opt/intel/system_studio_2019"
```

3. Create the POD:

```
kubectl create -f flexran.yaml
```

4. Check if the POD is up and running:

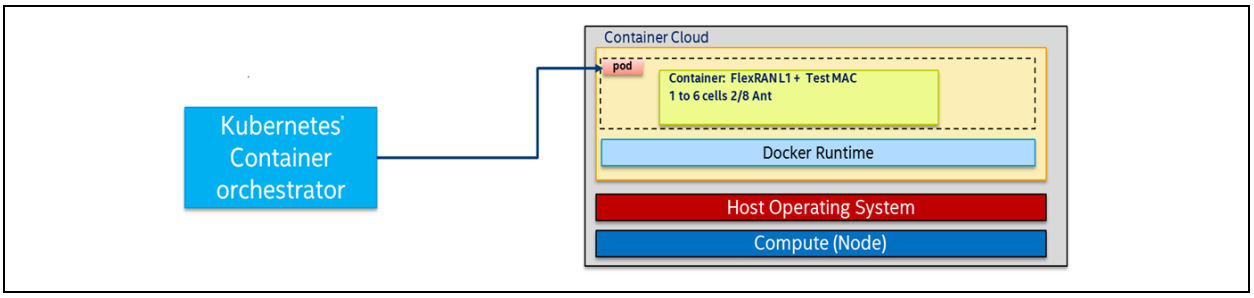
```
kubectl get pods
```

When the process is completed, the Pod should show a status – running – next to its name. This indicates that the Pod has been deployed.

5.2.3 Testing FlexRAN Inside a Kubernetes POD (Single Container)

The diagram in Figure 19 demonstrates a K8s POD containing a single container, running on top of the bare metal host.

Figure 19. FlexRAN in Single Kubernetes POD



Two terminals are required to test FlexRAN inside a container.

1. Start the terminal program in two separate windows.
2. In the first terminal, run the following command:

```
kubectl exec -it flexran-lte-pod  -- /bin/bash
```

This command will prompt the FlexRAN container from within the POD:

```
source set-l1-env.sh
./l1.sh -e
```

3. In the second terminal, run the following command:

```
kubectl exec -it flexran-lte-pod -- /bin/bash
```

This command will prompt the FlexRAN container from within the Pod:

```
source set-l2-env.sh
./l2.sh --testfile=skylake-sp/sklsp_p5_htoff_fecon.cfg
```

Note: The name of the test file may vary depending on the platform, hyperthreading, or FEC emulation scenario. To avoid the "zombie" processes, always exit the tests with the exit command.

5.2.4 Setting up FlexRAN Inside K8s POD with L1 and L2/L3 in Separate Containers

FlexRAN can be run with L1, and L2/L3 separated into different containers inside the same K8s POD.

1. Create a configuration file for the FlexRAN POD named flexranSplit.yaml:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: flexran-nr-pod
  name: flexran-nr-pod
spec:
  containers:
  - securityContext:
      privileged: false
      capabilities:
        add:
          - SYS_ADMIN
          - IPC_LOCK
          - SYS_NICE
    command:
      - sleep
      - infinity
    tty: true
    stdin: true
    image: flexran.docker.registry/flexran-5g:v1
    name: flexran-container1
    resources:
      requests:
        memory: "4Gi"
        hugepages-1Gi: 3Gi
      limits:
        memory: "4Gi"
        hugepages-1Gi: 3Gi
    volumeMounts:
    - name: hugepage
      mountPath: /hugepages
    - name: varrun
      mountPath: /var/run/dpdk
      readOnly: false
    - name: sys
      mountPath: /sys/
      readOnly: false
    - name: usrpath
      mountPath: /usr/
      readOnly: true
    - name: icc-path
      mountPath: /opt/intel/system_studio_2019
      readOnly: true
    - name: flexran
```

```
    mountPath: /root/flexran/
    readOnly: false
- securityContext:
    privileged: false
    capabilities:
      add:
        - SYS_ADMIN
        - IPC_LOCK
        - SYS_NICE
  command:
    - sleep
    - infinity
  tty: true
  stdin: true
  image: flexran.docker.registry/flexran-5g:v1
  name: flexran-container2
  resources:
    requests:
      memory: "1Gi"
    limits:
      memory: "1Gi"
  volumeMounts:
  - name: hugepage
    mountPath: /hugepages
  - name: varrun
    mountPath: /var/run/dpdk
    readOnly: false
  - name: sys
    mountPath: /sys/
    readOnly: false
  - name: usrpath
    mountPath: /usr/
    readOnly: true
  - name: icc-path
    mountPath: /opt/intel/system_studio_2019
    readOnly: true
  - name: flexran
    mountPath: /root/flexran/
    readOnly: false
volumes:
- name: dpdk
  hostPath:
    path: "/opt/dpdk-5g"
- name: flexran
  hostPath:
    path: "/opt/flexran"
- name: sys
  hostPath:
    path: "/sys"
- name: hugepage
```

```
    emptyDir:
      medium: HugePages
  - name: varrun
    emptyDir: {}
  - name: usrpath
    hostPath:
      path: "/usr"
  - name: icc-path
    hostPath:
      path: "/opt/intel/system_studio_2019"
```

2. Create the POD containing two containers:

```
kubectl create -f flexranSplit.yaml
```

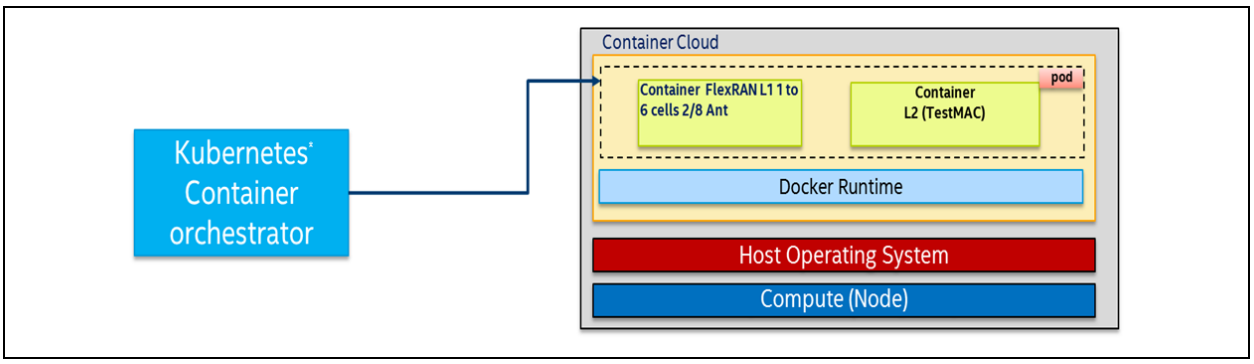
On a successful run, a POD with two running containers indicated by the status – Ready – and 2/2 containers running inside the PODs can be seen.

```
kubectl get pods
```

5.2.5 Testing FlexRAN Inside Kubernetes POD (1 POD, Multiple Containers)

The diagram in Figure 21 illustrates a K8s* POD containing multiple containers, running on top of a bare-metal host.

Figure 20. FlexRAN in Kubernetes POD with Multiple Containers



Two terminals are needed to test FlexRAN running within two separate containers for L1 and L2/L3.

1. In the first terminal, run the following command in container number one:

```
kubectl exec -it flexran-lte-pod -c flexran-container1 -- /bin/bash
```

2. In the second terminal, run the following command in container number two:

```
kubectl exec -it flexran-lte-pod -c flexran-container2 -- /bin/bash
```

3. Now that two separate terminals are running in two separate containers, in the first terminal/container, run the following commands:

```
source set-l1-env.sh
./l1.sh -e
```

4. In the second terminal/container, run these commands:

```
source set-l2-env.sh
./l2.sh --testfile=skylake-sp/sklsp_p5_htoff_fecon.cfg
```

Note: The name of the test file may vary depending on the platform, hyperthreading, or FEC emulation scenario. To avoid the "zombie" processes, always exit the tests with the exit command.

On successful completion, communication between L1 and L2 can be seen.

5.2.6 Starting Multiple FlexRAN PODs (LTE POD with 5G POD)

To start multiple FlexRAN PODs, a flexran-lte(5g)-prebuild.yaml specification file must be created for each POD, which allows for the use of FlexRAN prebuild Docker image.

1. Create .yaml specification files

   Specification 1:

```
vim flexran-lte-prebuild.yaml
```

   With the following content:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: flexran-lte-testmac
  name: flexran-lte-testmac
spec:
  containers:
  - securityContext:
      privileged: false
      capabilities:
        add:
          - SYS_ADMIN
          - IPC_LOCK
          - SYS_NICE
    command:
      - sleep
      - infinity
    tty: true
    stdin: true
    image: flexran.docker.registry/flexran-lte:v1
    name: flexran-container1
    resources:
      requests:
        memory: "4Gi"
        hugepages-1Gi: 3Gi
      limits:
        memory: "4Gi"
        hugepages-1Gi: 3Gi
    volumeMounts:
    - name: hugepage
      mountPath: /hugepages
    - name: varrun
      mountPath: /var/run/dpdk
      readOnly: false
```

```
    - name: icc-path
      mountPath: /opt/intel/system_studio_2019
      readOnly: true
    - name: test-path
      mountPath: /root/flexran/tests
      readOnly: false
  - securityContext:
      privileged: false
      capabilities:
        add:
          - SYS_ADMIN
          - IPC_LOCK
          - SYS_NICE
    command:
      - sleep
      - infinity
    tty: true
    stdin: true
    image: flexran.docker.registry/flexran-lte:v1
    name: flexran-container2
    resources:
      requests:
        memory: "1Gi"
      limits:
        memory: "1Gi"
    volumeMounts:
    - name: hugepage
      mountPath: /hugepages
    - name: varrun
      mountPath: /var/run/dpdk
      readOnly: false
    - name: icc-path
      mountPath: /opt/intel/system_studio_2019
      readOnly: true
    - name: test-path
      mountPath: /root/flexran/tests
      readOnly: false
  volumes:
  - name: hugepage
    emptyDir:
      medium: HugePages
  - name: varrun
    emptyDir: {}
  - name: icc-path
    hostPath:
      path: "/opt/intel/system_studio_2019"
  - name: test-path
    hostPath:
      path: "/root/flexran/tests"
```

Specification 2:

```
vim flexran-5g-prebuild.yaml
```

With the following content:

```
vim flexran-5g-prebuild.yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: flexran-nr-testmac
  name: flexran-nr-testmac
spec:
  containers:
  - securityContext:
      privileged: false
      capabilities:
        add:
          - SYS_ADMIN
          - IPC_LOCK
          - SYS_NICE
    command:
      - sleep
      - infinity
    tty: true
    stdin: true
    image: flexran.docker.registry/flexran-5g:v1
    name: flexran-container1
    resources:
      requests:
        memory: "16Gi"
        intel.com/intel_fpga: '1'
        hugepages-1Gi: 2Gi
      limits:
        memory: "16Gi"
        intel.com/intel fpga: '1'
        hugepages-1Gi: 2Gi
    volumeMounts:
    - name: hugepage
      mountPath: /hugepages
    - name: varrun
      mountPath: /var/run/dpdk
      readOnly: false
    - name: icc-path
      mountPath: /opt/intel/system_studio_2019
      readOnly: true
    - name: test-path
      mountPath: /root/flexran/tests
      readOnly: false
  - securityContext:
      privileged: false
      capabilities:
        add:
```

```
        - SYS_ADMIN
        - IPC_LOCK
        - SYS_NICE
    command:
      - sleep
      - infinity
    tty: true
    stdin: true
    env:
    image: flexran.docker.registry/flexran-5g:v1
    name: flexran-container2
    resources:
      requests:
        memory: "1Gi"
      limits:
        memory: "1Gi"
    volumeMounts:
    - name: hugepage
      mountPath: /hugepages
    - name: varrun
      mountPath: /var/run/dpdk
      readOnly: false
    - name: icc-path
      mountPath: /opt/intel/system_studio_2019
      readOnly: true
    - name: test-path
      mountPath: /root/flexran/tests
      readOnly: false
  volumes:
  - name: hugepage
    emptyDir:
      medium: HugePages
  - name: varrun
    emptyDir: {}
  - name: icc-path
    hostPath:
      path: "/opt/intel/system_studio_2019"
  - name: test-path
    hostPath:
      path: "/root/flexran/tests"
```

2. Create two FlexRAN PODs with the aid of the created specification files:

```
kubectl create -f flexran-lte-prebuild.yaml
kubectl create -f flexran-5g-prebuild.yaml
```

Note: Enable LTE/5G NR FEC FPGA Device Plugin following section Device Plugin Usage before create lte/5g POD.

3. Check that both PODs are running

```
kubectl get pods

Expected result:
```

```
NAME              READY     STATUS    RESTARTS   AGE
flexran-lte-testmac    2/2       Running   0          30s
flexran-5g-testmac     2/2       Running   0          9s
```

4. Execute into two PODs, and start the L1 app and testmac app in each POD.

Note: The core allocation, wls_dev_name, dpdkBasebandFecMode for each FlexRAN instance, must be set manually in phycfg_timer.xml, testmac_cfg.xml, and test configuration files. The "Vista Creek or Mount Bryce VF address must be set manually in phycfg_timer.xml for 5G POD.

5.2.6.1 LTE POD

1. From separate terminals, execute into the two containers of LTE POD.

```
kubectl exec -it flexran-lte-testmac -c flexran-container1 -- /bin/bash
kubectl exec -it flexran-lte-testmac -c flexran-container2 -- /bin/bash
```

2. From container 1, execute into the l1 directory.

```
source set-l1-env.sh LTE
```

3. From container 1, start L1.

```
./l1.sh -e
```

4. From container 2, execute into the testmac directory.

```
source set-l2-env.sh LTE
```

5. From container 2, start L2.

```
./l2.sh --testfile=skylake-sp/sklsp_p2_htoff_fecon.cfg
```

5.2.6.2 5G POD

1. From separate terminals, execute into the two containers of 5G POD.

```
kubectl exec -it flexran-5g-testmac -c flexran-container1 -- /bin/bash
kubectl exec -it flexran-5g-testmac -c flexran-container2 -- /bin/bash
```

2. From container 1, execute into the l1 directory.

```
export LD_LIBRARY_PATH=/root/flexran/libs/cpa/sub6/rec/lib/lib:/opt/intel/
  Node: if 5G mmware use /root/flexran/libs/cpa/mmw/rec/lib/lib:/opt/intel/
source set-l1-env.sh 5G
```

3. From container 1, start L1.

```
./l1.sh -e
```

4. From container 2, execute into the testmac directory.

```
source set-l2-env.sh 5G
```

5. From container 2, start L2.

```
./l2.sh --testfile=cascade_lake-sp/csxsp_mu0_20mhz_htoff.cfg
```

Upon success, both PODs will run an instance of L1/testmac.

5.2.7 Starting Multiple FlexRAN PODs (multiple 5G POD)

To start multiple 5G PODs, it is different with scenario showed in chapter 5.2.6. Some extra configuration needed. Following chapters give these extra configurations:

5.2.7.1 Enable multi GNB on one worker node

This section explains how to enable multi FlexRAN application on one worker node . The setup instructions can be found from the below. Please prepare the docker image and yaml files for creating the pod.

4. Prepare multi yaml files for the FlexRAN ,an example below

```
cp ${yaml_1.yaml} ${yaml_2.yaml}
```

Only change the pod name to let two separated pod for multi FlexRAN, so the two pods will be(example)

```
 flexran-5g-1
 flexran-5g-2
```

5. Create pods for FlexRAN through the two yaml files.

```
kubectl create -f ${yaml_1}
kubectl create -f ${yaml_2}
```

6. Make sure the pods are working and on the same worker node

```
kubectl get pods
     flexran-5g-1 2/2     Running   0          51m
     flexran-5g-2 2/2     Running   0          3h52m
kubectl describe pod flexran-5g-1 | grep Node:
     Node:         ${node name}/${node ip}
kubectl describe pod flexran-5g-2 | grep Node:
     Node:         ${node name}/${node ip}
```

5.2.7.2 Config through the config file

1. Get into the pods and config for the one FlexRAN applications , and check the

```
kubectl exec -it  flexran-5g-1 -c flexran-container1 /bin/bash
grep <dpdkFilePrefix> /root/flexran_l1_sw/bin/nr5g/gnb/l1/phycfg_timer.xml
                <dpdkFilePrefix>gnb0</dpdkFilePrefix>
```

2. Check the testmac config for the first FlexRAN application

```
kubectl exec -it  flexran-5g-1 -c flexran-container2 /bin/bash
grep <dpdkFilePrefix> /root/flexran_l1_sw/bin/nr5g/gnb/testmac/testmac_cfg.xml
                <dpdkFilePrefix>gnb0</dpdkFilePrefix>
```

3. Open another TTY config the `dpdkFilePrefix` to enable the scend FlexRAN application

```
kubectl exec -it  flexran-5g-2 -c flexran-container1 /bin/bash
grep <dpdkFilePrefix> /root/flexran_l1_sw/bin/nr5g/gnb/l1/phycfg_timer.xml
                <dpdkFilePrefix>gnb0</dpdkFilePrefix>
```

Change the second FlexRAN application to another dpdkFilePrefix.

```
sed -i 's/gnb0/gnb1/g'
/root/flexran_l1_sw/bin/nr5g/gnb/l1/phycfg_timer.xml
```

4. Also change the testmac config file to another `dpdkFilePrefix`

```
kubectl exec -it  flexran-5g-2 -c flexran-container2 /bin/bash
grep <dpdkFilePrefix> /root/flexran_l1_sw/bin/nr5g/gnb/testmac/testmac_cfg.xml
                <dpdkFilePrefix>gnb0</dpdkFilePrefix>
```

Change the second testmac application to another dpdkFilePrefix.

```
sed -i 's/gnb0/gnb1/g'
/root/flexran_l1_sw/bin/nr5g/gnb/testmac/testmac_cfg.xml
```

5.2.7.3 Run timer mode on multi FlexRAN in one worker node

1. Run the first FlexRAN

```
kubectl exec -it  flexran-5g-1 -c flexran-container1 /bin/bash
cd /root/flexran_l1_sw/bin/nr5g/gnb/l1
./l1.sh -e
…
…
Calling rte_eal_init: ./l1app -c 4 -n2 --file-prefix=gnb0 --socket-mem=6144 --socket-
limit=6144 -a0000:00:00.0 -a0000:1f:00.4 --iova-mode=pa


kubectl exec -it  flexran-5g-1 -c flexran-container2 /bin/bash
cd /root/flexran_l1_sw/bin/nr5g/gnb/testmac
./l2.sh
…
…
Calling rte_eal_init: testmac -c1 --proc-type=auto --file-prefix gnb0 --iova-mode=pa
```

2. Run the second FlexRAN

```
kubectl exec -it  flexran-5g-2 -c flexran-container1 /bin/bash
cd /root/flexran_l1_sw/bin/nr5g/gnb/l1
./l1.sh -e
…
…
Calling rte_eal_init: ./l1app -c 4 -n2 --file-prefix=gnb1 --socket-mem=6144 --socket-
limit=6144 -a0000:00:00.0 -a0000:1f:00.4 --iova-mode=pa


kubectl exec -it  flexran-5g-2 -c flexran-container2 /bin/bash
cd /root/flexran_l1_sw/bin/nr5g/gnb/testmac
./l2.sh
…
…
Calling rte_eal_init: testmac -c1 --proc-type=auto --file-prefix gnb1 --iova-mode=pa
```

6.0 FlexRAN run on container through CIR*

This chapter will give an automation method – CIR to deploy FlexRAN timer mode and radio mode E2E solution.

6.1 About CIR

Common Integration Repository (CIR) aims to supply easy orchestration and application deployment in cloud network and edge environment on Intel architecture platform. It provides tarball package of ansible playbooks for automating installation and configuration of ingredients from single source which can be deployed on supported Intel BKC platforms. With CIR, users can setup network cluster environment to verify application deployment easily, and internal ingredients can verify the system robustness and compatibility with easy installation and scaling management.

6.1.1 Ansible Host Prerequisites

Minimal three hosts/servers are needed in CIR deployment, one is for Ansible host to running CIR script, the other two are for Kubernetes cluster setup. Below are the steps need to be run on Ansible host firstly:

1. Prepare ansible host environment with following commands:

```
sudo yum install epel-release
sudo yum install ansible
easy_install pip
pip install jinja2 –upgrade
pip install netaddr
```

2. Enable passwordless login between all nodes in the cluster.

```
ssh-keygen
ssh-copy-id root@node-ip-address
```

6.1.2 Get CIR Package

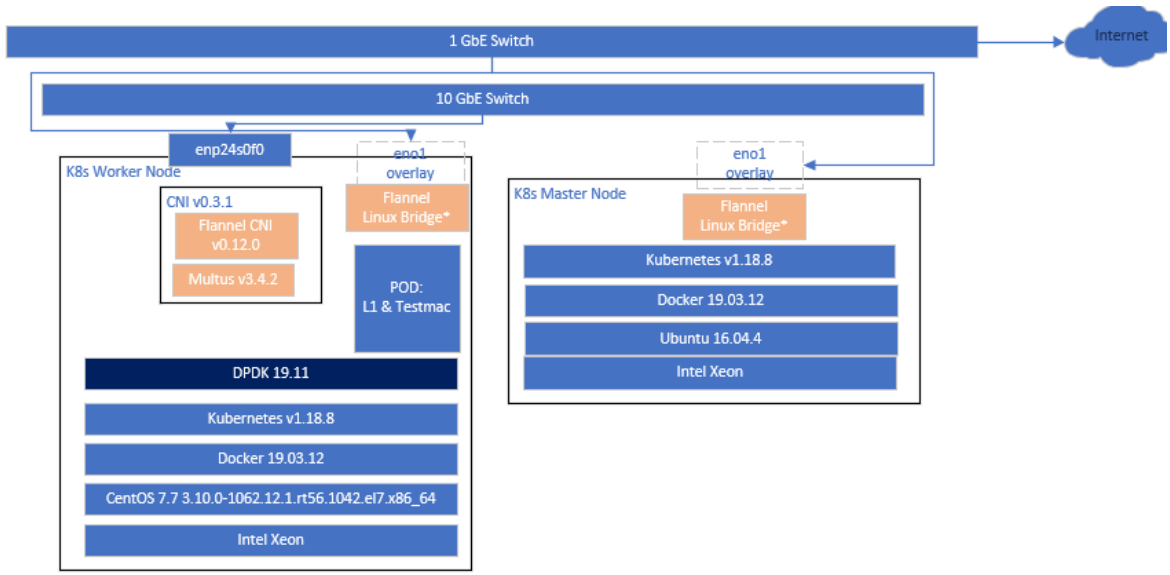
clone CIR repo to get CIR ansible scripts:

```
git clone
https://gitlab.devtools.intel.com/system_integration/common_integration_repository.git
cd common_integration_repository
```

6.2 Deploy FlexRAN Timer mode on Container through CIR

6.2.1 FlexRAN Timer Mode Topology

Figure 21. FlexRAN Timer Mode Topology



6.2.2 FlexRAN Timer Mode CIR Configuration

1. copy access_timer_inventory.ini to CIR folder and edit it with correct ips:

   cp examples/ flexran/access_timer_inventory.ini ./
   vi access_timer_inventory.ini
   Here is an example:

```
[all]
flexran-master ansible_host=10.240.224.106 ip=10.240.224.106 ansible_ssh_user=root
ansible_ssh_pass=root
flexran-node   ansible_host=10.240.224.241 ip=10.240.224.241 ansible_ssh_user=root
ansible ssh pass=root123
localhost

[kube-master]
flexran-master

[etcd]
flexran-master

[kube-node]
flexran-node

[k8s-cluster:children]
kube-master
kube-node
```

```
[calico-rr]

[all:vars]
ansible_python_interpreter=/usr/bin/python2.7
```

2. copy group_vars and host_vars to CIR folder, update below settings:

   - cp examples/flexran/access/group_vars examples/flexran/access/host_vars -rf ./
   - vi group_vars/all/all.yml and add the proxy setting and update some plugin settings as below:

```
…
Proxy configuration ##
http_proxy: "http://proxy-example.com"
https_proxy: "http://proxy-example.com"
additional_no_proxy: "127.0.0.1, all master and minion's ip listed here which is seperated
with comma"

cmk_enabled: false
sriov_net_dp_enabled: false
qat_dp_enabled: false
…
```

   - vi group_vars/all/usecase.yml for below settings: (a http share folder can be setup to store those flexran, icc release package and binary packages)

```
…
# Common configuration for both timer and E2E mode
#Intel C Compiler installer
icc_installer_url: http://ons-
archive.jf.intel.com/share/flexran/icc/system_studio_2019_update_3_composer_edition_offline.t
ar.gz
#Intel C Compiler license
icc_license_url: http://ons-archive.jf.intel.com/share/flexran/icc/license.lic
#FlexRAN SDK package
flexran_sdk_url: http://ons-archive.jf.intel.com/share/flexran/FlexRAN-20.08/FlexRAN-20.08-
FlexRAN_FULL-183.tar.gz
#BBDEV patch for FlexRAN
dpdk_bbdev_patch_url:
http://ons-archive.jf.intel.com/share/flexran/FlexRAN-20.08/dpdk_bbdev_19.11_20.08_rc2.patch

# FlexRAN timer mode specific configuration
flexran_timer_enabled: true

# FlexRAN E2E mode specific configuration
flexran_e2e_enabled: false
…
```

   - cp host_vars/node1.yml host_vars/flexran_node.yml (run multiple times if you have multiple nodes defined)
   - vi host_vars/ flexran_node.yml, update below settings according to your deployment.

```
…
update_qat_drivers: false
default_hugepage_size: 1G
hugepages_1G: 50
…
```

6.2.3 Deploy FlexRAN Timer Mode through CIR

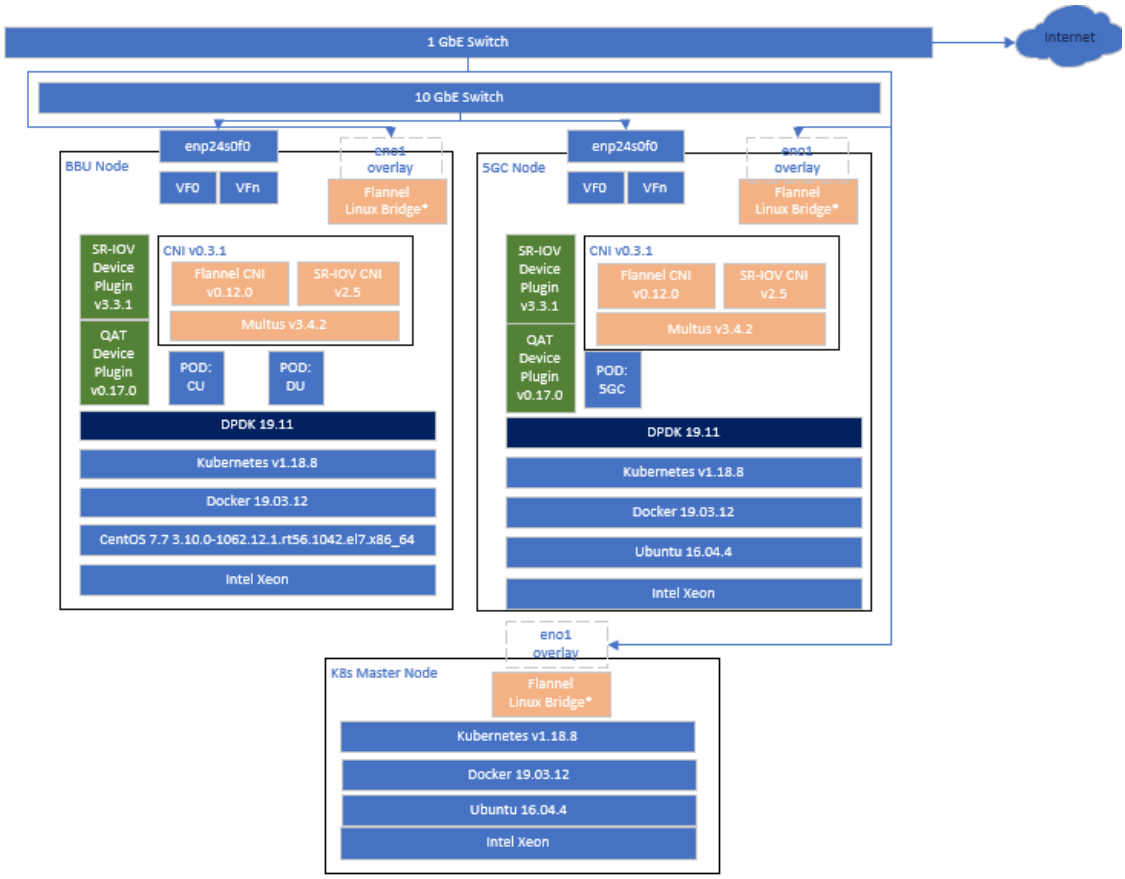
Run below command on Ansible host to deploy FlexRAN timer mode through CIR:

```
ansible-playbook -i access_e2e_inventory.ini playbooks/cir.yml --extra-vars "profile=access"
```

6.3 Deploy FlexRAN Radio mode (E2E) on Container through CIR

6.3.1 FlexRAN Radio Mode Topology

Figure 22. FlexRAN Radio Mode Topology



6.3.2 FlexRAN Radio Mode CIR Configuration

1. copy access_timer_inventory.ini to CIR folder and edit it with correct ips:

   cp examples/ flexran/access_timer_inventory.ini ./
   vi access_timer_inventory.ini

Here is an example:

```
[all]
flexran-v2x-cir-master ansible_host=10.240.224.36 ip=10.240.224.36 ansible_ssh_user=root
ansible_ssh_pass=npg
flexran-v2x-cir-bbu ansible_host=10.240.224.38 ip=10.240.224.38 ansible_ssh_user=root
ansible_ssh_pass=npg
flexran-v2x-cir-5gc ansible_host=10.240.224.37 ip=10.240.224.37 ansible_ssh_user=root
ansible_ssh_pass=npg ansible_python_interpreter=/usr/bin/python3
localhost

[kube-master]
flexran-v2x-cir-master

[etcd]
flexran-v2x-cir-master

[kube-node]
flexran-v2x-cir-bbu
flexran-v2x-cir-5gc

[k8s-cluster:children]
kube-master
kube-node

[flexran-bbu]
flexran-v2x-cir-bbu

[flexran-5gc]
flexran-v2x-cir-5gc

[calico-rr]

[workload:children]
k8s-cluster

[all:vars]
ansible_python_interpreter=/usr/bin/python2.7
```

2. copy group_vars and host_vars to CIR folder, update proxy and node configuration file name:
   - cp examples/flexran/access/group_vars examples/flexran/access/host_vars -rf ./
   - vi group_vars/all/all.yml and update setting as below according to your deployment:

```
…
Proxy configuration ##
http_proxy: "http://proxy-example.com"
https_proxy: "http://proxy-example.com"
additional_no_proxy: "127.0.0.1, all master and minion's ip listed here which is seperated
with comma"

sriovdp_config_data: |
    {
        "resourceList": [{
                "resourceName": "intel_sriov_1G",
                "selectors": {
```

```
                    "vendors": ["8086"],
                    "devices": ["1520"],
                    "drivers": ["i40evf", "igbvf", "iavf", "vfio-pci"],
                    "pfNames": ["enp136s0f1"]
                }
            },
            {
                "resourceName": "intel_sriov_10G",
                "selectors": {
                    "vendors": ["8086"],
                    "devices": ["154c"],
                    "drivers": ["i40evf", "iavf", "vfio-pci"],
                    "pfNames": ["enp24s0f0"]
                }
            },
            {
                "resourceName": "intel_sriov_40G",
                "selectors": {
                    "vendors": ["8086"],
                    "devices": ["154c"],
                    "drivers": ["iavf", "vfio-pci"],
                    "pfNames": ["enp134s0f0"]
                }
            },
            {
                "resourceName": "intel_fpga",
                "deviceType": "accelerator",
                "selectors": {
                    "vendors": ["8086"],
                    "devices": ["0d90"]
                }
            }
        ]
    }
…
```

- sriovdp_config_data: update to the real configuration according to your deployment.
  - cp host_vars/node1.yml host_vars/ flexran-v2x-cir-bbu.yml && cp host_vars/node1.yml host_vars/ flexran-v2x-cir-5gc.yml
  - vi host_vars/ flexran-v2x-cir-bbu.yml, update below settings according to your deployment.

```
dataplane_interfaces:
  - name: enp30s0f0                  # 40G interface name
    bus_info: "1e:00.0"              # pci bus info
    sriov_numvfs: 2                  # number of VFs to create for this PF
    vf_driver: vfio-pci             # VF driver to be attached for all VFs under this PF,
"iavf", "vfio-pci", "igb_uio"

  - name: enp175s0f0              #10G to 5GC
    bus_info: "af:00.0"
    sriov_numvfs: 2
    vf_driver: iavf
```

```
- name: eno1
  bus_info: "3f:00.0"
  sriov_numvfs: 10
  vf_driver: iavf
```

6.3.3 Deploy FlexRAN Radio mode* through CIR

Run below command on Ansible host to deploy FlexRAN through CIR:

```
ansible-playbook -i access_e2e_inventory.ini playbooks/cir.yml --extra-vars "profile=access"
```

7.0 FlexRAN run on container with VMware

This chapter provide instructions on how to create a FlexRAN Docker image on VMware Photon OS to deploy on VMware Telco Cloud Automation (TCA) cluster. About VMware TCP-RAN

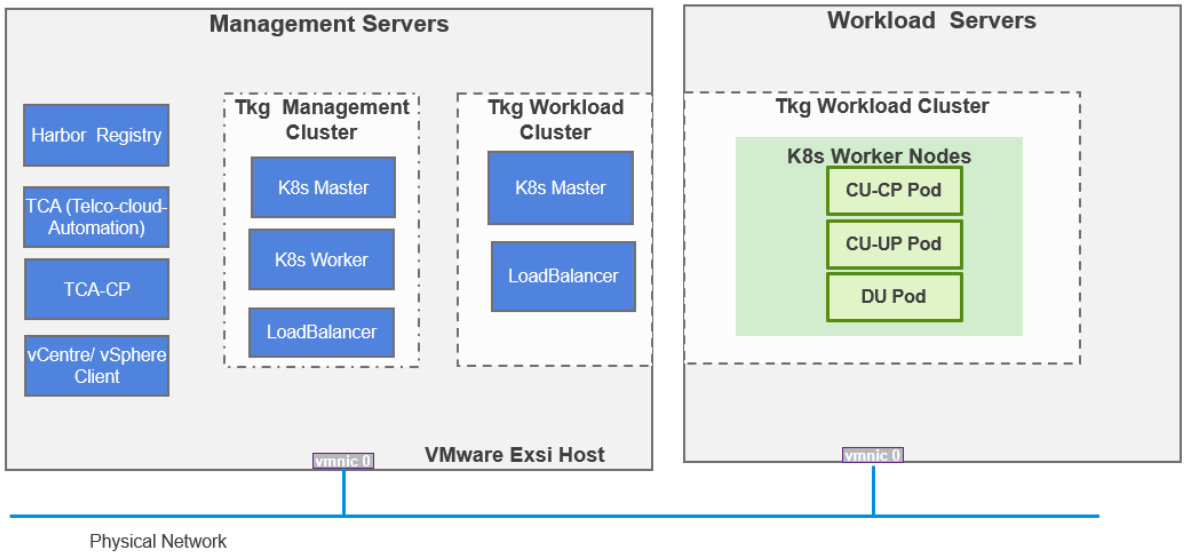
VMware Telco Cloud Platform RAN™ is a cloud-native RAN solution that is designed specifically for running RAN functions. It provides the RAN modernization path, evolving from legacy Radio Access Network (RAN) to virtualized RAN (vRAN) to OpenRAN. It transforms the RAN into a 5G multi-services hub “mini cloud”, enabling Communication Services Providers (CSPs) to monetize their RAN investments. VMware Telco Cloud Platform RAN is designed to meet the performance and latency requirements inherent to RAN workloads.

VMware TCP-RAN consists of three components:

1. VMware Telco Automation (TCA)
    - VMware Telco Cloud Manager - Provides Telco’s with NFV-MANO capabilities and enables the automation of deployment and configuration of Network Functions and Network Services.
    - VMware Telco Cloud Automation Control Plane (TCA-CP) - Provides the infrastructure for placing workloads across clouds using VMware Telco Cloud Automation.
2. VMware Tanzu, Kubernetes’s
3. VMware vSphere, hypervisor

Note: Configuration of TCA and its associated components is out of the scope of this document. The instructions below assumes the user already has the TCA, Harbor Repository, Vcenter, and K8S worker node deployed.

Figure 23. 3: High Level View of VMware Deployment Topology



7.1 Generating FlexRAN Docker Image for VMware K8S

This section will provide instructions to create a FlexRAN Docker image to run on VMware Photon OS.

In order to complete all the steps below to build and load the FlexRAN container onto VMware K8S environment, the following components are required:

- ESXi software for Controller and Worker Nodes
- VMware vCenter Server Appliance
- VMware Harbor Repository
- Telco Cloud Automation Software
- Photon OS with RT-kernel

To keep the Docker image manageable, this guide will only build the Photon-RT OS base image and user will mount the ICC and FlexRAN L1 package from the Worker Node.

7.1.1 Compiling FlexRAN on VMware Worker Node

Please following Section 5.2.1 to compile FlexRAN on Photon OS build server. Will repeat steps below for clarity.

7.1.2 Generating LTE/5G Docker Images with pre-build FlexRAN

The following prerequisites must be organized within the directory used for building the Docker image. Here, Intel® System Studio and DPDK* are still based on the host due to the size being too big to put in the Docker image.

In the following steps untar the FlexRAN release package and the built binaries.

Follow the steps below to build a pre-build image:

1. Untar the FlexRAN release package to build binaries:

```
mkdir /root/flexran
cd /root/flexran
tar  -zxvf   <flexran-release-tarball>
./extract.sh
```

2. Build the FlexRAN LTE Testmac and l1app:

```
export isa=avx512
source ./set_env_var.sh -i ${isa}
export RTE_SDK=/opt/dpdk-4g
./flexran_build.sh -v -e -i ${isa} -r lte -b -m all
```

This will build all the binaries for LTE timer mode.

3. If you are building a Docker image for FlexRAN 5GNR testmac and l1app, follow the command below instead of the above Step.

```
export isa=avx512
source ./set_env_var.sh -i ${isa}
export RTE_SDK=/opt/dpdk-5g
#build 5gnr sub6
./flexran_build.sh -v -e -i ${isa} -r 5gnr_sub6 -b -m all
```

```
#or build 5gnr mmwave
./flexran_build.sh -v -e -i ${isa} -r 5gnr_mmw -b -m all
```

This will build all the binaries for 5GNR timer mode.

Note: The FlexRAN folder is the source code and binaries from the FlexRAN release package. Here the flexran/tests folder is not built into Docker image because its size is larger than 9 GB, which will make the Docker images size too big. Instead, this test folder will be mounted from the host into the container. The Kubernetes master yaml file does the configuration.

4. Create a Dockerfile to be used to build the Photon OS base Docker image.

```
vim /root/photon_os/Dockerfile
```

```
FROM docker.io/photon:latest

http_proxy: "http://proxy-example.com"
https_proxy: "http://proxy-example.com"
additional_no_proxy: "127.0.0.1, all master and minion's ip listed here which is seperated
with comma"

RUN tdnf install -y \
    sudo \
    gcc \
    glib \
    awk \
    coreutils \
    make \
    cmake \
    binutils \
    util-linux \
    linux-api-headers \
    glibc-devel \
    libhugetlbfs-devel \
    libnuma-devel

WORKDIR /root/

#install kernel sources to compile DPDK
RUN tdnf install -y linux-rt-4.19.177-rt72-2.ph3-rt linux-rt-devel-4.19.177-rt72-2.ph3.x86_64
linux-tools ncurses-devel zlib-devel binutils-devel elfutils-libelf-devel bc libstdc++
libstdc++-devel
```

5. Create the Docker image:

For Base Photon OS w/o FlexRAN:

```
docker build -t flexran.docker.registry/photon-rt-os-only:v1 .
```

7.1.3 Creating FlexRAN Pods

For simplification, we will create a single pod with mounted folder to ICC and FlexRAN release package on the pod. This will be run on the base Photon RT OS Docker image created in the section above

1. Pod YAML file:

```
apiVersion: v1
kind: Pod
metadata:
  name: flexran-pod
  annotations:
    k8s.v1.cni.cncf.io/networks: '[
            ]'
spec:
  containers:
  - name: flexran-pod
    image: flexran.docker.registry/photon-rt-os-only:v1
    command:
      - "sleep"
      - "99999999d"
    volumeMounts:
      - name: hugepage
        mountPath: /hugepages
      - name: sys
        mountPath: /sys/
      - name: dev
        mountPath: /dev/
      - name: icc
        mountPath: /opt/intel_2019/system_studio_2019
      - name: flx
        mountPath: /root/flexran
    resources:
      limits:
        cpu: 20
        memory: 40Gi
        hugepages-1Gi: 16Gi
      requests:
        cpu: 20
        memory: 40Gi
        hugepages-1Gi: 16Gi
    securityContext:
      privileged: true
  restartPolicy: Always
  volumes:
    - name: hugepage
      emptyDir:
        medium: HugePages
    - name: sys
      hostPath:
        path: /sys
    - name: dev
      hostPath:
        path: /dev
    - name: icc
      hostPath:
        path: /opt/intel_2019/system_studio_2019
    - name: flx
```

```
    hostPath:
      path: /root/flexran
```